VIRTUAL UNIVERSITY OF PAKISTAN

Web Design and Development

CS506

Course Handouts

GRADUP

Practice. Pass. Graduate.

thegradup.com

Lecture 1: Java Features 4
Lecture 2: Java Virtual Machine & Runtime Environment 8
Lecture 3: Learning Basics 19
Lecture 4: Object Oriented Programming 28
Lecture 5: Inheritance 38
Lecture 6: Collections 45
Lecture 7: Intro to Exceptions 54
Lecture 8: Streams 68
Lecture 9: Abstract Classes and Interfaces 82
Lecture 10: Graphical User Interfaces 88
Lecture 11: Event Handling 103
Lecture 12: More Examples of Handling Events 113
Lecture 13: Adapter Classes 118
Lecture 14: Java Database Connectivity 129
Lecture 15: More On JDBC 137
Lecture 16: Result Set 143
Lecture 17: Meta Data 156
Lecture 18: Java Graphics 165
Lecture 19: How to Animate? 174
Lecture 20: Applets 181
Lecture 21: Socket Programming 193
Lecture 22: 202
Lecture 23: Multithreading 209
Lecture 24: More on Multithreading 220
Lecture 25: Web Application Development 231

© Virtual University of Pakistan

Lecture 26: Java Servlets 245
Lecture 27: Creating a Simple Web Application in Tomcat 258
Lecture 28: Servlets Lifecycle 267
Lecture 29: More on Servlets 278
Lecture 30: Dispatching Requests 287
Lecture 31: Session Tracking 297
Lecture 32: Session Tracking 2 308
Lecture 33: Address Book Case Study Using Servlets 322
Lecture 34: Java Server Pages 334
Lecture 35: JavaServer Pages 341
Lecture 36 349
Lecture 37: JSP Action Elements and Scope 361
Lecture 38: JSP Custom Tags 375
Lecture 39: MVC + Case Study 389
Lecture 40: Model 2 Architecture MVC 404
Lecture 41: Layers and Tiers 427
Lecture 42: Expression Language 443
Lecture 43: JavaServer Pages Standard Tag Library (JSTL) 461
Lecture 44: Client Side Validation & JavaServer Faces (JSF) 472
Lecture 45: JavaServer Faces 481

© Virtual University of Pakistan

Lecture 1: Java Features

This handout is a traditional introduction to any language features. You might not be able to comprehend some of the features fully at this stage but don't worry, you'll get to know about these as we move on with the course.

1.1 Design Goals of Java

The massive growth of the Internet and the World-Wide Web leads us to a completely new way of looking at development of software that can run on different platforms like Windows, Linux and Solaris etc.

1.1.1 Right Language, Right Time

- Java came on the scene in 1995 to immediate popularity.
- Before that, C and C++ dominated the software development
  - compiled, no robust memory model, no garbage collector causes memory leakages, not great support of built in libraries
- Java brings together a great set of "programmer efficient" features
  - Putting more work on the CPU to make things easier for the programmer.

1.1.2 Java - Buzzwords (Vocabulary)

- From the original Sun Java whitepaper: "Java is a simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multi-threaded, and dynamic language."
- Here are some original java buzzwords.

1.1.3 Java -- Language + Libraries

- Java has two parts.
  - The core language -- variables, arrays, objects
    - The Java Virtual Machine (JVM) runs the core language
    - The core language is simple enough to run on small devices phones, smart cards, PDAs.
  - The libraries
    - Java includes a large collection of standard library classes to provide "off the shelf" code. (Useful built-in classes that comes with the language to perform basic tasks)
    - Example of these classes is String, ArrayList, HashMap, StringTokenizer (to break string into substrings), Date ...
    - Java programmers are more productive in part because they have access to a large set of standard, well documented library classes.

1.1.4 Simple

- Very similar C/C++ syntax, operators, etc.
- The core language is simpler than C++ -- no operator overloading, no pointers, no multiple inheritance
- The way a java program deals with memory is much simpler than C or C++.

1.1.5 Object-Oriented

- Java is fundamentally based on the OOP notions of classes and objects.
- Java uses a formal OOP type system that must be obeyed at compile-time and run-time.
- This is helpful for larger projects, where the structure helps keep the various parts consistent. Contrast to Perl, which has a more anything-goes feel.

1.1.6 Distributed / Network Oriented

- Java is network friendly -- both in its portable, threaded nature, and because Common networking operations are built-in to the Java libraries.

1.1.7 Robust / Secure / Safe

- Java is very robust
  - Both, vs. unintentional errors and vs. malicious code such as viruses.
  - Java has slightly worse performance since it does all this checking. (Or put the other way, C can be faster since it doesn't check anything.)
- The JVM "verifier" checks the code when it is loaded to verify that it has the correct Structure -- that it does not use an uninitialized pointer, or mix int and pointer types. This is one-time "static" analysis -- checking that the code has the correct structure without running it.
- The JVM also does "dynamic" checking at runtime for certain operations, such as pointer and array access, to make sure they are touching only the memory they should. You will write code that runs into
- As a result, many common bugs and security problems (e.g. "buffer overflow") are not possible in java. The checks also make it easier to find many common bugs easy, since they are caught by the runtime checker.
- You will generally never write code that fails the verifier, since your compiler is smart enough to only generate correct code. You will write code that runs into the runtime checks all the time as you debug -- array out of bounds, null pointer.
- Java also has a runtime Security Manager can check which operations a particular piece of code is allowed to do. As a result, java can run untrusted code in a "sandbox" where, for example, it can draw to the screen but cannot access the local file system.

1.1.8 Portable

- "Write Once Run Anywhere", and for the most part this works.
- Not even a recompile is required -- a Java executable can work, without change, on any Java enabled platform.

1.1.9 Support for Web and Enterprise Web Applications

- Java provides an extensive support for the development of web and enterprise

applications

- Servlets, JSP, Applets, JDBC, RMI, EJBs and JSF etc. are some of the Java technologies that can be used for the above mentioned purposes.

1.1.10 High-performance

- The first versions of java were pretty slow.
- Java performance has gotten a lot better with aggressive just-in-time-compiler (JIT) techniques.
- Java performance is now similar to C -- a little slower in some cases, faster in a few cases. However memory use and startup time are both worse than C.
- Java performance gets better each year as the JVM gets smarter. This works, because making the JVM smarter does not require any great change to the java language, source code, etc.

1.1.11 Multi-Threaded

- Java has a notion of concurrency wired right in to the language itself.
- This works out more cleanly than languages where concurrency is bolted on after the fact.

1.1.12 Dynamic

- Class and type information is kept around at runtime. This enables runtime loading and inspection of code in a very flexible way.

1.1.13 Java Compiler Structure

- The source code for each class is in a .java file. Compile each class to produce ".class" file.
- Sometimes, multiple .class files are packaged together into a .zip or .jar "archive" file.
- On UNIX or windows, the java compiler is called "javac". To compile all the .java files in a directory use "javac *.java".

1.1.14 Java: Programmer Efficiency

- Faster Development
  - Building an application in Java takes about 50% less time than in C or C++. So, Faster time to market
  - Java is said to be "Programmer Efficient".
- OOP
  - Java is thoroughly OOP language with robust memory system
  - Memory errors largely disappear because of the safe pointers and garbage collector. The lack of memory errors accounts for much of the increased programmer productivity.
- Libraries
  - Code re-uses at last -- String, ArrayList, Date, available and documented in a standard way

1.1.15 Microsoft vs. Java

- Microsoft hates Java, since a Java program (portable) is not tied to any particular operating system. If Java is popular, then programs written in Java might promote non-Microsoft operating systems. For basically the same reason, all the non-Microsoft vendors think Java is a great idea.
- Microsoft's C# is very similar to Java, but with some improvements, and some questionable features added in, and it is not portable in the way Java is. Generally it is considered that C# will be successful in the way that Visual Basic is: a nice tool to build Microsoft only software.
- Microsoft has used its power to try to derail Java somewhat, but Java remains very popular on its merits.

1.1.16 Java Is For Real

- Java has a lot of hype, but much of it is deserved. Java is very well matched for many modern problem
- Using more memory and CPU time but less programmer time is an increasingly appealing tradeoff.
- Robustness and portability can be very useful features
- A general belief is that Java is going to stay here for the next 10-20 years

1.1.17 References

- Majority of the material in this handout is taken from the first handout of course cs193j at Stanford.
- The Java™ Language Environment, White Paper, by James Gosling & Henry McGilton
    - Java's Sun site: http://java.sun.com
    - Java World : www.javaworld.com

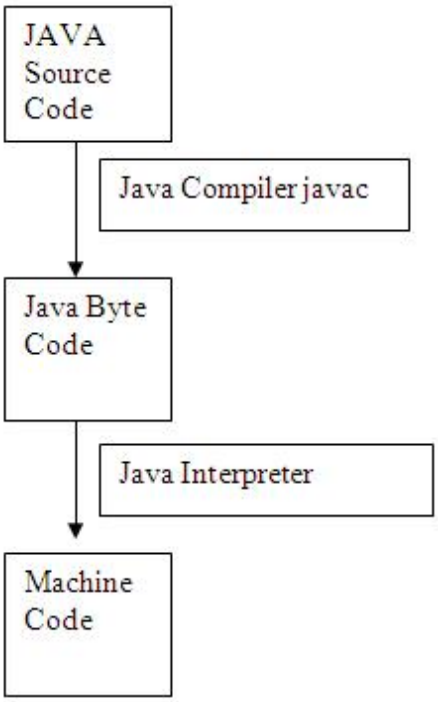
Lecture 2: Java Virtual Machine & Runtime Environment

2.1 Basic Concept

When you write a program in C++ it is known as source code. The C++ compiler converts this source code into the machine code of underlying system (e.g. Windows) If you want to run that code on Linux you need to recompile it with a Linux based compiler. Due to the difference in compilers, sometimes you need to modify your code.

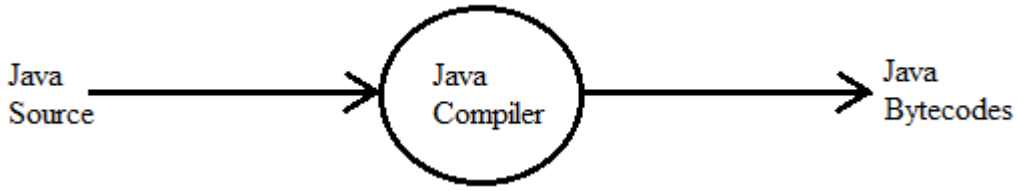
Java has introduced the concept of WORA (write once run anywhere). When you write a java program it is known as the source code of java. The java compiler does not compile this source code for any underlying hardware system, rather it compiles it for a software system known as JVM (This compiled code is known as byte code). We have different JVMs for different systems (such as JVM for Windows, JVM for Linux etc). When we run our program the JVM interprets (translates) the compiled program into the language understood by the underlying system. So we write our code once and the JVM runs it everywhere according to the underlying system.

This concept is discussed in detail below



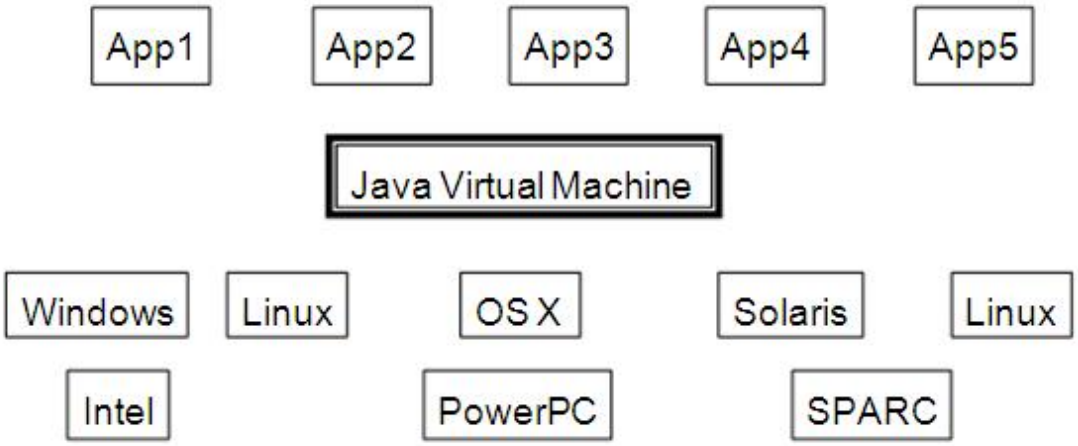
2.1.1 Byte code

- Java programs (Source code) are compiled into a form called Java bytecodes.
- The Java compiler reads Java language source (.java) files, translates the source into Java bytecodes, and places the bytecodes into class (.class) files.
- The compiler generates one class file for each class contained in java source file.



2.1.2 Java Virtual Machine (JVM)

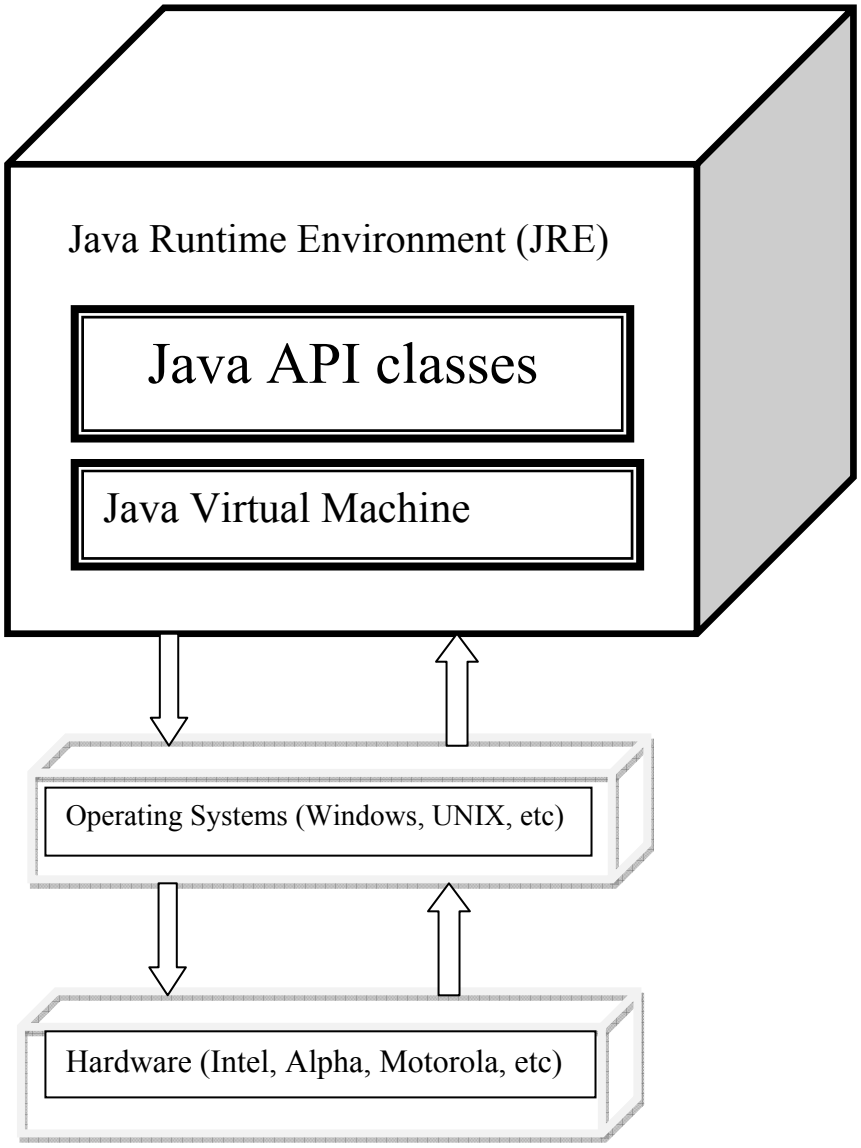
- The central part of java platform is java virtual machine.
- Java bytecode executes by special software known as a "virtual machine".
- Most programming languages compile source code directly into machine code, suitable for execution
- The difference with Java is that it uses bytecode - a special type of machine code.
- The JVM executes Java bytecodes, so Java bytecodes can be thought of as the machine language of the JVM.



- JVM are available for almost all operating systems.
- Java byte code is executed by using any operating system's JVM. Thus achieve portability.

2.1.3 Java Runtime Environment (JRE)

- The Java Virtual Machine is a part of a large system i.e. Java Runtime Environment (JRE).
- Each operating system and CPU architecture requires different JRE.
- The JRE consists of set of built-in classes, as well as a JVM.
- Without an available JRE for a given environment, it is impossible to run Java software.



2.1.4 References

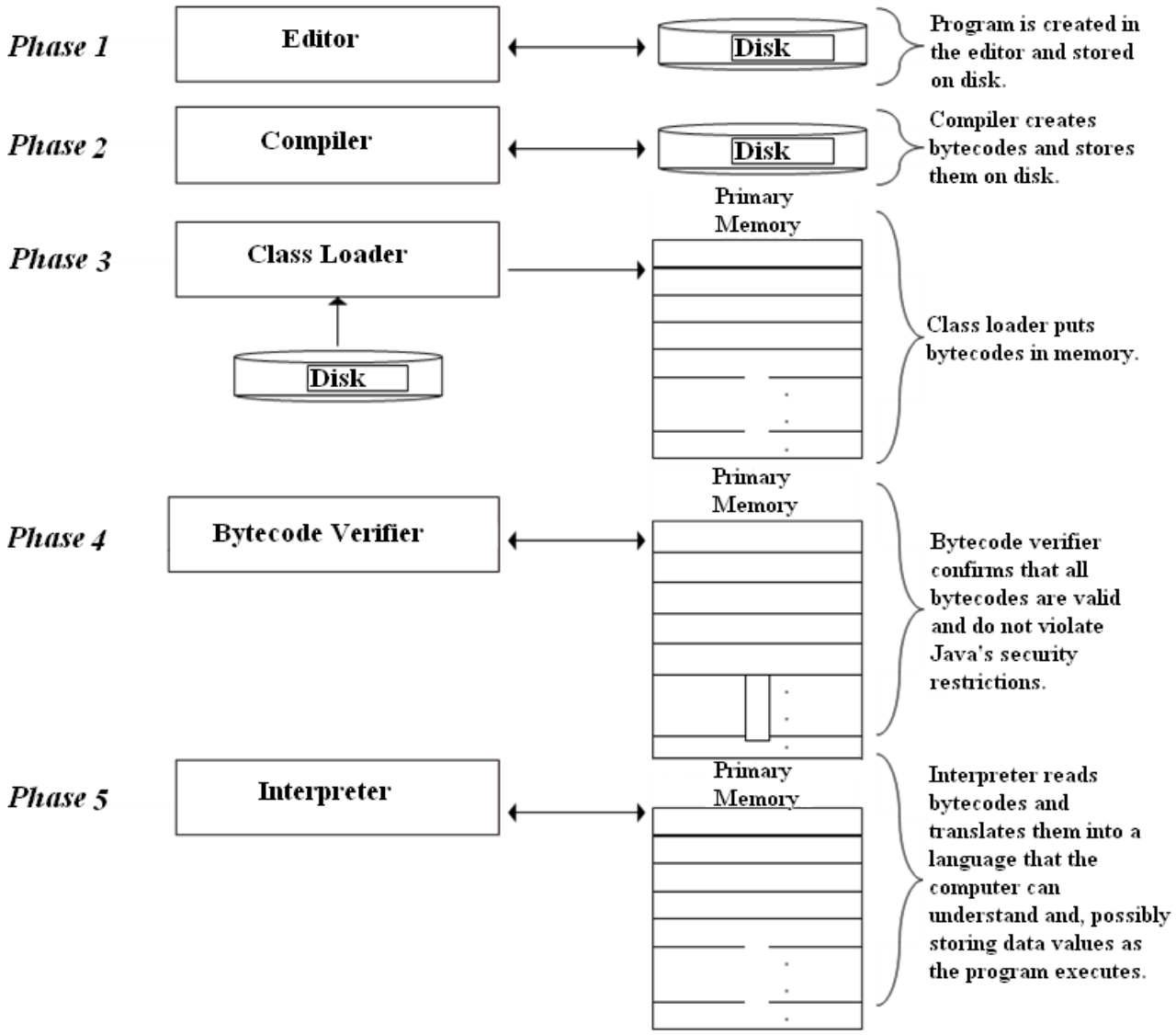
- Java World: http://www.javaworld.com
- Inside Java: http://www.javacoffeebreak.com/articles/inside_java

2.2 Java Program Development and Execution Steps

Java program normally go through five phases. These are

- Edit,
- Compile,
- Load,
- Verify and
- Execute

We look over all the above mentioned phases in a bit detail. First consider the following figure that summarizes the all phases of a java program.



2.2.1 Phase 1: Edit

Phase 1 consists of editing a file. This is accomplished with an editor program. The programmer types a java program using the editor like notepad, and make corrections if necessary.

When the programmer specifies that the file in the editor should be saved, the program is stored on a secondary storage device such as a disk. Java program file name ends with a .java extension.

On Windows platform, notepad is a simple and commonly used editor for the beginners. However java integrated development environments (IDEs) such as NetBeans, Borland JBuilder, JCreator and IBM's Eclipse Java built-in editors that are smoothly integrated into the programming environment.

2.2.2 Phase 2: Compile

In Phase 2, the programmer gives the command javac to compile the program. The java compiler translates the java program into byte codes, which is the language understood by the java interpreter.

To compile a program called Welcome.java type **javac Welcome.java** at the command window of your system. If the program compiles correctly, a file called Welcome. Class is produced. This is the file containing the byte codes that will be interpreted during the execution phase.

2.2.3 Phase 3: Loading

In phase 3, the program must first be placed in memory before it can be executed. This is done by the *class loader*, which takes the .class file (or files) containing the byte codes and transfers it to memory. The .class file can be loaded from a disk on your system or over a network (such as your local university or company network or even the internet).

Applications (Programs) are loaded into memory and executed using the *java interpreter* via the command java. When executing a Java application called Welcome, the command

Java Welcome

Invokes the interpreter for the Welcome application and causes the class loader to load information used in the Welcome program.

2.2.4 Phase 4: Verify

Before the byte codes in an application are executed by the java interpreter, they are verified by the *byte code verifier* in Phase 4. This ensures that the byte codes for class that are loaded form the internet (referred to as *downloaded classes*) are valid and that they do not violate Java's security restrictions. Java enforces strong security because java programs arriving over the network should not be able to cause damage to your files and your system (as computer viruses might).

2.2.5 Phase 5: Execute

Finally in phase 5, the computer, under the control of its CPU, interprets the program one byte code at a time. Thus performing the actions specified by the program.Programs may not work on the first try. Each of the preceding phases can fail because of various errors. This would cause the java program to print an error message. The programmer would return to the edit phase, make the necessary corrections and proceed through the remaining phases again to determine if the corrections work properly.

2.2.6 References:

- Java™ How to Program 5th edition by Deitel & Deitel
- Sun Java online tutorial: http://java.sun.com/docs/books/tutorial/java/index.html

2.3 Installation and Environment Setting

2.3.1 Installation

- Download the latest version j2se5.0 (java 2 standard edition) from http://www.oracle.com/technetwork/java/javase/downloads/index.htmls or get it from any other source like CD.

- Note: j2se also called jdk (java development kit). You can also use the previous versions like jdk 1.4 or 1.3 etc. but it is recommended that you use either jdk1.4 or jdk5.0
- Install j2se5.0 on your system

Note: For the rest of this handout, assume that j2se is installed in C:\Program Files\Java\jdk1.5.0

2.3.2 Environment Setting

Once you successfully installed the j2se, the next step is environment or path setting. You can accomplish this in either of two ways.

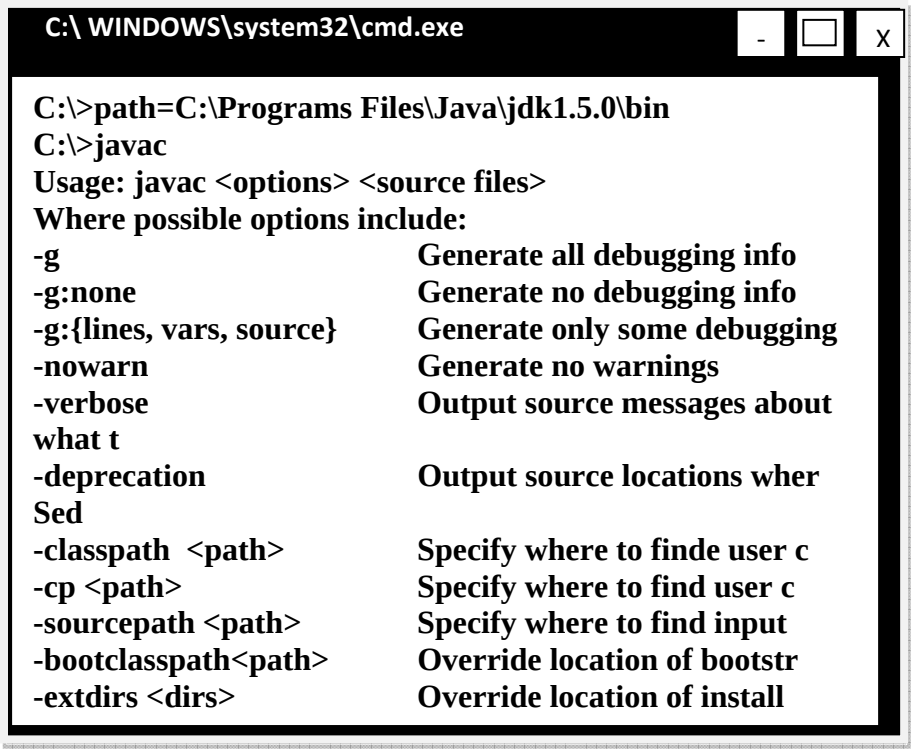
2.3.2.1 Temporary Path Setting

- Open the command prompt from Start -> Programs -> Accessories -> Command Prompt. The command prompt screen would be opened in front of you.
- Write the command on the command prompt according to the following format

  path = < java installation directory\bin >
- So, according to handout, the command will look like this

  path = C:\Program Files\Java\jdk1.5.0\bin
- To Test whether path has been set or not, write javac and press ENTER. If the list of options displayed as shown in the below figure means that you have successfully completed the steps of path setting.

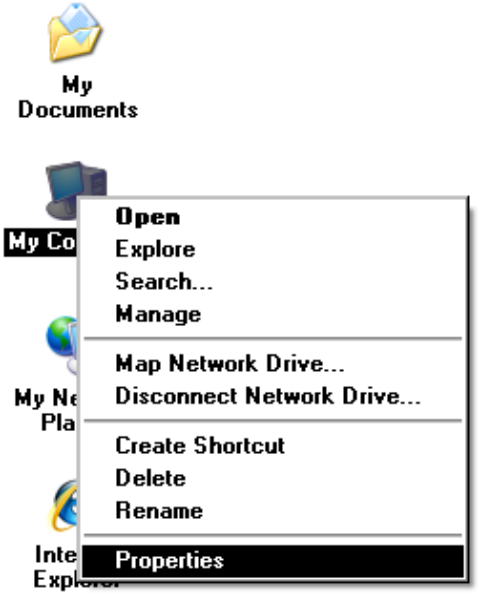
The above procedure is illustrates in the given below picture.



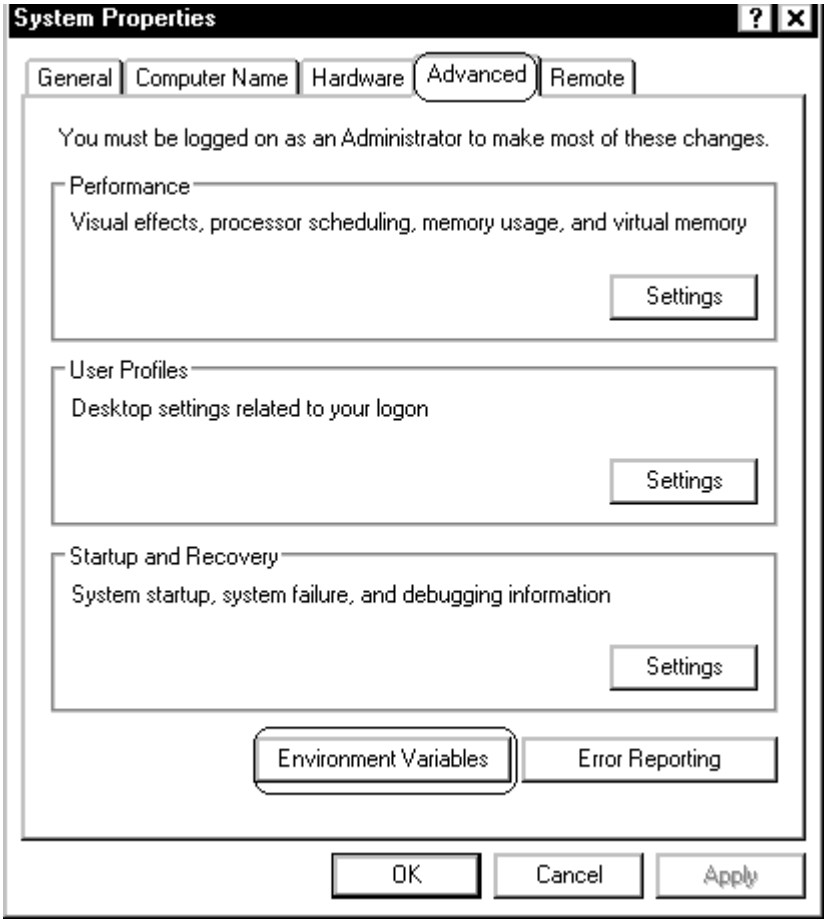
Note: The issue with the temporary path setting is you have to repeat the above explained procedure again and again each time you open a new command prompt window. To avoid this overhead, it is better to set your path permanently

2.3.2.2 Permanent Path Setting

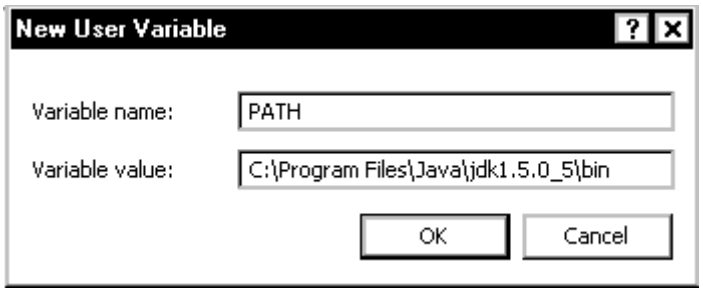
- In Windows NT (XP, 2000), you can set the permanent environment variable.
- Right click on my computer icon click on properties as shown below



A System Properties frame would appear as shown in the picture.



- Select the advanced tab followed by clicking the Environment Variable button. The Environment variables frame would be displayed in front of you
- Locate the Path variable in the System or user variables, if it is present there, select it by single click. Press Edit button. The following dialog box would be appeared.



- Write; **C:\Program Files\Java\jdk1.5.0\bin** at the end of the value field. Press OK button. Remember to write semicolon (;) before writing the path for java installation directory as illustrated in the above figure.
- If Path variable does not exist, click the New button. Write variable name "PATH", variable value **C:\Program Files\Java\jdk1.5.0\bin** and press OK button.

- Now open the command prompt and write **javac**, press enter button. You see the list of options would be displayed.
- After setting the path permanently, you have no need to set the path for each new opened command prompt.

2.3.3 References

Entire material for this handout is taken from the book JAVA A Lab Course by Umair Javed. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose without the consent of author.

2.4 First Program in Java

Like any other programming language, the java programming language is used to create applications. So, we start from building a classical "Hello World" application, which is generally used as the first program for learning any new language.

2.4.1 HelloWorldApp

- Open notepad editor from Start -> ProgarmFiles -> Accessories-> Notepad.
- Write the following code into it.

Note: Don't copy paste the given below code. Probably it gives errors and you can't able to remove them at the beginning stage.

```
1./*The HelloWorldApp class implements an application that
2.simply displays "Hello World!" to the standard output.
3.*/
4.  public class HelloWorldApp {
5.public static void main(String[] args) {
6.//Display the string. No global main
7.System.out.println("Hello World");
8. }
9. }
```

- To save your program, move to File menu and choose save as option.
- Save your program as "HelloWorldApp.java" in some directory. Make sure to add double quotes around class name while saving your program. For this example create a folder known as "examples" in D: drive

Note: Name of file must match the name of the public class in the file (at line 4). Moreover, it is case sensitive. For example, if your class name is MyClasS, than file name must be MyClasS. Otherwise the Java compiler will refuse to compile the program.

For the rest of this handout, we assume that program is saved in D:\examples directory.

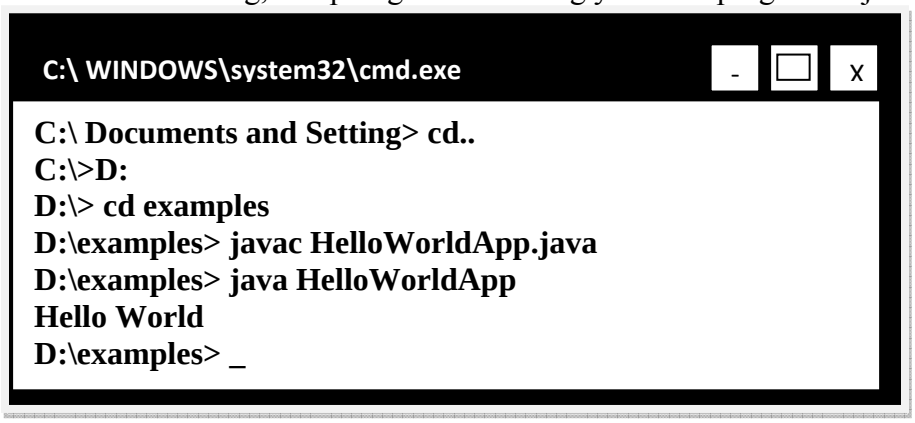
2.4.2 HelloWorldApp Described

- **Lines 1-3**
  - Like in C++, You can add multiple line comments that are ignored by the compiler.
- **Lines 4**
  - Line 4 declares the class name as HelloWorldApp. In java, every line of code must reside inside class. This is also the name of our program (HelloWorldApp.java).The compiler creates the HelloWorldApp.class if this program successfully gets compiled.
- **Lines 5**
  - Line 5 is where the program execution starts. The java interpreter must find this defined exactly as given or it will refuse to run the program. (However you can change the name of parameter that is passed to main. i.e. you can write String[] argv or String[] someParam instead of String[] args)
  - Other programming languages, notably C++ also use the main()declaration as the starting point for execution. However the main function in C++ is global and resides outside of all classes where as in Java the main function must reside inside a class. In java there are no global variables or functions. The various parts of this main function declaration will be covered at the end of this handout.
- **Lines 6**
  - Again like C++, you can also add single line comment
- **Lines 7**
  - Line 7 illustrates the method call. The println() method is used to print something on the console. In this example println()method takes a string argument and writes it to the standard output i.e. console.

- **Lines 8-9**
  - Line 8-9 of the program, the two braces, close the method main() and the classHelloWorldApprespectively.

2.4.3 Compiling and Running HelloWorldApp

- Open the command prompt from Start -> Program Files -> Accessories. OR alternatively you can write cmd in the run command window.
- Write cd.. to came out from any folder, and cd [folder name] to move inside the specified directory. To move from one drive to another, use [Drive Letter]: See figure given below
- After reaching to the folder or directory that contains your source code, in our case HelloWorldApp.java.
- Use "javac" on the command line to compile the source file (".java" file).
  - D:\examples> javac HelloWorld.java
- If program gets successfully compiled, it will create a new file in the same directory named HelloWorldApp.class that contains the byte-code.
- Use "java" on the command line to run the compiled .class file. Note ".class" would be added with the file name.
  - D:\examples> java HelloWorld
- You can see the Hello World would be printed on the console. Hurrah! You are successful in writing, compiling and executing your first program in java



2.4.4 Points to Remember

- Recompile the class after making any changes
- Save your program before compilation
- Only run that class using java command that contains the main method, because program executions always starts form main

2.5 An Idiom Explained

- You will see the following line of code often:
  - public static void main(String args[]) { …}
- About main()
  - "main" is the function from which your program starts
  - Why public?

- Since main method is called by the JVM that is why it is kept public so that it is accessible from outside. Remember private methods are only accessible from within the class

- Why static?
  - Every Java program starts when the JRE (Java Run Time Environment) calls the main method of that program. If main is not static then the JRE have to create an object of the class in which main method is present and call the main method on that object (In OOP based languages method are called using the name of object if they are not static). It is made static so that the JRE can call it without creating an object.
  - Also to ensure that there is only one copy of the main method per class
- Why void?
  - Indicates that main ( ) does not return anything.
- What is String args[] ?
  - Way of specifying input (often called command-line arguments) at startup of application. More on it latter

2.6 References

- Entire material for this handout is taken from the book JAVA A Lab Course by Umair Javed. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose without the consent of author.

Note: Coding exercises in working condition for this lecture are also available on "Downloads" section of LMS.

Lecture 3: Learning Basics

3.1 Strings

A *string* is commonly considered to be a sequence of characters stored in memory and accessible as a unit. Strings in java are represented as objects.

3.1.1 String Concatenation

- "+" operator is used to concatenate strings
    - System.out.println("Hello" + "World") will print Hello World on console
- String concatenated with any other data type such as int will also convert that datatype to String and the result will be a concatenated String displayed on console. For example,
    - int i = 4;
    - int j = 5;
        - System.out.println ("Hello" + i);// will print Hello 4 on screen
    - However
        - System.out.println(i+j);//will print 9 on the console because both i and j are of type int.

3.1.2 Comparing Strings

For comparing Strings never use == operator, use *equals* method of String class.

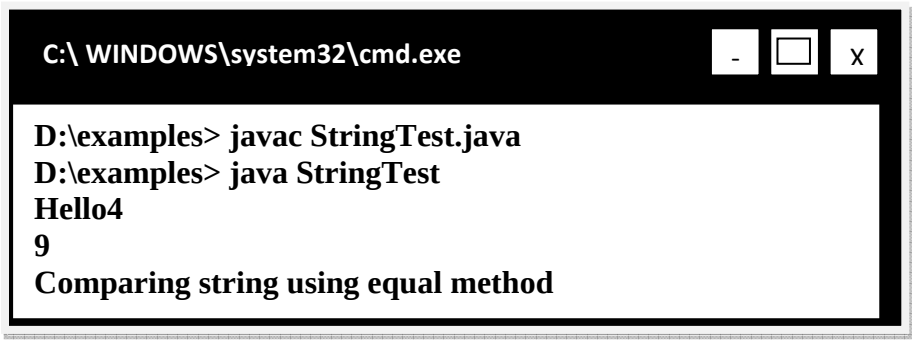
- == operator compares addresses (shallow comparison) while equals compares values (deep comparison)
    - E.g. string1.equals(string2)

Example Code: String concatenation and comparison

```
public class StringTest {
public static void main(String[] args) {
int i = 4;
int j = 5;
System.out.println("Hello" + i); // will print Hello4
System.out.println(i + j); // will print 9
String s1 = new String (“pakistan”);


String s2 = “pakistan”;
if (s1 == s2) {
System.out.println(“comparing string using == operator”);
}
if (s1.equals( s2) ) {
System.out.println(“comparing string using equal method”);
}
}
}
```

On execution of the above program, following output will produce



3.2 Taking in Command Line Arguments

In Java, the program can be written to accept *command-line-arguments*.

Example Code: command-line arguments

```
/* This Java application illustrates the use of Java command-line arguments. */
```

```
public static void main(String[] args){ //main method
System.out.println(”First argument ” + args[0]);
System.out.println(”Second argument ” +args[1]);
}//end main
}//End class.
```

To execute this program, we pass two arguments as shown below:

```
C:\java CmdLineArgsApp Hello World
First argument Hello
Second argument World
```

- These parameters should be separated by space.
- The parameters that we pass from the command line are stored as Strings inside the “args” array. You can see that the type of “args” array is String.

Example Code: Passing any number of arguments

In java, array knows their size by using the length property. By using, length property we can determine how many arguments were passed. The following code example can accept any number of arguments.

```
/* This Java application illustrates the use of Java command-line arguments. */
public class AnyArgsApp {
public static void main(String[] args){ //main method
for(int i=0; i < args.length; i++)
System.out.println(“Argument:” + i + “ value ” +args[i]);
}//end main
}//End class.
```

Output

```
C:\java AnyArgsApp
i can pass any number of arguments
Argument:0 value i
Argument:1 value can
Argument:2 value pass
Argument:3 value any
Argument:4 value number
Argument:5 value of
Argument:6 value arguments
```

3.3 Primitives vs. Objects

Everything in Java is an “Object”, as every class by default inherits from class “Object”,

except a few primitive data types, which are there for efficiency reasons.

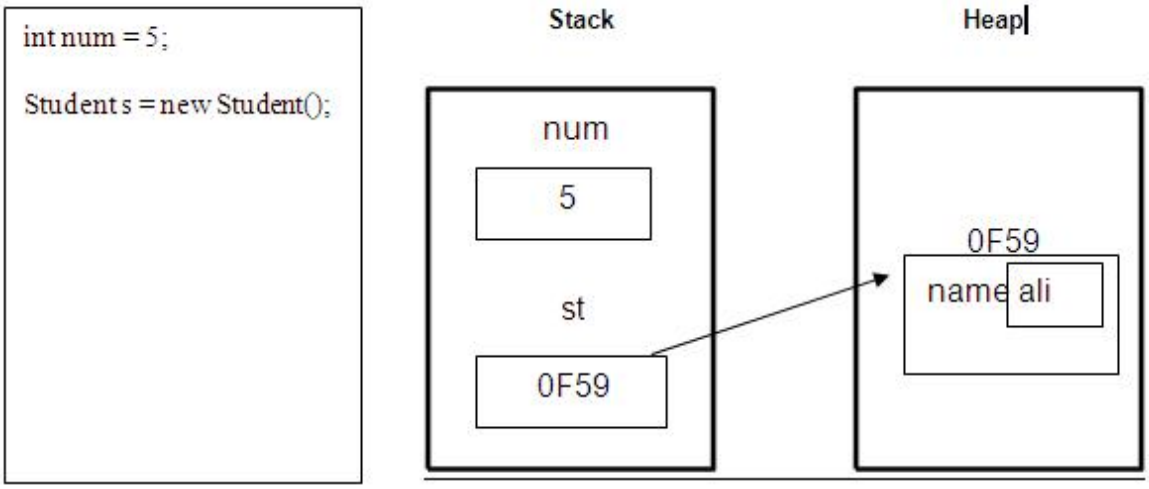
- Primitive Data Types
    - Primitive Data types of java

boolean, byte	1 byte
char, short	2 bytes
int, float	4 bytes
long, double	8 bytes

- Primitive data types are generally used for local variables, parameters and instance variables (properties of an object)
- Primitive data types are located on the stack and we can only access their value, while objects are located on heap and we have a reference to these objects
- Also primitive data types are always passed by value while objects are always passed by reference in java. There is no C++ like methods
    - void someMethod(int &a, int & b) // not available in java

3.4 Stack vs. Heap

Stack and heap are two important memory areas. Primitives are created on the stack while objects are created on heap. This will be further clarified by looking at the following diagram that is taken from Java Lab Course.



3.5 Wrapper Classes

Each primitive data type has a corresponding object (wrapper class). These wrapper classes provides additional functionality (conversion, size checking etc.), which a primitive data type cannot provide.

Primitive Data Type	Corresponding Object Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

3.5.1 Wrapper Use

You can create an object of Wrapper class using a String or a primitive data type

- Integer num = new Integer(4); or
- Integer num = new Integer("4");

Note: num is an object over here not a primitive data type

You can get a primitive data type from a Wrapper using the corresponding value function

- int primNum = num.intValue();

3.5.2 Converting Strings to Numeric Primitive Data Types

To convert a string containing digits to a primitive data type, wrapper classes can help. *parseXxx* method can be used to convert a String to the corresponding primitive data type.

- String value = "532";
  int d = Integer.parseInt(value);
- String value = "3.14e6";
  double d = Double.parseDouble(value);

The following table summarizes the parser methods available to a java programmer.

Data Type	Convert String using either…
byte	Byte.parseByte(string) new Byte(string).byteValue()
short	Short.parseShort(string) new Short(string).shortValue()
int	Integer.parseInteger(string) new Integer(string).intValue()
long	Long.parseLong(String) new Long(string).longValue()
float	Float.parseFloat(string) new Float(string).floatValue()
double	Double.parseDouble(string) new Double(string).doubleValue()

Example Code: Taking Input / Output

So far, we learned how to print something on console. Now the time has come to learn how to print on the GUI. Taking input from console is not as straightforward as in C++. Initially we'll study how to take input through GUI (by using JOPtionPane class).

The following program will take input (a number) through GUI and prints its square on the console as well on GUI.

```
1. import javax.swing.*;
2. public class InputOutputTest {
3.     public static void main(String[] args) {
4. //takes input through GUI
5. String input = JOptionPane.showInputDialog("Enter number");
6. int number = Integer.parseInt(input);
7. int square = number * number;
8. //Display  square on console
9. System.out.println("square:" + square);
10. //Display square on GUI
11. JOptionPane.showMessageDialog(null, "square:"+ square);
12. System.exit(0);
13. }
14. }
```

On line 1, swing package was imported because it contains the *JOptionPane* class that will be used for taking input from GUI and displaying output to GUI. It is similar to header classes of C++.

On line 5, showInputDialog method is called of JOptionPane class by passing string argument that will be displayed on GUI (dialog box). This method always returns back a

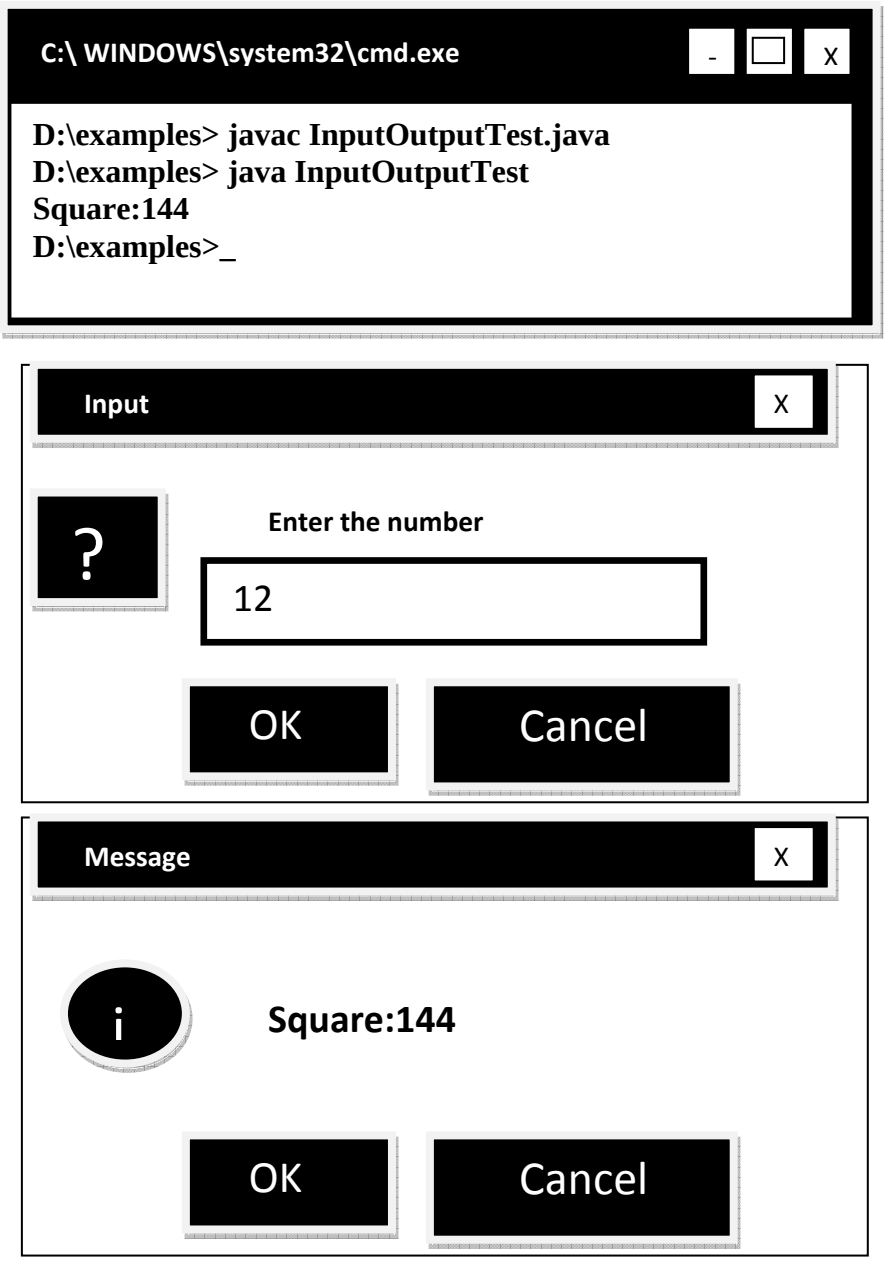
String regardless of whatever you entered (int, float, double, char) in the input filed.

Our task is to print square of a number on console, so we first convert a string into a number by calling parseInt method of Integer wrapper class. This is what we done on line number 6.

Line 11 will display square on GUI (dialog box) by using showMessageDialog method of JOptionPane class. The first argument passed to this method is null and the second argument must be a String. Here we use string concatenation.

Line 12 is needed to return the control back to command prompt whenever we use JoptionPane class.

Compile & Execute



3.6 Selection & Control Structure

The if-else and switch selection structures are exactly similar to we have in C++. All relational operators that we use in C++ to perform comparisons are also available in java with same behavior. Likewise for, while and do-while control structures are alike to C++.

3.7 Reference:

- Java tutorial: http://www.dickbaldwin.com/java
- Example code, their explanations and corresponding figures for this handout are taken from

the book JAVA A Lab Course by Umair Javed. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose without the consent of author.

Note: Coding exercises in working condition for this lecture are also available on"**Downloads**" section of LMS.

Lecture 4: Object Oriented Programming

Java is fundamentally object oriented. Every line of code you write in java must be inside a class (not counting import directives).OOP fundamental stones Encapsulation, Inheritance and Polymorphism etc. are all fully supported by java.

4.1 OOP Vocabulary Review

4.1.1 Classes

- Definition or a blueprint of a user defined data type
- Prototypes for objects
- Think of it as a map of the building on a paper

4.1.2 Objects

- Nouns, things in the world
- Anything we can put a thumb on
- Objects are instantiated or created from class

4.1.3 Constructor

- A special method that is implicitly invoked. Used to create an Object (that is, an Instance of the Class) and to initialize it.

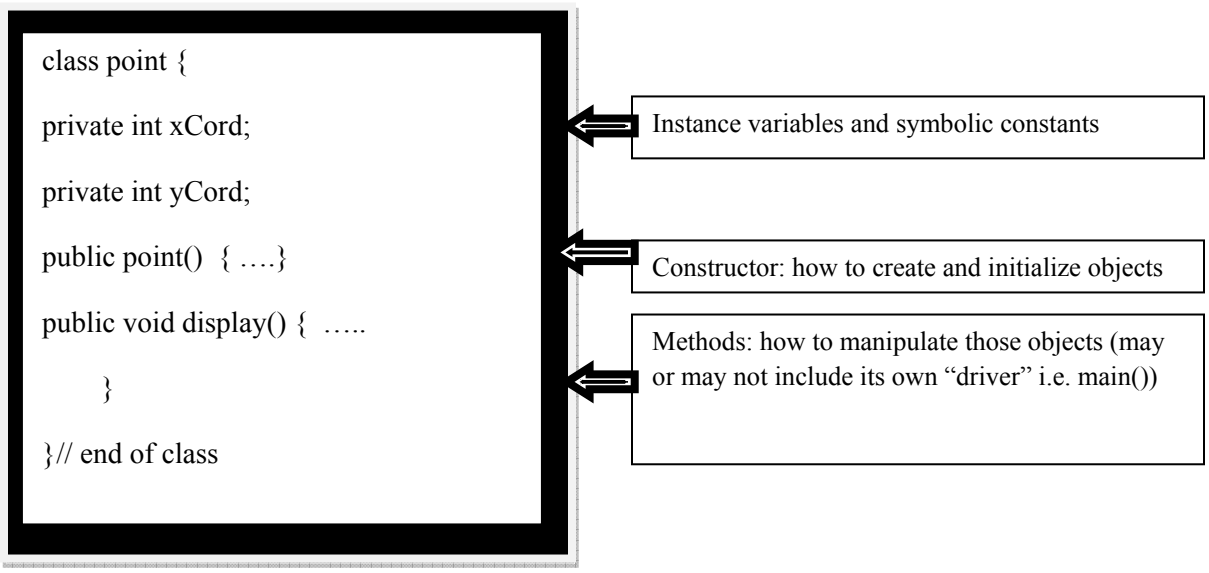
4.1.4 Attributes

- Properties an object has

4.1.5 Methods

- Actions that an object can do

4.2 Defining a Class



4.3 Comparison with C++

Some important points to consider when defining a class in java as you probably noticed from the above given skeleton are

- There are no global variables or functions. Everything resides inside a class. Remember we wrote our main method inside a class.(For example, in HelloWorldApp program)
- Specify access modifiers (public, private or protected) for each member method or data members at every line.
  - *public:* accessible anywhere by anyone
  - *private:* Only accessible within this class
  - *protect:* accessible only to the class itself and to its subclasses or other classes in the same package.
  - *default:* default access if no access modifier is provided. Accessible to all classes in the same package.
- There is no semicolon (;) at the end of class.
- All methods (functions) are written inline. There are no separate header and implementation files.
- Automatic initialization of class level data members if you do not initialize them
  - Primitives
    - Numeric (int, float etc) with zero.
    - Char with null
    - Boolean with false
  - Object References
    - With null

Note: Remember, the same rule is not applied to local variables (defined inside

method body). Using a local variable without initialization is a compile time error

```
public void someMethod( ) {
int x; //local variable
System.out.println(x); // compile time error
}
```

- Constructor
    - Same name as class name
    - Does not have a return type
    - No initialization list
    - JVM provides a *zero argument* (default) constructor only if a class doesn't define its own constructor
- Destructors
    - Are not required in java class because memory management is the responsibility of JVM.

4.4 Task - Defining a Student class

The following example will illustrate how to write a class. We want to write a "Student" class that

- Should be able to store the following characteristics of student
    - Roll No
    - Name
- Provide default, parameterized and copy constructors
    - Provide standard getters/setters (discuss shortly) for instance variables
        - Make sure, roll no has never assigned a negative value i.e. ensuring the correct state of the object
        - Provide print method capable of printing student object on console

4.5 Getters / Setters

The attributes of a class are generally taken as private or protected. So to access them outside of a class, a convention is followed knows as getters & setters. These are generally public methods. The words *set* and *get* are used prior to the name of an attribute. Another important purpose for writing getter & setters to control the values assigned to an attribute.

Student Class Code

```
// File Student.java
public class Student {
private String name;
```

```
private int rollNo;
// Standard Setters
public void setName (String name) {
this.name = name;
}
// Note the masking of class level variable rollNo
public void setRollNo (int rollNo) {
if (rollNo > 0) {
this.rollNo = rollNo; }else {
this.rollNo = 100;
}
}
// Standard Getters
public String getName ( ) {
return name;
}
public int getRollNo ( ) {
return rollNo;
}
// Default Constructor
public Student() {
 name = “not set”;
rollNo = 100;
}
// parameterized Constructor for a new student
public Student(String name, int rollNo) {
setName(name);//call to setter of name
setRollNo(rollNo); //call to setter of rollNo
}
// Copy Constructor for a new student public Student(Student s) {
name = s.name;
rollNo = s.rollNo;
}
// method used to display method on console public void print () {
System.out.print("Student name: " +name);
System.out.println(", roll no: " +rollNo);    }
} // end of class
```

4.6 Using a Class

Objects of a class are always created on heap using the “new” operator followed by constructor

- Student s = new Student ( ); // no pointer operator “*” between Student and s
- Only String constant is an exception
    - String greet = “Hello” ; // No new operator

- However you can also use
- String greet2 = new String("Hello");

Members of a class (member variables and methods also known as instance variables/methods) are accessed using "." operator. There is no "–>" operator in java

- s.setName("Ali");
- s–>setName("Ali") is incorrect and will not compile in java

Note: Objects are always passed by reference and primitives are always passed by value in java.

4.6.1 Task - Using Student Class

- Create objects of student class by calling default parameterize and copy constructor
- Call student class various methods on these objects.

Student client code

```
// File Test.java
/* This class create Student class objects and demonstrates how to call various methods on objects
*/

public class Test{
public static void main (String args[]){
// Make two student obejcts

Student s1 = new Student("ali", 15);
Student s2 = new Student(); //call to default costructor
s1.print(); // display ali and 15
s2.print(); // display not set and 100
s2.setName("usman");
s2.setRollNo(20);

System.out.print("Student name:" + s2.getName());
System.out.println(" rollNo:" + s2.getRollNo());

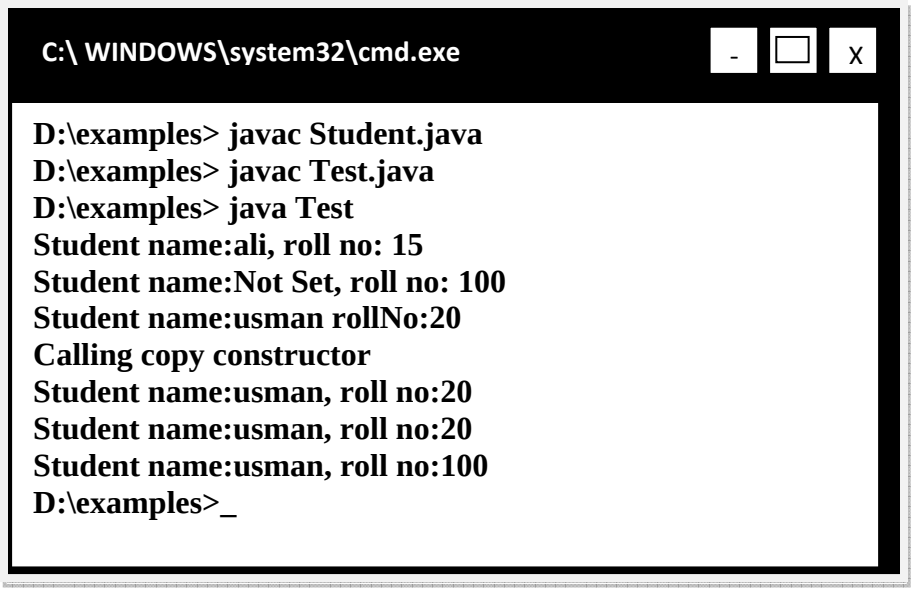
System.out.println("calling copy constructor");
Student s3 = new Student(s2); //call to copy constructor

s2.print();
s3.print();
s3.setRollNo(-10); //Roll No of s3 would be set to 100
s3.print();
/*NOTE: public vs. private
A statement like "b.rollNo = 10;" will not compile in a client of the Student class when rollNo is
```

```
declared
protected or private
*/
} //end of main
} //end of class
```

Compile & Execute

Compile both classes using **javac** command. Run Test class using **java** command.



4.7 More on Classes

4.7.1 Static

A class can have static variables and methods. Static variables and methods are associated with the class itself and are not tied to any particular object. Therefore statics can be accessed without instantiating an object. Static methods and variables are generally accessed by class name.

The most important aspect of statics is that they occur as a single copy in the class regardless of the number of objects. Statics are shared by all objects of a class. Non static methods and instance variables are not accessible inside a static method because no this reference is available inside a static method.

We have already used some static variables and methods. Examples are

- System.out.println("some text");---out is a static variable

- JOptionPane.**showMessageDialog**(null,"sometext");-----

showMessageDialog is a static method

4.7.2 Garbage Collection & Finalize

Java performs garbage collection and eliminates the need to free objects explicitly. When an object has no references to it anywhere except in other objects that are also unreferenced, its space can be reclaimed. Before an object is destroyed, it might be necessary for the object to perform some action. For example: to close an opened file. In such a case, define a *finalize()* method with the actions to be performed before the object is destroyed.

4.7.2.1 Finalize

When a finalize method is defined in a class, Java run time calls *finalize()* whenever it is about to recycle an object of that class. It is noteworthy that a garbage collector reclaims objects in any order or never reclaims them. We cannot predict and assure when garbage collector will get back the memory of unreferenced objects.

The garbage collector can be *requested* to run by calling *System.gc()* method. It is not necessary that it accepts the request and run.

Example Code: using static & finalize ()

We want to count exact number of objects in memory of a Student class the one defined earlier. For this purpose, we'll modify Student class.

- Add a static variable *countStudents* that helps in maintaining the count of student objects.
- Write a getter for this static variable. (Remember, the getter also must be static one. Hoping so, you know the grounds).
- In all constructors, write a code that will increment the countStudents by one.
- Override *finalize()* method and decrement the countStudents variable by one.
- Override *toString()* method.

Class *Object* is a superclass (base or parent) class of all the classes in java by default. This class has already *finalize()* and *toString()* method (used to convert an object state into string). Therefore we are actually overriding these methods over here. (We'll talk more about these in the handout on inheritance).

By making all above modifications, student class will look like

```
public class Student {
private String name;
private int rollNo;
private static int countStudents = 0;
// Standard Setters
public void setName (String name) {
this.name = name;
}

// Note the masking of class level variable rollNo
public void setRollNo (int rollNo) {
if (rollNo > 0) {
this.rollNo = rollNo; }else {
this.rollNo = 100;
}
}
// Standard Getters
public String getName ( ) {
return name;
}
public int getRollNo ( ) {
return rollNo;
}
// gettter of static countStudents variable
public static int getCountStudents(){
return countStudents;
}
// Default Constructor public Student() {
name = “not set”;
rollNo = 100;
countStudents += 1;
}
// method used to display method on console
public void print () {
System.out.print("Student name: " +name);
System.out.println(", roll no: " +rollNo);
}
// overriding toString method of java.lang.Object class
public String toString(){
return “name: ” + name + “ RollNo: ” + rollNo;
}
// overriding finalize method of Object class
public void finalize(){
countStudents -= 1;
}
} // end of class
```

Next, we'll write driver class. After creating two objects of student class, we deliberately loose object's reference and requests the JVM to run garbage collector to reclaim the memory. By printing countStudents value, we can confirm that. Coming up code is of the Test class.

```
// File Test.java public class Test{
public static void main (String args[]){
 int numObjs;
// printing current number of objects i.e 0
numObjs = Student.getCountStudents();
System.out.println("Students Objects" + numObjs);
// Creating first student object & printing its values
Student s1 = new Student("ali", 15);
System.out.println("Student: " + s1.toString());
// printing current number of objects i.e. 1
numObjs = Student.getCountStudents();
System.out.println("Students Objects" + numObjs);
// Creating second student object & printing its values
Student s2 = new Student("usman", 49);

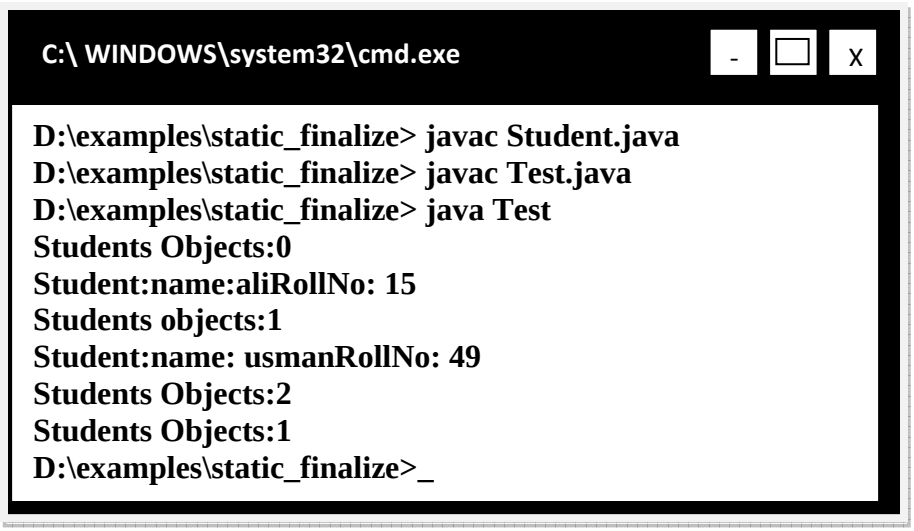
// implicit call to toString() method
System.out.println("Student: " + s2);

// printing current number of objects i.e. 2
numObjs = Student.getCountStudents();
System.out.println("Students Objects" + numObjs);

// loosing object reference s1 = null;
// requesting JVM to run Garbage collector but there is
// no guarantee that it will run
```

```
System.gc();
// printing current number of objects i.e. unpredictable
numObjs = Student.getCountStudents();
System.out.println("Students Objects" + numObjs);
} //end of main
} //end of class
```

The compilation and execution of the above program is given below. Note that output may be different one given here because it all depends whether garbage collector reclaims the memory or not. Luckily, in my case it does.



4.8 Reference:

- Sun java tutorial: http://java.sun.com/docs/books/tutorial/java
- Thinking in java by Bruce Eckle
- Beginning Java2 by Ivor Hortan
- Example code, their explanations and corresponding execution figures for this handout are taken from the book JAVA A Lab Course by Umair Javed. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose without the consent of author.

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 5: Inheritance

In general, inheritance is used to implement a "is-a" relationship. Inheritance saves code rewriting for a client thus promotes reusability.

In java parent or base class is referred as *super* class while child or derived class is known as *sub* class.

5.1 Comparison with C++

- Java only supports single inheritance. As a result a class can only inherit from one class at one time.
- Keyword *extends* is used instead of "**:**" for inheritance.
- All functions are virtual by default
- All java classes inherit from Object class (more on it later).
- To explicitly call the super class constructor, use *super* keyword. It's important to remember that call to super class constructor must be first line.
- Keyword super is also used to call overridden methods.

Example Code: using inheritance

We'll use three classes to get familiar you with inheritance. First one is Employee class. This will act as super class. Teacher class will inherit from Employee class and Test class is driver class that contains main method. Let's look at them one by one

```
class Employee{
protected int id;
protected String name;
//parameterized constructor
public Employee(int id, String name){
this.id = id;
this.name = name;
}
//default constructor
public Employee(){
// calling parameterized constructor of same (Employee)
// class by using keyword this
this (10, "not set");
}
//setters
public void setId (int id) {
```

```
this.id = id;
}
public void setName (String name) {
this.name = name;
}

//getters
public int getId () {
return id;
}
public String getName () {
return name;
}
// displaying employee object on console
public void display(){
System.out.println(“in employee display method”);
System.out.println("Employee id:" + id + " name:" + name);
}
//overriding object’s class toString method
public String toString() {
System.out.println(“in employee toString method”);
return "id:" + id + "name:" + name;
}
}//end class
```

The Teacher class extends from Employee class. Therefore Teacher class is a subclass of Employee. The teacher class has an additional attribute i.e. qualification.

```
class Teacher extends Employee{
private String qual;
//default constructor
public Teacher () {
//implicit call to superclass default construct qual = "";
}
//parameterized constructor
public Teacher(int i, String n, String q){
//call to superclass param const must be first line
super(i,n);
qual = q;
}
```

```
//setter
public void setQual (String qual){
this.qual = qual;
}
//getter
public String getQual(){
return qual;
}
//overriding display method of Employee class
public void display(){
System.out.println("in teacher's display method");

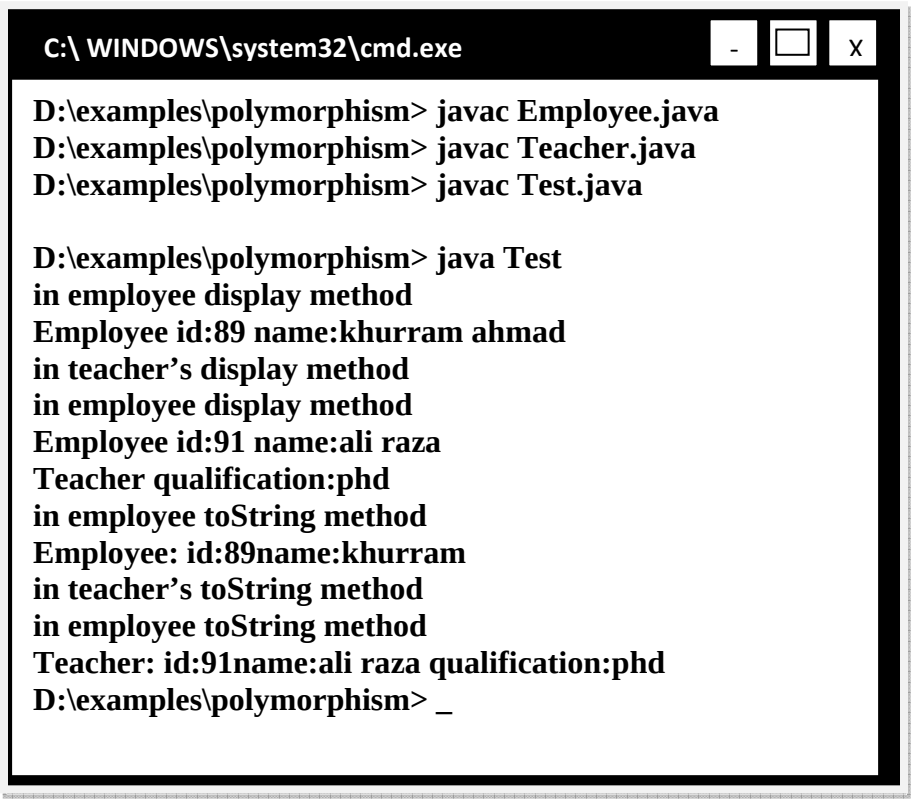
super.display(); //call to superclass display method
System.out.println("Teacher qualification:" + qual);
}
//overriding toString method of Employee class
public String toString() {
System.out.println("in teacher's toString method");
String emp = super.toString();
return emp +" qualification:" + qual;
}
}//end class
```

Objects of Employee & Teacher class are created inside main method in Test class. Later calls are made to display and toString method using these objects.

```
class Test{
public static void main (String args[]){
System.out.println("making object of employee");
Employee e = new Employee(89, "khurram ahmad");
System.out.println("making object of teacher");
Teacher t = new Teacher (91, "ali raza", "phd");
e.display(); //call to Employee class display method
t.display(); //call to Teacher class display method
```

```
// calling employee class toString method explicitly
System.out.println("Employee: " +e.toString());
// calling teacher class toString implicitly
System.out.println("Teacher: " + t);
} //end of main
}//end class
```

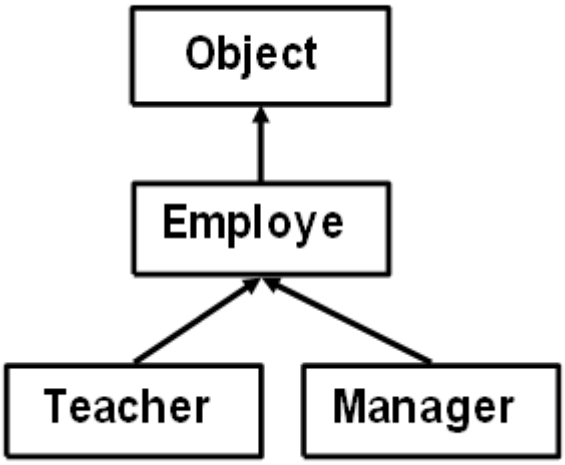
Output



5.2 Object - The Root Class

The Object class in Java is a superclass for all other classes defined in Java's class libraries, as well as for user-defined Java classes. For user defined classes, its not necessary to mention the Object class as a super class, java does it automatically for you.

The class Hierarchy of Employee class is shown below. Object is the super class of Employee class and Teacher is a subclass of Employee class. We can make another class Manager that can also extends from Employee class.



5.3 Polymorphism

"Polymorphic" literally means "of multiple shapes" and in the context of OOP, Polymorphic means "having multiple behaviors".

- A parent class reference can point to the subclass objects because of is-a relationship. For example a Employee reference can point to:
    - Employee Object
    - Teacher Object
    - Manager Object
- A polymorphic method results in different actions depending on the object being referenced
    - Also known as late binding or run-time binding

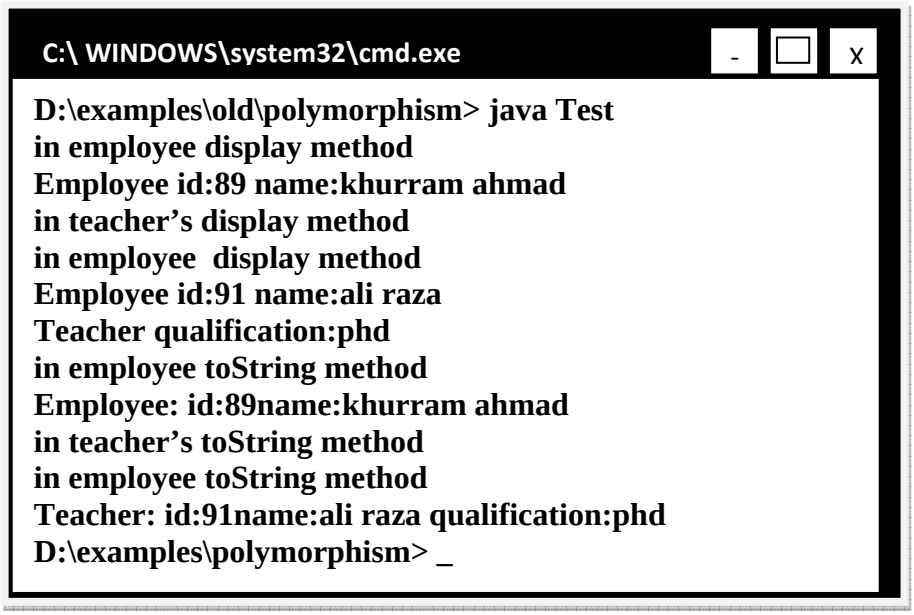
Example Code: using polymorphism

This Test class is the modification of last example code. Same Employee & Teacher classes are used. Objects of Employee & Teacher class are created inside main methods and calls are made to display and toString method using these objects.

```
class Test{
public static void main (String args[]){
// Make employee references
Employee ref1, ref2;
ref1 = new Employee(89, "khurram ahmad");
// assign employee object to first employee reference
ref2 = new Teacher (91, "ali raza", "phd");
```

```
// is-a relationship, polymorphism
//call to Employee class display method
ref1.display();
//call to Teacher class display method
ref2.display();
System.out.println("Employee: " +ref1.toString());// call to Employee class toString method
System.out.println("Teacher: " + ref2.toString());// call to Teacher class toString method
} //end of main
}//end class
```

Output



5.4 Type Casting

In computer science, **type conversion** or **typecasting** refers to changing an entity of one data type into another. Type casting can be categorized into two types

5.4.1 Up-casting

- converting a smaller data type into bigger one
- Implicit - we don't have to do something special
- No loss of information
- Examples of
  - Primitives

```
int a = 10;
double b = a;
```

  - Classes

```
Employee e = new Teacher( );
```

5.4.2 Down-casting

- converting a bigger data type into smaller one
- Explicit - need to mention
- Possible loss of information
- Examples of
  - o Primitives

```
double a = 7.65;
int    b = (int) a;
```

  - o Classes

```
Employee e = new Teacher( );// up-casting
Teacher t= (Teacher) e;// down-casting
```

5.5 References:

- Java tutorial: http://java.sun.com/docs/books/tutorial/java/javaOO/
- Stanford university
- Example code, their explanations and corresponding figures for handout 5-1,5-2 are taken from the book JAVA A Lab Course by Umair Javed. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose without the consent of author.

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 6: Collections

A collection represents group of objects know as its elements. Java has a built-in support for collections. Collection classes are similar to STL in C++. An advantage of a collection over an array is that you don't need to know the eventual size of the collection in order to add objects to it. The java.utilpackage provides a set of collection classes that helps a programmer in number of ways.

6.1 Collections Design

All classes almost provides same methods like get (), size (), isEmpty() etc. These methods will return the object stored in it, number of objects stored and whether collection contains an object or not respectively.

Java collections are capable of storing any kind of objects. Collections store references to objects. This is similar to using a void* in C. therefore down casting is required to get the actual type. For example, if string in stored in a collection then to get it back, we write

```
String element = (String)arraylist.get(i);
```

6.2 Collection messages

Some basic messages (methods) are:

- Constructor
  - creates a collection with no elements.
- int size()
  - returns the number of elements in a collection.
- boolean add(Object)
  - adds a new element in the collection.
  - returns true if the element is added successfully false otherwise.
- boolean isEmpty()
  - returns true if this collection contains no element false otherwise.
- boolean contains(Object)
  - returns true if this collection contains the specified element by using iterative search.
- boolean remove(Object)
  - removes a single instance of the specified element from this collection, if it is present .

6.3 Array List

It's like a resizable array. Array List actually comes as a replacement the old "Vector" collection. As we add or remove elements into or from it, it grows or shrinks over time.

6.3.1 Useful Methods

- add (Object)
  - With the help of this method, any object can be added into Array List because Object is the superclass of all classes.
  - Objects going to add will implicitly up cast.
- Object get(int index)
  - Returns the element at the specified position in the list
  - Index ranges from 0 to size()-1
  - Must cast to appropriate type
- remove (int index)
  - Removes the element at the specified position in this list.
  - Shifts any subsequent elements to the left (subtracts one from their indices).
- int size( )

Example Code: Using Array List class

We'll store Student objects in the Array List. We are using the same student class which we built in previous lectures/handouts.

We'll add three student objects and later prints all the student objects after retrieving them from Array List. Let's look at the code:

```
import java.util.*;
public class ArrayListTest {
public static void main(String[] args) {

// creating arrayList object by calling constructor
ArrayList<Student> al= new ArrayList<Student>();

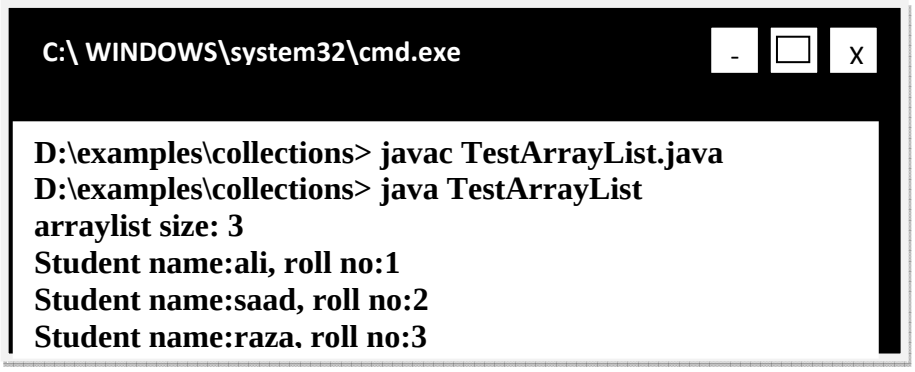
// creating three Student objects
Student s1 = new Student ("ali" , 1);
Student s2 = new Student ("saad" , 2);
Student s3 = new Student ("raza" , 3);
// adding elements (Student objects) into arralylist
```

```
 al.add(s1);
al.add(s2);
al.add(s3);

// checking whether arraylist is empty or not Boolean
b = al.isEmpty ();
if (b == true) {
System.out.println("arraylist is empty");
}
else {
int size = al.size();
System.out.println("arraylist size: " + size);
}

// using loop to iterate. Loops starts from 0 to one
// less than size
for (int i=0; i<al.size(); i++ ){
// retrieving object from arraylist
Student s = (Student) al.get(i);
// calling student class print method
s.print();
} // end for loop
} // end main
} // end class
```

Output



6.4 HashMap

Store elements in the form of *key- value* pair form. A key is associated with each object that is stored. This allows fast retrieval of that object. Keys are unique.

6.4.1 Useful Methods

- put(Object key, Object Value)
    - Keys & Values are stored in the form of objects (implicit up casting is performed).
    - Associates the specified value with the specified key in this map.
    - If the map previously contained a mapping for this key, the old value is replaced.
- Object get(Object key)
    - Returns the value to which the specified key is mapped in this identity hash map, or nullif the map contains no mapping for this key.
    - Must downcast to appropriate type when used
- int size( )

Example Code: using HashMap class

In this example code, we'll store Student objects as values and their rollnos in the form of strings as keys. Same Student class is used. The code is:

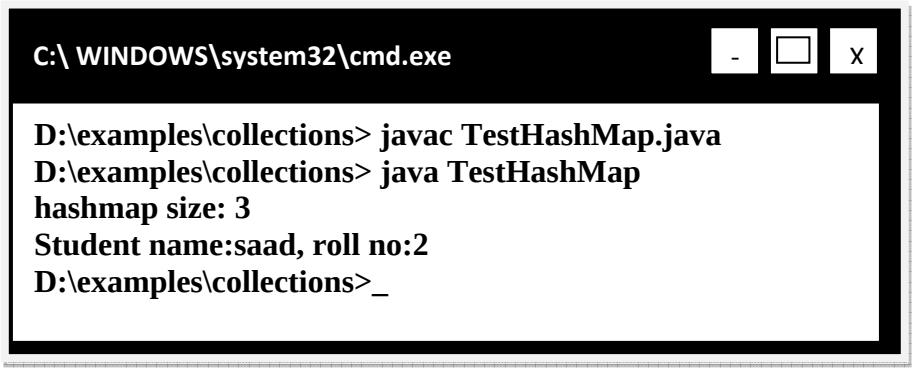
```
import java.util.*;
public class HashMapTest {
public static void main(String[] args) {
// creating HashMap object
HashMap <String, Student> h =new HashMap<String, Student>();
// creating Student objects
Student s1 = new Student ("ali" , 1);
Student s2 = new Student ("saad" , 2);
Student s3 = new Student ("raza" , 6);

// adding elements (Student objects) where roll nos
// are stored as keys and student objects as values
h.put("one",s1);
h.put("two",S2);
h.put("three", S3);
// checking whether hashmap is empty or not
boolean b = h.isEmpty ();
if (b == true) {
System.out.println("hashmap is empty");
}
else {
int size = h.size();
System.out.println("hashmap size: " + size);
}
```

```
// retrieving student object against rollno two and
// performing downcasting
Student s = (Student) h.get("two");
// calling student's class print method s.print();

} // end main
} // end class
```

Output



6.5 References:

- J2SE 5.0 new features: http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html
- Technical Article: http://java.sun.com/developer/technicalArticles/releases/j2se15/
- Beginning Java2 by Ivor Horton
- Example code, their explainations and corresponding figures for this handout are taken from the book JAVA A Lab Course by Umair Javed. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose without the consent of author.

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

6.6 Address Book

Warning: It is strongly advised that you have to type the code given in this example by yourself. Do not **copy/paste** it; most probably you will get unexpected errors that you have never seen. Some bugs are deliberately introduced as well to avoid copy-pasting. TAs will not cooperate with you in debugging such errors.

6.6.1 Problem

- We want to build an address book that is capable of storing name, address & phone number of a person.
- Address book provides functionality in the form of a JOptionPane based menu. The feature list includes
    - Add - to add a new person record
    - Delete - to delete an existing person record by name
    - Search - to search a person record by name
    - Exit - to exit from application
- The Address book should also support persistence for person records

6.6.2 Approach for Solving Problem

Building a small address book generally involves 3 steps. Let us briefly discuss each step and write a solution code for each step

6.6.2.1 Step1 - Make PersonInfo class

- First of all you need to store your desired information for each person. For this you can create a user-defined data type (i.e. a class). Make a class PersonInfowith name, address and phone number as its attributes.
- Write a parameterized constructor for this class.
- Write print method in Person class that displays one person record on a message dialog box.

The code for **PersonInfo** class is given below.

```
import javax.swing.*;
class PersonInfo {

String name;
String address;
String phoneNum;
//parameterized constructor
public PersonInfo(String n, String a, String p) {
name = n; address = a; phoneNum = p;
}

//method for displaying person record on GUI
public void print( ) {
```

```
phoneNum);
}
}
```

Note: Not declaring attributes as private is a bad approach but we have done it to keep things simple here.

6.6.2.2 Step2 - Make Address Book class

- Take the example of daily life; generally address book is used to store more than one person records and we don't know in advance how many records are going to be added into it.
- So, we need some data structure that can help us in storing more than one PersonInfoobjects without concerning about its size.
- Array Listcan be used to achieve the above functionality
- Create a class Address Book with an ArrayList as its attribute. This arraylist will be used to store the information of different persons in the form of PersonInfo Objects. This class will also provide *addPerson, deletePerson & searchPerson* methods. These methods are used for adding new person records, deleting an existing person record by name and searching among existing person records by name respectively.
- Input/Output will be performed through JOptionPane.

The code for **AddressBook**class is

```
import javax.swing.*;
import java.util.*;

class AddressBook {
ArrayList<PersonInfo> persons;
//constructor
public AddressBook ( ) {
persons = new ArrayList()<PersonInfo>();
}

//add new person record to arraylist after taking input
public void addPerson( ) {
String name = JOptionPane.showInputDialog("Enter name");
String add    = JOptionPane.showInputDialog("Enter address"); String pNum =
JOptionPane.showInputDialog("Enter phone no");
//construct new person object
PersonInfo p = new PersonInfo(name, add, pNum);
//add the above PersonInfo object to arraylist
persons.add(p);
}
//search person record by name by iterating over arraylist
public void searchPerson (String n) {
```

```
for (int i=0; i< persons.size(); i++) {
PersonInfo p = (PersonInfo)persons.get(i);
if ( n.equals(p.name) ) {
p.print();
}
} // end for
} // end searchPerson
//delete person record by name by iterating over arraylist
public void deletePerson (String n) {
for (int i=0; i< persons.size(); i++) {
PersonInfo p = (PersonInfo)persons.get(i);
 if ( n.equals(p.name))
{
persons.remove(i);
}
}
}
} // end class
```

The *addperson* method first takes input for name, address and phone number and than construct a PersonInfo object by using the recently taken input values. Then the newly constructed object is added to the arraylist - *persons*.

The *searchPerson* & *deletePerson* methods are using the same methodology i.e. first they search the required record by name and then prints his/her detail or delete the record permanently from the ArrayList.

Both the methods are taking string argument, by using this they can perform their search or delete operation. We used forloop for iterating the whole ArrayList.By using the size method of ArrayList, we can control our loop as ArrayList indexes range starts from 0 to one less than size.

Notice that, inside loop we retrieve each PersonInfo object by using down casting operation. After that we compare each PersonInfo object's name by the one passed to these methods using equal method since Strings are always being compared using equalmethod.

Inside ifblock of *searchPerson*, *print* method is called using PersonInfo object that will display person information on GUI. On the other hand, inside ifblock of *deletePerson* method, remove method of ArrayList class is called that is used to delete record from *persons* i.e. ArrayList.

6.6.2.3 Step3 - Make Test class (driver program)

- This class will contain a mainmethod and an object of AddressBook class.
- Build GUI based menu by using switch selection structure
- Call appropriate methods of AddressBookclass

The code for **Test**class is

```
import javax.swing.*;
class Test {
public static void main (String args[]) {
AddressBook ab = new AddressBook();
String input, s;
int ch;
while (true) {
input = JOptionPane.showInputDialog("Enter 1 to add " + "\n Enter 2 to Search \n Enter 3 to
Delete" +
"\n Enter 4 to Exit");
ch = Integer.parseInt(input);
switch (ch) {
case 1:
ab.addPerson();
break;
case 2:
s = JOptionPane.showInputDialog( "Enter name to search ");
ab.searchPerson(s);
break;
case 3:
s = JOptionPane.showInputDialog( "Enter name to delete ");
ab.deletePerson(s);
break;
case 4: System.exit(0);
}
}//end while
}//end main
}
```

Note that we use *infinite* while loop that would never end or stop given that our program should only exit when user enters 4 i.e. exit option.

Compile & Execute

Compile all three classes and run Test class. Bravo, you successfully completed the all basic three steps. Enjoy!

6.7 Reference

Entire content for this handout are taken from the book JAVA A Lab Course by Umair Javed. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose.

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 7: Intro to Exceptions

7.1 Types of Errors

Generally, you can come across three types of errors while developing software. These are Syntax, Logic & Runtime errors.

7.1.1 Syntax Errors

- Arise because the rules of the language are not followed.

7.1.2 Logic Errors

- Indicates that logic used for coding doesn't produce expected output.

7.1.3 Runtime Errors

- Occur because the program tries to perform an operation that is impossible to complete.
- Cause exceptions and may be handled at runtime (while you are running the program)
- For example divide by zero

7.2 What is an Exception?

- An exception is an event that usually signals an erroneous situation at run time
- Exceptions are wrapped up as objects
- A program can deal with an exception in one of three ways:
    - ignore it
    - handle it where it occurs
    - handle it an another place in the program

7.3 Why handle Exceptions?

- Helps to separate error handling code from main logic (the normal code you write) of the program.
- As different sort/type of exceptions can arise, by handling exceptions we can

distinguish between them and write appropriate handling code for each type for example we can differently handle exceptions that occur due to division by Zero and exceptions that occur due to non-availability of a file.

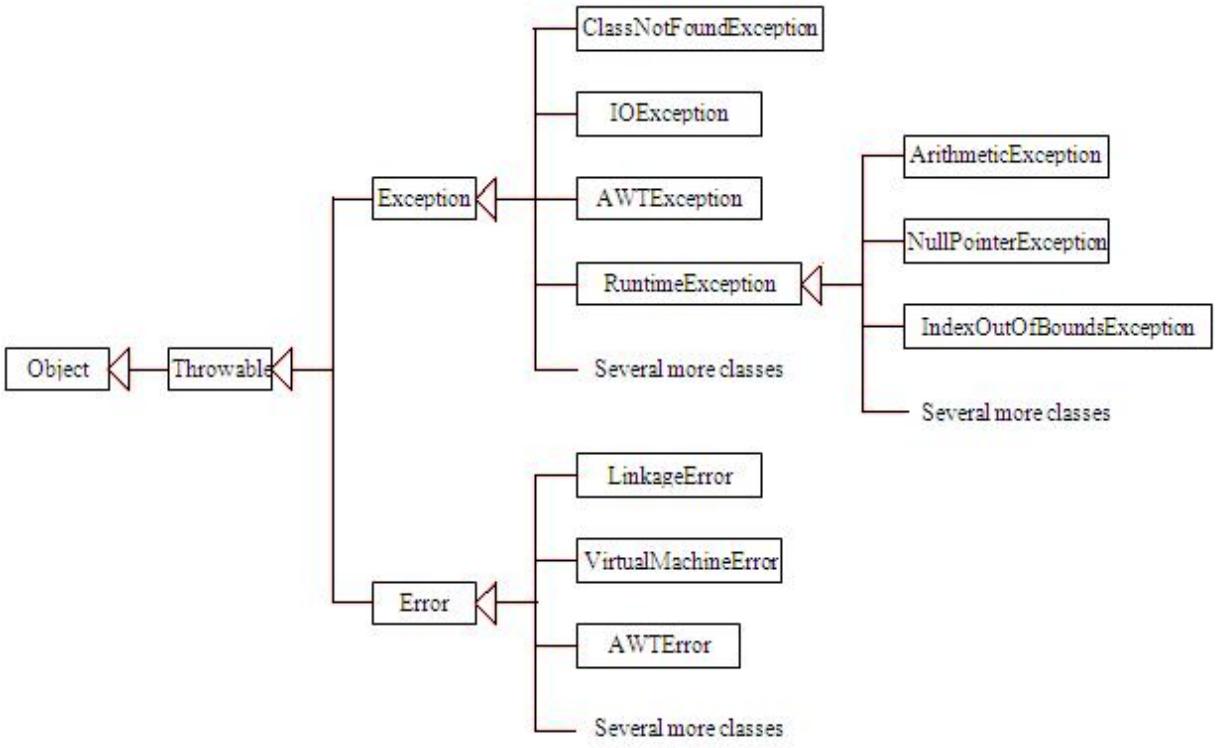
- If not handled properly, program might terminate.

7.4 Exceptions in Java

- An exception in java is represented as an object that's created when an abnormal situation arises in the program. Note that an error is also represented as an object in Java, but usually represents an unrecoverable situation and should not be caught
- The exception object stores information about the nature of the problem. For example, due to network problem or class not found etc.
- All exceptions in java are inherited from a class know as *Throwable*.

7.5 Exception Hierarchy

Following diagram is an abridged version of Exception class hierarchy



7.6 Types of Exceptions

Exceptions can be broadly categorized into two types, Unc*hecked* & *Checked Exceptions*.

7.6.1 Unchecked Exceptions

- Subclasses of Runtime Exception and Error.
- Does not require explicit handling
- Run-time errors are internal to your program, so you can get rid of them by debugging your code
- For example, null pointer exception; index out of bounds exception; division by zero exception.

7.6.2 Checked Exceptions

- Must be caught or declared in a throws clause
- Compile will issue an error if not handled appropriately
- Subclasses of Exception other than subclasses of Runtime Exception.
- Other arrives from external factors, and cannot be solved by debugging
- Communication from an external resource - e.g. a file server or database

7.7 How Java handles Exceptions

Java handles exceptions via 5 keywords. **Try**, **catch**, **finally**, **throw** & **throws.**

7.7.1 try block

- Write code inside this block which could generate errors

7.7.2 Catch block

- Code inside this block is used for exception handling
- When the exception is raised from try block, only than catch block would execute.

7.7.3 finally block

- This block always executes whether exception occurs or not.
- Write clean up code here, like resources (connection with file or database) that are opened may need to be closed.

The basic structure of using try - catch - finally block is shown in the picture below:

```
try                                                    //try block
{
        // write code that could generate exceptions
} catch (<exception to be caught>)                     //catch block
{
        //write code for exception handling
}
 ……

catch (<exception to be caught>)                       //catch block
{
        //code for exception handling
} finally                                              //finally block
{
        //any clean-up code, release the acquired resources
}
```

7.7.4 throw

- To manually throw an exception, keyword throwis used. **Note:** we are not covering throw clause in this handout

7.7.5 throws

- If method is not interested in handling the exception than it can throw back the exception to the caller method using throwskeyword.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.

7.8 References:

- Java tutorial by Sun: http://java.sun.com/docs/books/turorial
- Beginning Java2 by Ivor Hortan
- Thinking in Java by Bruce Eckle
- CS193j Stanford University

7.9 Code Examples of Exception Handling

7.9.1 Unchecked Exceptions

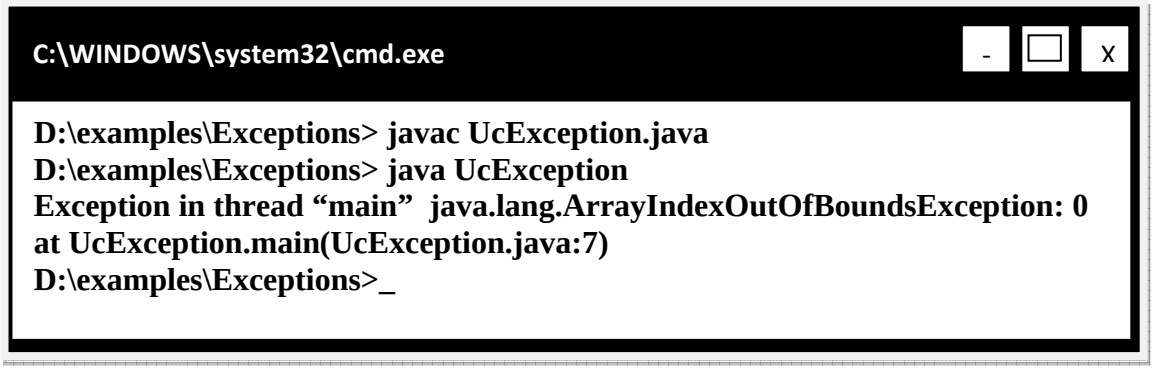
Example Code: UcException.java

The following program takes one command line argument and prints it on the console

```
// File UcException.java public class UcException {
public static void main (String args[ ]) {
System.out.println(args[0]);
}
}
```

Compile & Execute

If we compile & execute the above program without passing any command line argument, an ArrayIndexOutOfBoundsException would be thrown. This is shown in the following picture



7.9.2 Why?

Since we have passed no argument, therefore the size of String args[]is zero, and we have tried to access the first element (first element has index zero) of this array.

From the output window, you can find out, which code line causes the exception to be raised. In the above example, it is

System.out.println (args[0]);

7.9.3 Modify UcException.java

Though it is not mandatory to handle unchecked exceptions we can still handle Unchecked Exceptions if we want to. These modifications are shown in bold.

```
// File UcException.java
public class UcException {
public static void main (String args[ ]) {
```

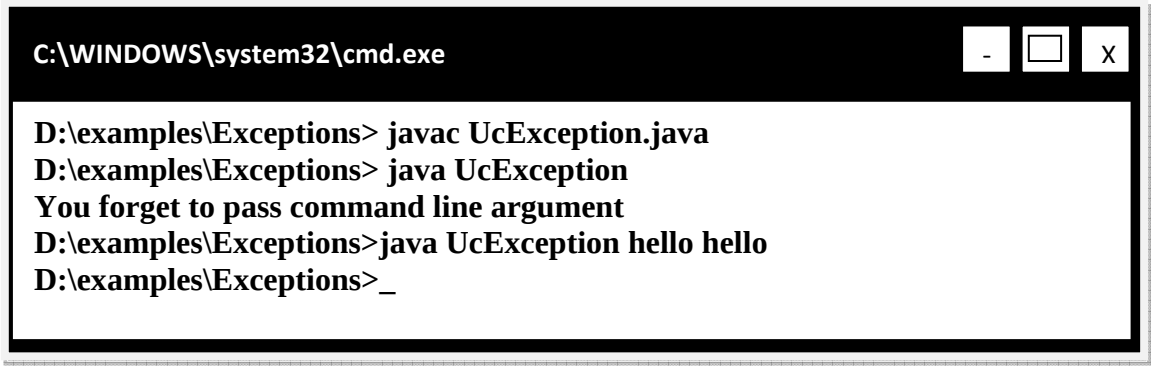
```
try {
System.out.println(args[0]);
catch (IndexOutOfBoundsException ex) {
System.out.println("You forget to pass command line argument");
}
}
```

The possible exception that can be thrown is IndexOutOfBoundsException, so we handle it in the catch block.

When an exception occurs, such as IndexOutOfBoundsException in this case, then an object of type IndexOutOfBoundesException is created and it is passed to the corresponding catch block (i.e. the catch block which is capable of handling this exception). The catch block receives the exception object inside a variable which is *ex* in this case. It can be any name; it is similar to the parameter declared in the method signature. It receives the object of exception type (IndexOutOfBoundsExceptoin) it is declared.

Compile & Execute

If we execute the modified program by passing command line argument, the program would display on console the provided argument. After that if we execute this program again without passing command line argument, this time information message would be displayed which is written inside catch block.



7.10 Checked Exceptions

Example Code: CException.java

The following program reads a line *(hello world)* from a file and prints it on the console. The File reading code is probably new for you. We'll explain it in the coming handouts (Streams). For now, assumed that the code written inside the main read one line from a file and prints that to console.

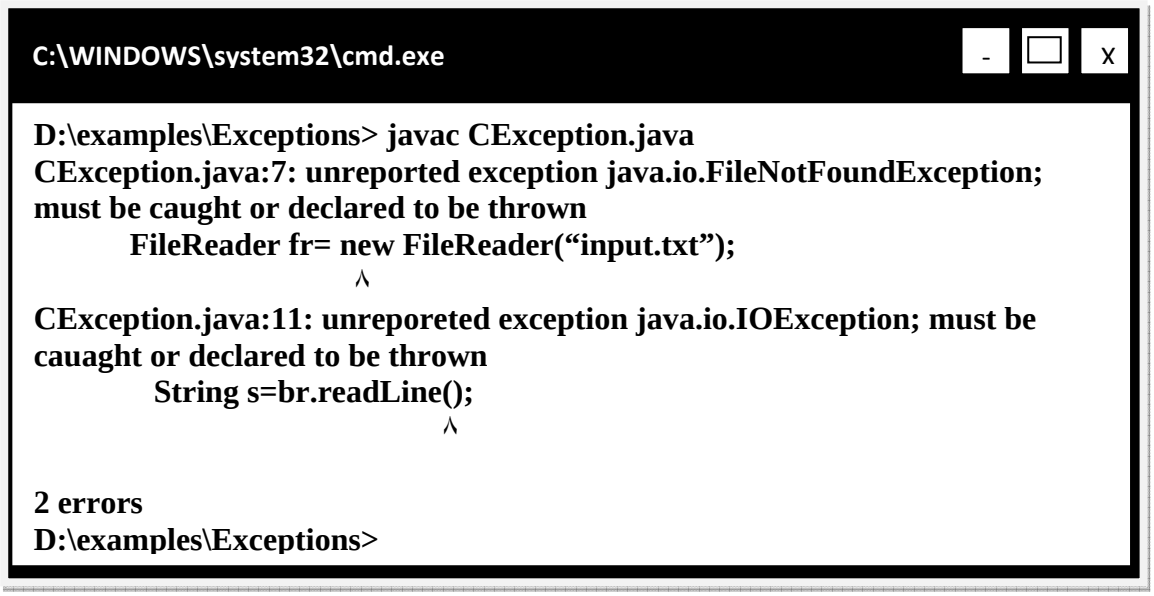
```
// File CException.java
```

```
 import java.io.* ;
public class CException {
public static void main (String args[ ]) { FileReader fr = new FileReader (“input.txt”);
BufferedReader br = new BufferedReader (fr);

//read the line form file
String line = br.readLine(); System.out.println(line);        }
}
```

Compile & Execute

If you try to compile this program, the program will not compile successfully and displays the message of unreported exception. This happens when there is code that can generate a checked exception but you have not handled that exception. Remember checked exceptions are detected by compiler. As we early discussed, without handling Checked exception, out program won’t compile.



Modify CException.java

As we have discussed earlier, it is mandatory to handle checked exceptions. In order to compile the code above, we modify the above program so that file reading code is placed inside a try block. The expected exception (IOException) that can be raised is caught in catch block.

```
// File CException.java import java.io.* ;
```

```
public class CException {
public static void main (String args[ ]) {
try{
FileReader fr = new FileReader (“input.txt”);
BufferedReader br = new BufferedReader (fr);

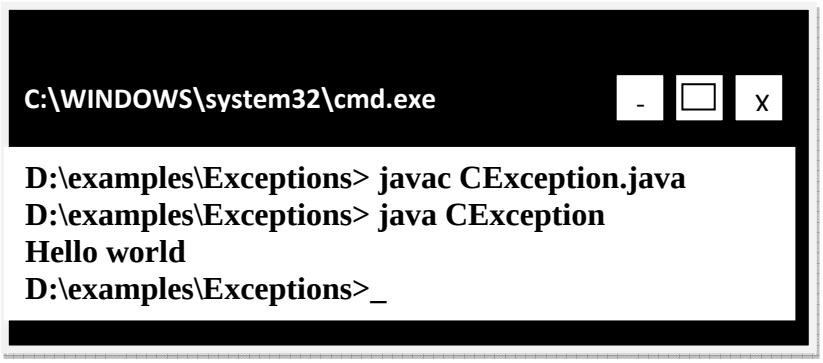
//read the line form file
String line = br.readLine();
System.out.println(line);
}
catch( IOException ex) {
System.out.println(ex);
}
}
}
```

The code line written inside the catch block will print the exception name on the console if exception occurs

Compile & Execute

After making changes to your program, it would compile successfully. On executing this program, *hello world* would be displayed on the console

Note: Before executing, make sure that a text file named *input.txt* must be placed in the same directory where the program is saved. Also write *hello world* in that file before saving it.



7.11 The finally block

The finally block always executes regardless of exception is raised or not while as you remembered the catch block only executes when an exception is raised.

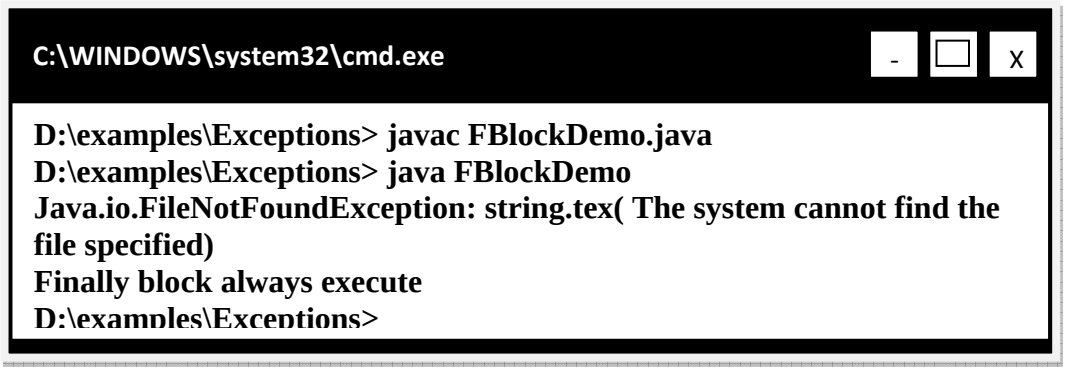
Example Code: FBlockDemo.java

```
// File FBlockDemo.java
```

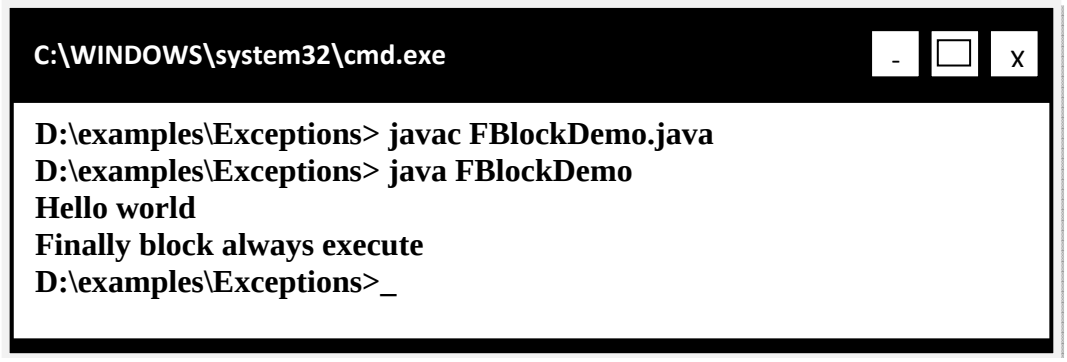
```
import java.io.* ;
public class FBlockDemo {
public static void main (String args[ ])
{
try{
FileReader fr = new FileReader ("strings.txt");
BufferedReader br = new BufferedReader (fr);
//read the line form file
String line = br.readLine();
System.out.println(line);
}catch( IOException ex) {
System.out.println(ex);
}
finally {
System.out.println("finally block always execute");
}
}
}
```

Compile & Execute

The program above, will read one line from *string.txt* file. If *string.tx* is not present in the same directory the FileNotFoundException would be raised and catch block would execute as well as the finally block.



If *string.txt* is present there, no such exception would be raised but still finally block executes. This is shown in the following output diagram



7.12 Multiple catch blocks

- Possible to have multiple catch clauses for a single try statement
  - Essentially checking for different types of exceptions that may happen
- Evaluated in the order of the code
  - Bear in mind the Exception hierarchy when writing multiple catch clauses!
  - If you catch Exception first and then IOException, the IOException will never be caught!

Example code: MCatchDemo.java

The following program would read a number form a file *numbers.txt* and than prints its square on the console

```
// File MCatchDemo.java
 import java.io.* ;
public class MCatchDemo {
public static void main (String args[ ]) {
try{
// can throw FileNotFound or IOException
FileReader fr = new FileReader ("numbers.txt");
BufferedReader br = new BufferedReader (fr);

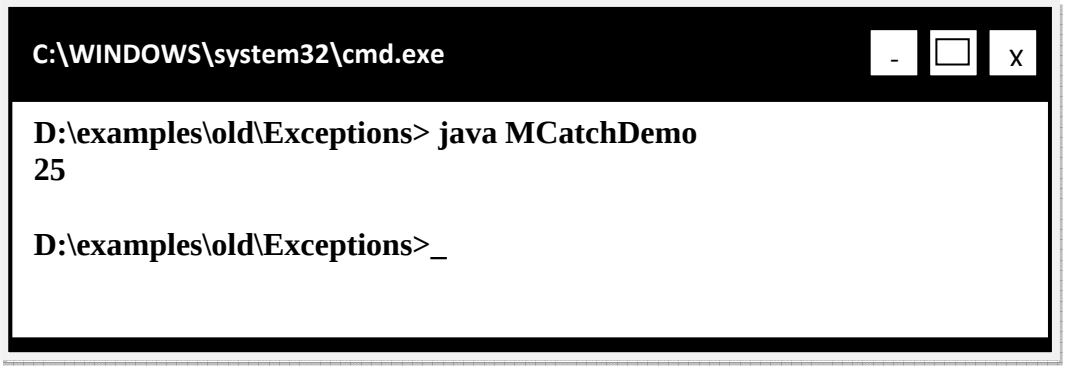
//read the number form file
String s = br.readLine();
//may throws NumberFormatException, if s is not a no.
int number = Integer.parseInt(s);
System.out.println(number * number);
}
catch( NumberFormatException nfEx) {
System.out.println(nfEx);
}
catch( FileNotFoundException fnfEx) {
System.out.println(fnfEx);
}
catch( IOException ioEx) {
System.out.println(ioEx);
}
}
}
```

We read everything from a file (numbers, floating values or text) as a String. That's why we first convert it to number and then print its square on console.

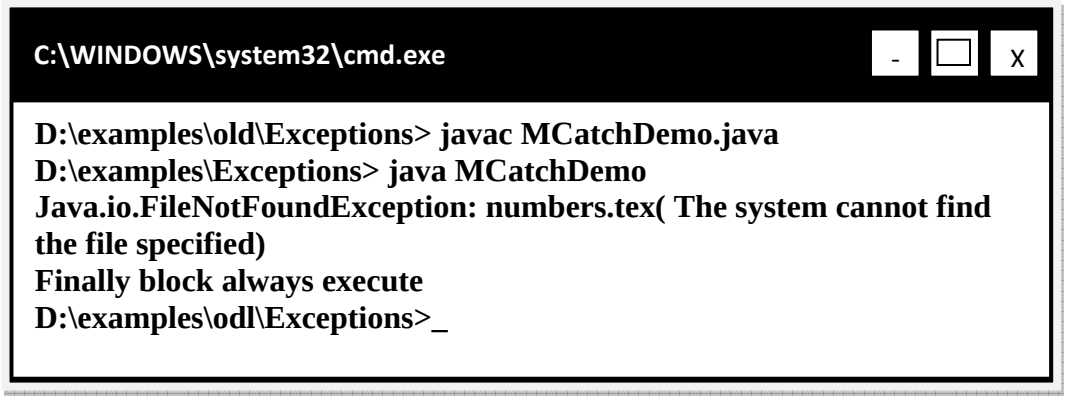
Compile & Execute

If file *numbers.txt* is not present in the same directory, the FileNotFoundException would be

thrown during execution.



If *numbers.txt* present in the same directory and contains a number, than hopefully no exception would be thrown.



7.13 The throws clause

The following code examples will introduce you with writing & using throws clause.

Example Code: ThrowsDemo.java

The ThrowsDemo.javacontains two methods namely method1 & method2and one main method. The mainmethod will make call to method1and than method1 will call method2. The method2 contains the file reading code. The program looks like one given below

```
// File ThrowsDemo.java
import java.io.* ;
public class ThrowsDemo {

// contains file reading code
public static void method2( ) {
```

```
try{
FileReader fr = new FileReader (“strings.txt”);
BufferedReader br = new BufferedReader (fr);

//read the line form file
String s = br.readLine();
System.out.println(s);
}
catch( IOException ioEx) {
ioEx.printStackTrace(); }
}// end method 2
//only calling method2
public static void method1( )
{
method2();
}
public static void main (String args[ ]) {
ThrowsDemo.method1();
}
}
```

7.14 printStackTrace method

- Defined in the Throwableclass - superclass of Exception & Error classes
- Shows you the full method calling history with line numbers.
- extremely useful in debugging

Modify: ThrowsDemo.java

- Let method2 doesn’t want to handle exception by itself, so it throws the exception to the caller of method2i.e. method1
- So method1 either have to handle the incoming exception or it can re-throw it to its caller i.e. main.
- Let method1 is handling the exception, so method1& method2 would be modified as:

```
// File ThrowsDemo.
java import java.io.* ;
public class ThrowsDemo {
// contains file reading code
public static void method2( ) throws IOException{
FileReader fr = new FileReader (“strings.txt”);
BufferedReader br = new BufferedReader (fr);

//read the line form file
```

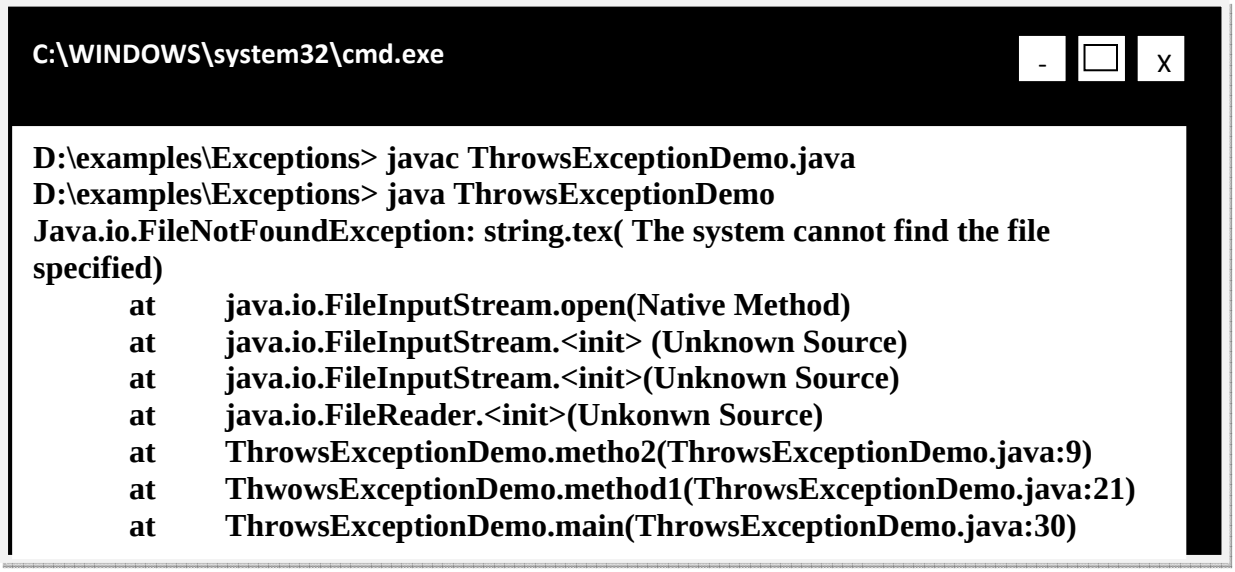
```
String s = br.readLine();
System.out.println(s);
}// end method 2

// calling method2 & handling incoming exception
public static void method1( ) {
try {
method2();
}
catch (IOException ioEx) {
ioEx.printStackTrace(); }
}

public static void main (String args[ ]) {
ThrowsDemo.method1();
}
}
```

Compile & Execute

If file *strings.txt* is not present in the same directory, method2will throw an exception that would be caught by method1and the printStackTracemethod will print the full calling history on console. The above scenario is shown in the output below:



If file *strings.txt* exist there, than hopefully line would be displayed on the console.

7.15 Reference

- Example code, their explanations and corresponding figures for this handout are taken from the book JAVA A Lab Course by Umair Javed. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose without the consent of author.

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 8: Streams

I/O libraries often use the abstraction of a *stream*, which represents any data source or sink as an object capable of producing or receiving pieces of data.

The Java library classes for I/O are divided by input and output. You need to import java.io package to use streams. There is no need to learn all the streams just do it on the need basis.

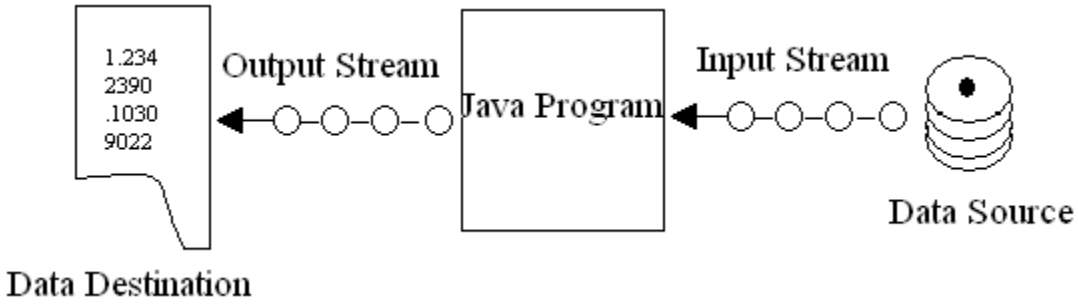
8.1 The concept of "streams"

- It is an abstraction of a data source/sink
- We need abstraction because there are lots of different devices (files, consoles, network, memory, etc.). We need to talk to the devices in different ways (sequential, random access, by lines, etc.) Streams make the task easy by acting in the same way for every device. Though inside handling of devices may be quite different, yet on the surface everything is similar. You might read from a file, the keyboard, memory or network connection, different devices may require specialization of the basic stream, but you can treat them all as just "streams". When you read from a network, you do nothing different than when you read from a local file or from users typing

```
//Reading from console
BufferedReader stdin =       new BufferedReader(new InputStreamReader(
System.in ));
-------- ( your console)
// Reading from file
BufferedReader br=new BufferedReader(new
FileReader("input.txt"));

//Reading from network
BufferedReader br = new BufferedReader(new InputStreamReader
(s.getInputStream()));
---- "s" is the socket
```

- So you can consider stream as a data path. Data can flow through this path in one direction between specified terminal points (your program and file, console, Socket etc.)



8.2 Stream classification based on Functionality

Based on functionality streams can be categorized as Node Stream and Filter Stream. Node Streams are those which connect directly with the data source/sick and provide basic functionality to read/write data from that source/sink

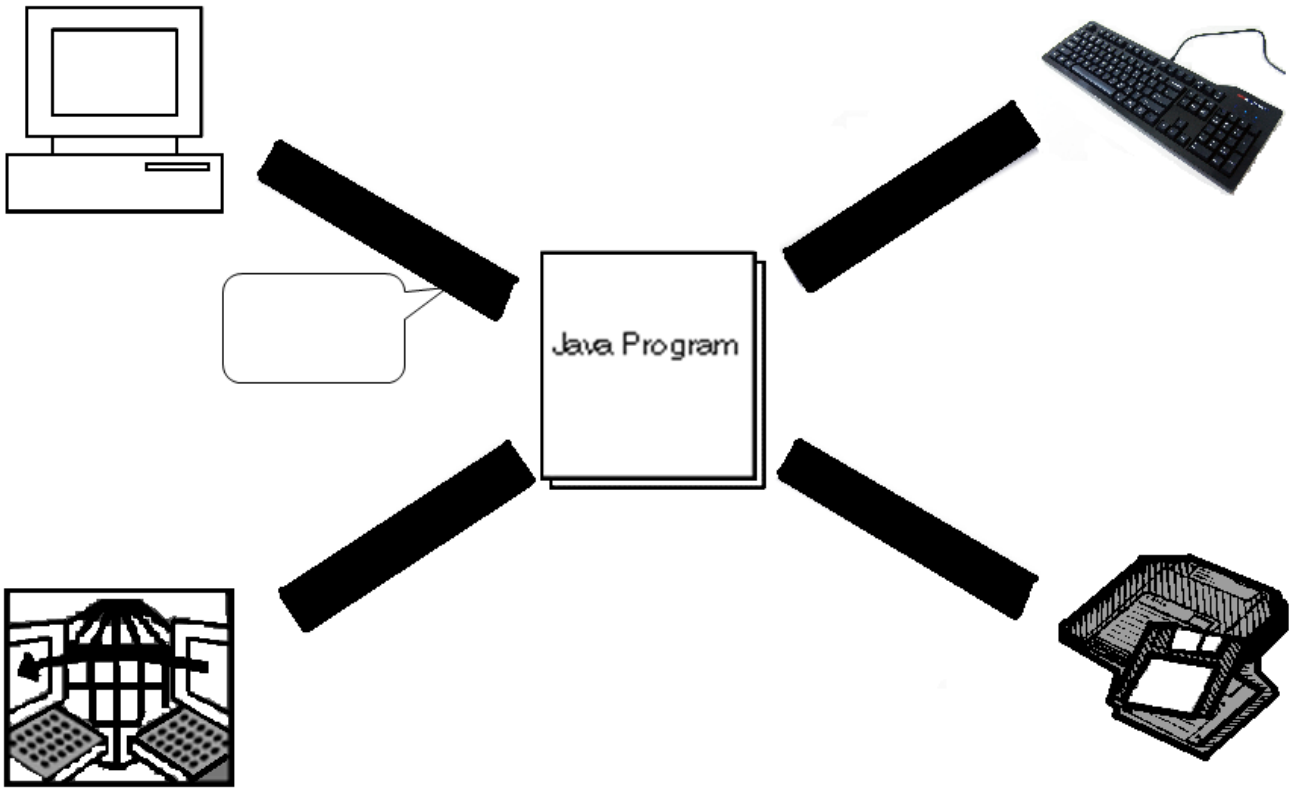
FileReader fr = new FileReader("input.txt");

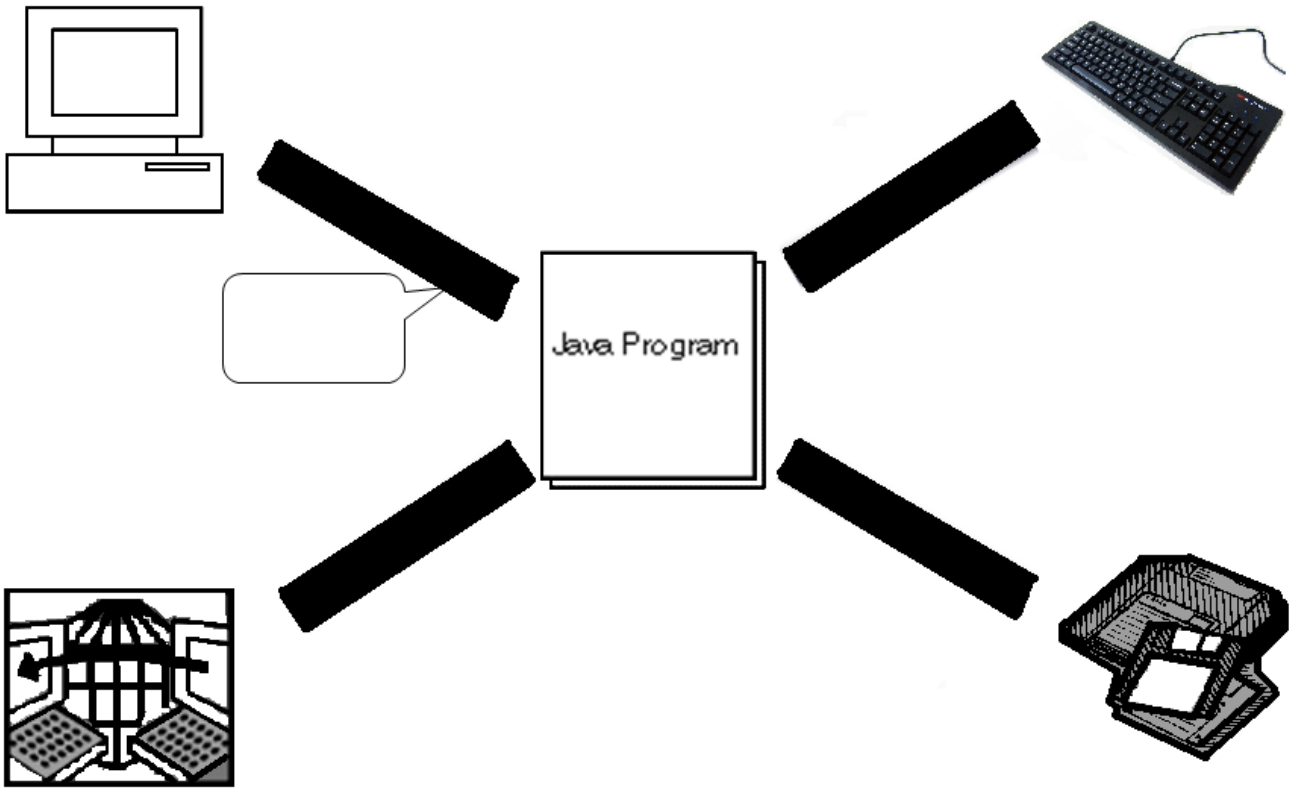
You can see that FileReader is taking a data/source "input.txt" as its argument and hence it is a node stream.

FilterStreams sit on top of a node stream or chain with other filter stream and provide some additional functionality e.g. compression, security etc. FilterStreams take other stream as their input.

BufferedReader bt = new BufferedReader(fr);

BufferedReader makes the IO efficient (enhances the functionality) by buffering the input before delivering. And as you can see that BufferedReader is sitting on top of a node stream which is FileReader.


Java Program

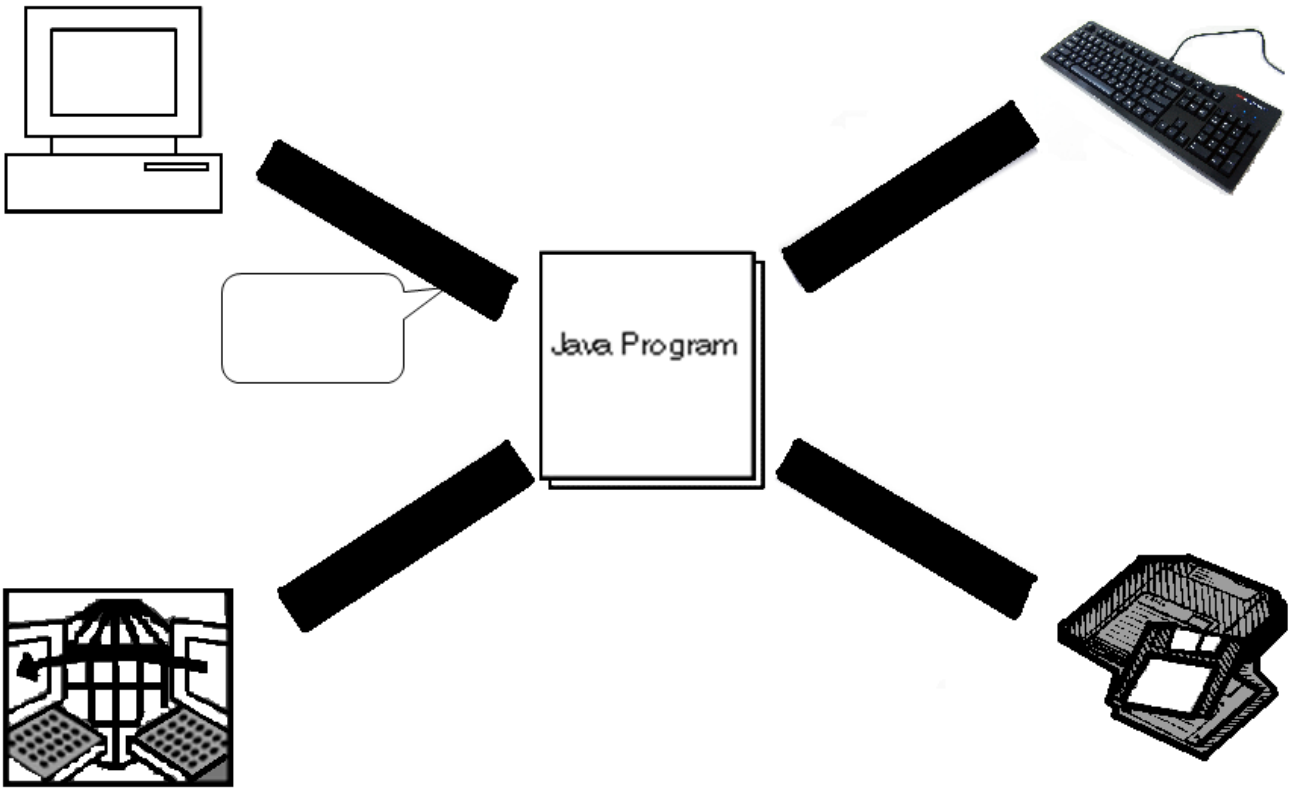


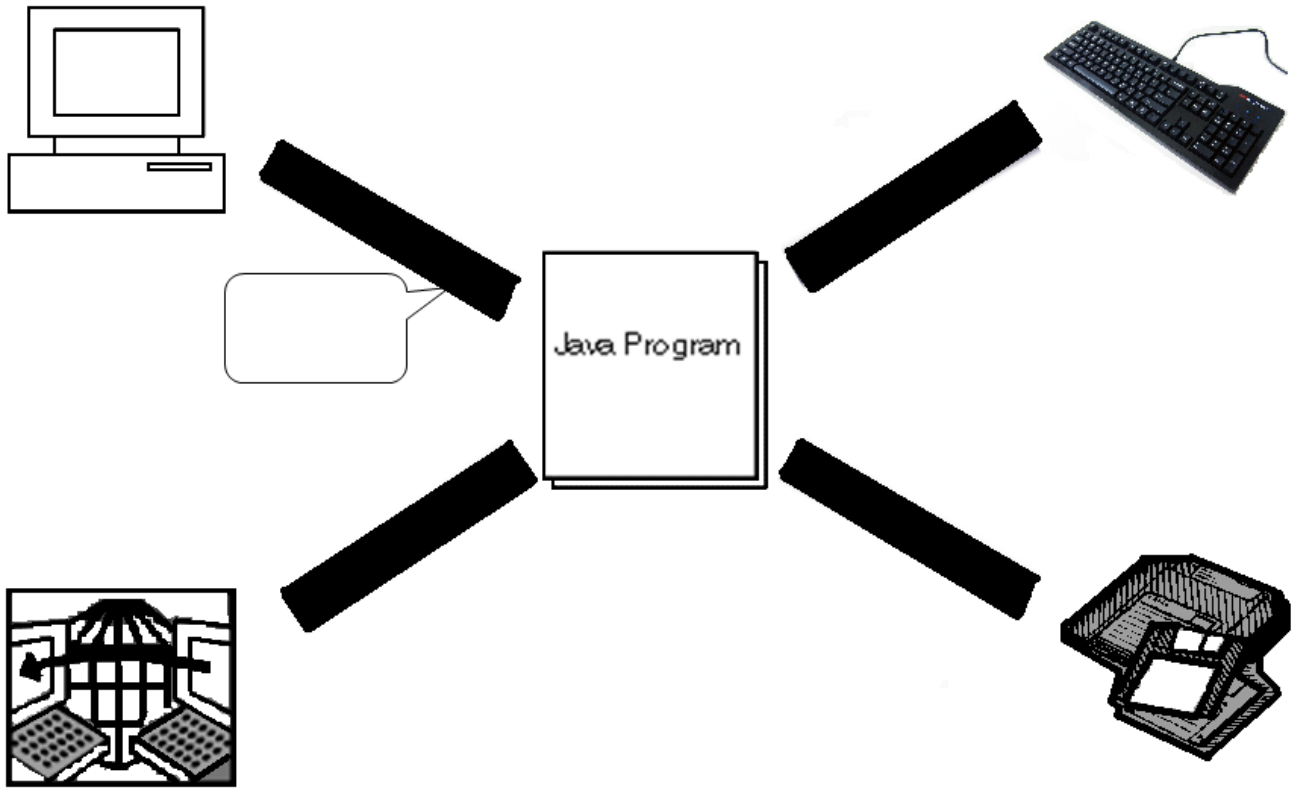
8.3 Stream classification based on data

Two types of classes exist:

Classes which contain the word stream in their name are byte oriented and are here since JDK1.0. These streams can be used to read/write data in the form of bytes. Hence classes with the word stream in their name are byte-oriented in nature. Examples of byte oriented streams are FileInputStream, ObjectOutputStream etc.

Classes which contain the word Reader/Writer are character oriented and read and write data in the form of characters. Readers and Writers came with JDK1.1. Examples of Reader/Writers are FileReader, PrintWriter etc


Java Program


Java Program

Example Code 8.1: Reading from File

The ReadFileEx.javareads text file line by line and prints them on console. Before we move on to the code, first create a text file (*input.txt*) using notepad and write following text lines inside it.

Text File: input.txt

```
Hello World
Pakistan is our homeland
Web Design and Development
```

```
// File ReadFileEx.
java import java.io.*;
public class ReadFileEx {
public static void main (String args[ ]) {
FileReader fr = null;
BufferedReader br = null;
try {
// attaching node stream with data source
fr = new FileReader("input.txt");

// attatching filter stream over node stream
br = new BufferedReader(fr);

// reading first line from file
String line = br.readLine();

// printing and reading remaining lines
while (line != null){
System.out.println(line);
line = br.readLine();
}
// closing streams
br.close();
fr.close();

}catch(IOException ioex){
System.out.println(ioex);
}
} // end main
} // end class
```

Example Code 8.2: Writing to File

The WriteFileEx.javawrites the strings into the text file named “output.txt”. If “output.txt” file does not exist, the java will create it for you.

```
// File WriteFileEx.java

 import java.io.*;
public class WriteFileEx {
public static void main (String args[ ]) {
FileWriter fw = null;
PrintWriter pw = null;

try {
// attaching node stream with data source
// if file does not exist, it automatically creates it
fw = new FileWriter (“output.txt”);

// attatching filter stream over node stream
pw = new PrintWriter(fw);

String s1 = “Hello World”;
String s2 = “Web Design and Development”;
// writing first string to file
pw.println(s1);
// writing second string to file
pw.println(s2);
// flushing stream
pw.flush();
// closing streams
pw.close();
fw.close();

}catch(IOException ioex){
System.out.println(ioex);
}
} // end main
} // end class
```

After executing the program, check the output.txt file. Two lines will be written there.

8.4 Reference

- Example code, their explanations and corresponding figures for this handout are taken from the book JAVA A Lab Course by Umair Javed. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose without the consent of author.

8.5 Modification of Address Book Code

8.5.1 Adding Persistence Functionality

Hopefully, your address book you built previously is giving you the required results except one i.e. persistence. You might have noticed that after adding some person records in the address book; if you exit from the program next time on re-executing address book all the previous records are no more available.

To overcome the above problem, we will modify our program so that on exiting/starting of address book, all the previously added records are available each time. To achieve this, we have to provide the persistence functionality. Currently, we will accomplish this task by saving person records in some text file.

Supporting simple persistence by any application requires handling of two scenarios. These are

- On start up of application - data (person records) must be read from file
- On end/finish up of application - data (person records) must be saved in file

To support persistence, we have to handle the above mentioned scenarios

8.5.1.1 Scenario 1 - Start Up

- Establish a data channel with a file by using streams
- Start reading data (person records) from file line by line
- Construct PersonInfoobjects from each line you have read
- Add those PersonInfoobjects in arraylist persons.
- Close the stream with the file
- Perform these steps while application is loading up

We will read records from a text file named persons.txt. The person records will be present in the file in the following format.

```
Ali, defence, 9201211
Usman, gulberg, 5173940
Salman, LUMS, 5272670
```

persons.txt

As you have seen, each person record is on a separate line. Person's name, address & phone number is separated using comma (,).

We will modify our AddressBook.javaby adding a new method loadPersons into it. This method will provide the implementation of all the steps. The method is shown below:

```
public void loadPersons ( ){

String tokens[] = null;
String name, add, ph;

try {

FileReader fr = new FileReader("persons.txt");
BufferedReader br = new BufferedReader(fr);

String line = br.readLine();
while ( line != null ) {
tokens = line.split(",");
name = tokens[0];
add = tokens[1];
ph = tokens[2];
PersonInfo p =        new PersonInfo(name, add, ph);
persons.add(p);
line = br.readLine();
}

br.close();
fr.close();

}catch(IOException ioEx){
System.out.println(ioEx);
}
}
```

- First, we have to connect with the text file in order to read line by line person records from it. This task is accomplished with the following lines of code:

  FileReader fr = new FileReader("persons.txt");

  BufferedReader br = new BufferedReader(fr);

FileReaderis a character based (node) stream that helps us in reading data in the form of characters. As we are using streams, so we have to import the java.iopackage in the AddressBookclass.

- We passed the file name persons.txtto the constructor of the FileReader.

Next we add BufferedReader (filter stream) on top of the FileReader because BufferedReader facilitates reading data line by line. (As you can recall from the lecture that filter streams are attached on top of node streams). That's why the constructor of BufferedReader is receiving the fr - the FileReader object.

- The next line of code will read line from file by using readLine() method of BufferedReaderand save it in a string variable called line.

```
String line = br.readLine();
```

- After that while loop starts. The condition of while loop is used to check whether the file is reached to end (returns null) or not. This loop is used to read whole file till the end. When end comes (null), this loop will finish.

```
while (line != null)
```

- Inside loop, the first step we performed is tokenizing the string. For this purpose, we have used split method of String class. This method returns substrings (tokens) according to the regular expression or delimiter passed to it.

```
tokens = line.split(",");
```

The return type of this method is array of strings that's why we have declared tokens as a String array in the beginning of this method as

```
String tokens[];
```

For example, the line contains the following string

```
Ali,defence,9201211
```

Now by calling split(",") method on this string, this method will return back three substrings *ali defence* and *9201211* because the delimiter we have passed to it is comma. The delimiter itself is not included in the substrings or tokens.

- The next three lines of code are simple assignments statements. The tokens[0]contains the *name* of the person because the name is always in the beginning of the line, tokens[1] contains *address* of the person and tokens[2] contains the phone number of the person.

```
name = tokens[0]; add=
tokens[1]; ph= tokens[2];
```

The name, add and ph are of type String and are declared in the beginning of this method.

- After that we have constructed the object of PersonInfo class by using parameterized constructor and passed all these strings to it.

```
PersonInfo p = new PersonInfo(name, add, ph);
```

- Afterward the PersonInfo object's p is added to the arraylist i.e. persons.

```
persons.add(p);
```

- The last step we have done inside loop is that we have again read a line from the file by using the readLine()method.
- By summarizing the task of while loop we can conclude that it reads the line from a file,tokenize that line into three substrings followed by constructing the PersonInfo object by using these tokens. And adding these objects to the arraylist. This process continues till the file reaches its end.
  The last step for reading information from the file is ordinary one - closing the streams, because files are external resources, so it's better to close them as soon as possible. Also observe that we used try/catch block because using streams can result in raising exceptions that falls under the checked exceptions category - that needs mandatory handling.
- The last important step you have to perform is to call this method while loading up.
- The most appropriate place to call this method is from inside the constructor of AddressBook.java. So the constructor will now look like similar to the one given below:

```
.....
......
public AddressBook () { Persons = new
ArrayList(); loadPersons();
}
.....
......
```

AddressBook.java

8.5.1.2 Scenario 2 - End/Finish Up

- Establish a datachanel(stream) with a file by using streams
- Take out PersonInfoobjects from ArrayList (persons)
- Build a string for each PersonInfo object by inserting commas (,) between name & address and address & phone number.
- Write the constructed string to the file
- Close the connection with file
- Perform these steps while exiting from address book.

Add another method savePersons into AddressBook.java. This method will provide the implementation of all the above mentioned steps. The method is shown below:

```
public void savePersons ( ){
try {
PersonInfo p;
```

```
String line;
FileWriter fw = new FileWriter("persons.txt");
PrintWriter pw = new PrintWriter(fw);
for(int i=0; i<persons.size(); i++)
{
p = (PersonInfo)persons.get(i);
line = p.name +","+ p.address +","+ p.phoneNum;

// writes line to file (persons.txt)
pw.println(line);
} pw.flush();
pw.close();
fw.close();
}
catch(IOException ioEx){
System.out.println(ioEx);
}
}
```

- As you can see, that we have opened the same file (persons.txt)again by using a set of streams.
- After that we have started for loop to iterate over arraylist as we did in searchPersonand deletePersonmethods.
- Inside forloop body, we have taken out PersonInfoobject and after type casting it we have assigned its reference to a PersonInfotype local variable p. This is achieved by the help of following line of code

```
p = (PersonInfo)persons.get(i);
```

- Next we build a string and insert commas between the PersonInfoattributes and assign the newly constructed string to string's local variable lineas shown in the following line of code.

```
line = p.name +","+ p.address +","+ p.phoneNum;
```

Note: Since, we haven't declare PersonInfo attributes private, therefore we are able to directly access them inside AddressBook.java.

- The next step is to write the line representing one PersonInfo object's information, to the file. This is done by using printlnmethod of PrintWriteras shown below

```
pw.println(line);
```

After writing line to the file, the printlnmethod will move the cursor/control to the next line. That's why each line is going to be written on separate line.

- The last step for saving information to the file is ordinary one - closing the streams but before that notice the code line that you have not seen/performed while loading persons records from file. That is

```
pw.flush( );
```

The above line immediately flushes data by writing any buffered output/data to file. This step is necessary to perform or otherwise you will most probably lose some data for the reason that PrintWriter is a Buffered Stream and they have their own internal memory/storage capacity for efficiency reasons. Buffered Streams do not send the data until their memory is full.

- Also we have written this code inside try-catch block.
- The last important step you have to perform is to call this method before exiting from the address book. The most appropriate place to call this method is under case4(exit scenario) in Test.java. So the case 4will now look like similar to the one given below:

```
……………
case 4:
ab.savePersons();
System.exit(0);
……………….
```

Test.java

Compile & Execute

Now again after compiling all the classes, run the Test class. Initially we are assuming that out persons.txt file is empty, so our arraylist persons will be empty on the first start up of address book. Now add some records into it, perform search or delete operations. Exit from the address book by choosing option 4. Check out the persons.txt file. Don't get surprised by seeing that it contains all the person records in the format exactly we have seen above.

Next time you will run the address book; all the records will be available to you. Perform the search or delete operation to verify that. Finally you have done it!!!

8.6 References

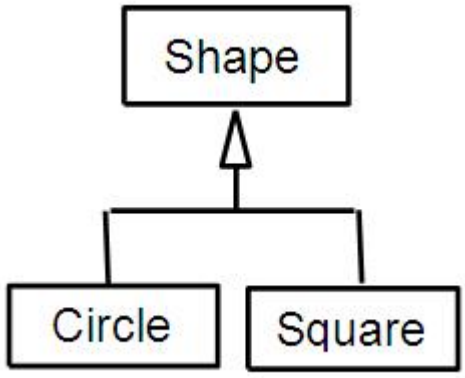
- Example code, their explanations and corresponding figures for this handout are taken from the book JAVA A Lab Course by Umair Javed. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose without the consent of author.

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 9: Abstract Classes and Interfaces

9.1 Problem and Requirements

Before moving on to abstract classes, first examine the following class hierarchy shown below:



- Suppose that in order to exploit polymorphism, we specify that 2-D objects must be able to compute their area.
    - All 2-D classes must respond to area() message.
- How do we ensure that?
    - Define area method in class Shape
    - Force the subclasses of Shape to respond area() message
- Java's provides us two solutions to handle such problem
    - Abstract Classes
    - Interfaces

9.2 Abstract Classes

Abstract classes are used to define only part of an implementation. Because, information is not complete therefore an abstract class cannot be instantiate. However, like regular classes, they can also contain instance variables and methods that are fully implemented. The class that inherits from abstract class is responsible to provide details.

Any class with an abstract method (a method has no implementation similar to pure virtual function in C++) must be declared abstract, yet you can declare a class *abstract* that has no abstract method.

If subclass overrides all abstract methods of the superclass, than it becomes a concrete (a class whose object can be instantiate) class otherwise we have to declare it as abstract or we cannot compile it.

The most important aspect of abstract class is that reference of an abstract class can point to the object of concrete classes.

Code Example of Abstract Classes

The **Shape** class contains an abstract method calculateArea() with no definition.

```
public abstract class Shape{
public abstract void calculateArea();
}
```

Class **Circle** extends from abstract Shape class, therefore to become concrete class it must provides the definition of calculateArea() method.

```
public class Circle extends Shape {
private int x, y;
private int radius;

public Circle() {
x = 5;
y = 5;
radius = 10;
}

// providing definition of abstract method
public void calculateArea () {
double area = 3.14 * (radius * radius);
System.out.println("Area: " + area);
}
}//end of class
```

The Test class contains main method. Inside main, a reference s of abstract Shape class is created. This reference can point to Circle (subclass of abstract class Shape) class object as it is a concrete class. With the help of reference, method calculateArea() can be invoked of Circle class. This is all shown in the form of code below:

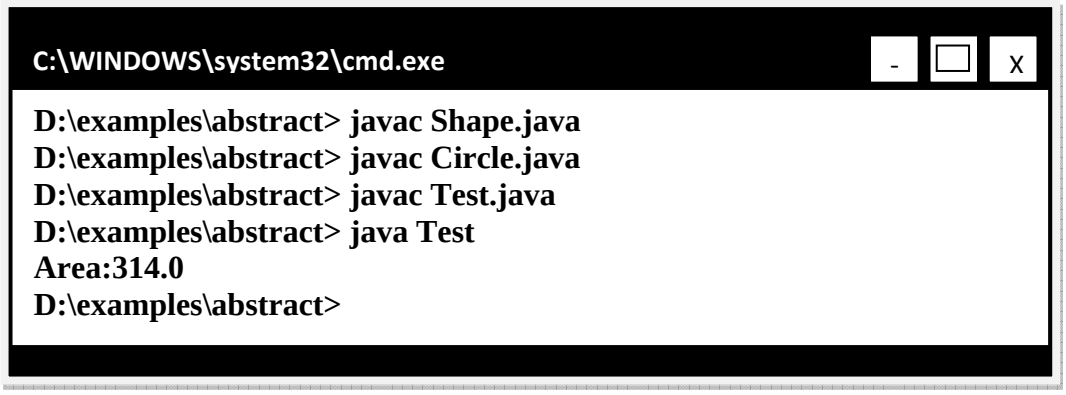
```
public class Test {
```

```
//can only create references of A.C.

Shape s = null;

//Shape s1 = new Shape(); //cannot instantiate
//abstract class reference can point to concrete subclass
s = new Circle();
s.calculateArea();
}
}//end of class
```

The compilation and execution of the above program is shown below:



9.3 Interfaces

As we seen one possible java's solution to problem discussed in start of the tutorial. The second possible java's solution is Interfaces.

Interfaces are special java type which contains only a set of method prototypes, but does not provide the implementation for these prototypes. All the methods inside an interface are abstract by default thus an interface is tantamount to a pure abstract class - a class with zero implementation. Interface can also contains static final constants

9.3.1 Defining an Interface

Keyword interface is used instead of class as shown below:

```
public interface Speaker{
public void speak();
}
```

9.3.2 Implementing (using) Interface

Classes *implement* interfaces. Implementing an interface is like *signing a contract*. A class that implements an interface will have to provide the definition of all the methods that are present inside an interface. If the class does not provide definitions of all methods, the class would not compile. We have to declare it as an abstract class in order to get it compiled.

Relationship between a class and interface is equivalent to *"responds to"* while *"is a"* relationship exists in inheritance.

Code Example of Defining & Implementing an Interface The interface **Printable** contains print() method.

```
public interface Printable{
public void print();
}
```

Class **Student** is implementing the interface Printable. Note the use of keyword *implements* after the class name. Student class has to provide the definition of print method or we are unable to compile.

The code snippet of student class is given below:

```
public class Student implements Printable {
private String name;
private String address;
public String toString () {
return "name:"+name +" address:"+address;
}
//providing definition of interface's print method
public void print() {
System.out.println("Name:" +name+" address"+address);
}
}//end of class
```

9.4 Interface Characteristics

Similar to abstract class, interfaces imposes a design structure on any class that uses the

interface. Contrary to inheritance, a class can implement more than one interfaces. To do this separate the interface names with comma. This is java's way of multiple inheritance.

class Circle implements Drawable , Printable { ……. }

Objects of interfaces also cannot be
instantiated.

Speaker s = new Speaker(); // not compile

However, a reference of interface can be created to point any of its implementation class. This is interface based polymorphism.

Code Example: Interface based polymorphism

Interface Speaker is implemented by three classes Politician, Coach and Lecturer. Code snippets of all these three classes are show below:

```
public class Politician implements Speaker{
public void speak(){
System.out.println("Politics Talks");
}
}
```

```
public class Coach implements Speaker{
public void speak(){
System.out.println("Sports Talks");
}
}
```

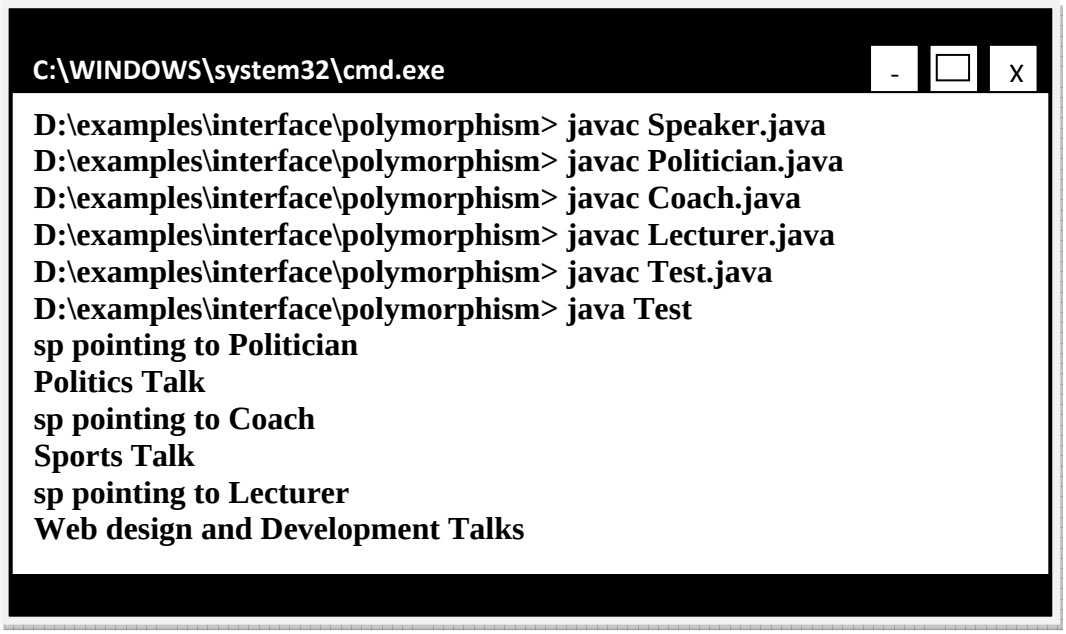
```
public class Lecturer implements Speaker{
public void speak(){
System.out.println("Web Design and Development Talks");
}
}
```

As usual, Test class contains main method. Inside main, a reference spis created of Speaker class. Later, this reference is used to point to the objects of Politician, Coach and Lecturer class. On calling speak method with the help of sp, will invoke the method of a class to which sp is pointing.

```
public class Test{
public static void main (String args[ ]) {
Speaker sp = null;
System.out.println("sp pointing to Politician");
sp = new Politician();
```

```
sp.speak();
System.out.println("sp pointing to Coach");
sp = new Coach();
sp.speak();
System.out.println("sp pointing to Lecturer");
sp = new Lecturer();
sp.speak();
}
}
```

The compilation and execution of the above program is shown below:



9.5 References

- Example code, their explanations and corresponding figures for this handout are taken from the book JAVA A Lab Course by Umair Javed. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose without the consent of author.

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 10: Graphical User Interfaces

A graphical user interface is a visual interface to a program. GUIs are built from GUI components (buttons, menus, labels etc). A GUI component is an object with which the user interacts via the mouse or keyboard.

Together, the appearance and how user interacts with the program are known as the program look and feel.

10.1 Support for GUI in Java

The classes that are used to create GUI components are part of the "java.awt" or "javax.swing" package. Both these packages provide rich set of user interface components.

10.2 GUI classes vs. Non-GUI Support Classes

The classes present in the awt and swing packages can be classified into two broad categories. GUI classes & Non-GUI Support classes.

The GUI classes as the name indicates are visible and user can interact with them. Examples of these are JButton, JFrame & JRadioButton etc

The Non-GUI support classes provide services and perform necessary functions for GUI classes. They do not produce any visual output. Examples of these classes are Layout managers (discussed latter) & Event handling (see handout on it) classes etc.

10.3 java.awt package

AWT stands for "Abstract Windowing Toolkit" contains original GUI components that came with the first release of JDK. These components are tied directly to the local platform's (Windows, Linux, MAC etc) graphical user interface capabilities. Thus results in a java program executing on different java platforms (windows, linux, solaris etc) has a different appearance and sometimes even different user interaction on each platform.

AWT components are often called Heavy Weight Components (HWC) as they rely on the local platform's windowing system to determine their functionality and their look and feel. Every time you create an AWT component it creates a corresponding process on the operating system. As

compared to this SWING components are managed through threads and are known as Light Weight Components.

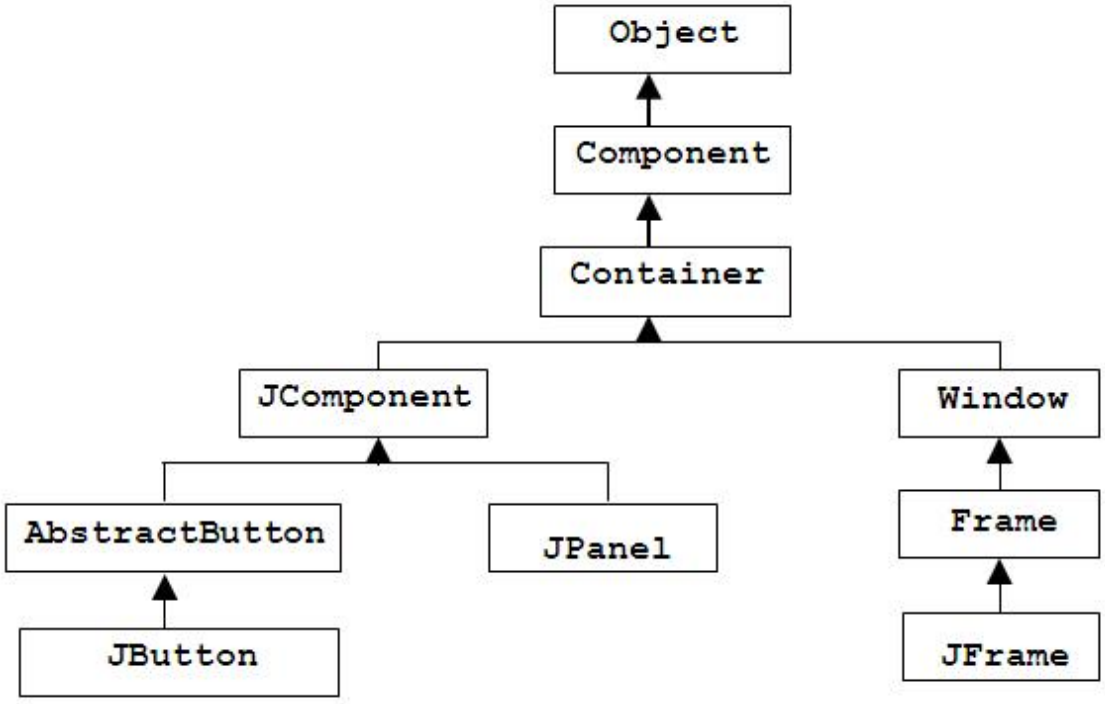
This package also provides the classes for robust event handling (see handout on it) and layout managers.

10.4 javax.swing package

These are the newest GUI components. Swing components are written, manipulated and displayed completely in java, therefore also called pure java components. The swing components allow the programmer to specify a uniform look and feel across all platforms.

Swing components are often referred to as Light Weight Components as they are completely written in java. Several swing components are still HWC e.g. JFrame etc.

10.5 A part of the Framework



10.6 GUI Creation Steps

10.6.1 import required packages

- import java.awt.* and/or javax.swing.* package.

10.6.2 Setup the top level containers

- A container is a collection of related components, which allows other components to be nested inside it. In application with JFrame, we attach components to the content pane - a container.
- Two important methods the container class has **add** and **setLayout.**

- The add method is used for adding components to the content pane while setLayout method is used to specify the layout manager.
- Container are classified into two broad categories that are Top Level Containers and General Purpose Containers
- Top level containers can contain (add) other containers as well as basic Components (buttons, labels etc) while general purpose containers are Typically used to collect basic components and are added to top level containers.
- General purpose containers cannot exist alone they must be added to top level containers
- Examples of top level container are JFrame, Dialog and Applet etc. Our application uses one of these.
- Examples of general purpose container are JPanel, Toolbar and ScrollPane etc.
- So, take a top level container and create its instance. Consider the following code of line if JFrame is selected as a top level container

JFrame *frame* = new JFrame();

10.6.3 Get the component area of the top level container

- Review the hierarchy given above, and observe that JFrame *is a* frame *is a* window. So, it can be interpreted as JFrame *is a* window.
- Every window has two areas. System Area & Component Area
- The programmer cannot add/remove components to the System Area.
- The Component Area often known as Client area is a workable place for the programmer. Components can be added/removed in this area.
- So, to add components, as you guessed right component area of the JFrame is required. It can be accomplished by the following code of line

  Container *con* = *frame*.getContentPane();
- *frame* is an instance of JFrame and by calling *getContentPane()* method on it, it returns the component area. This component area is of type container and that is why it is stored in a variable of a Container class. As already discussed, container allows other components to be added / removed.

10.6.4 Apply layout to component area

- The layout (size & position etc. How they appear) of components in a container is usually governed by Layout Managers.
- The layout manager is responsible for deciding the layout policy and size of its components added to the container.
- Layout managers are represented in java as classes. (Layout Managers are going to be discussed in detail later in this handout)
- To set the layout, as already discussed use setLayout method and pass object of layout manager as an argument.

```
con.setLayout( new FlowLayout( ) );
```

- We passed an object of FlowLayout to the setLayout method here.
- We can also use the following lines of code instead of above.

```
FlowLayout layout = new FlowLayout();

con.setLayout(layout);
```

10.6.5 Create and Add components

- Create required components by calling their constructor.

```
JButton button = new JButton ( );
```

- After creating all components you are interested in, the next task is to add these components into the component area of your JFrame (i.e ContentPane, the reference to which is in variable con of type Container)
- Use *add* method of the Container to accomplish this and pass it the component to be added.

```
con.add(button);
```

10.6.6 Set size of frame and make it visible

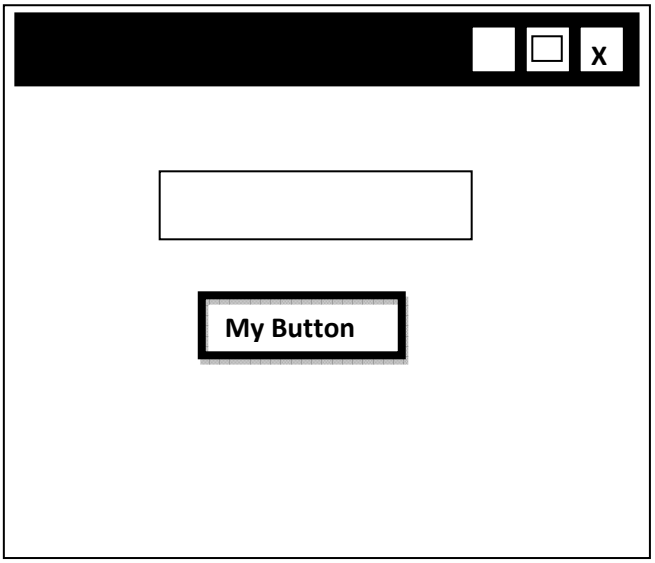
- A frame must be made visible via a call to setVisible(true) and its size defined via a call setSize(rows in pixel, columns in pixel) to be displayed on the screen.

```
frame.setSize(200,300) ;
frame.setVisible(true) ;
```

Note: By default, all JFrame's are invisible. To make visible frame visible we have passed *true* to the setVisible method.

```
frame.setVisible(false) ;
```

Example: Making a Simple GUI



The above figured GUI contains one text field and a button. Let's code it by following the six GUI creation steps we discussed.

Code for Simple GUI:

```
// File GUITest.java

//Step 1: import packages
import java.awt.*;
import javax.swing.*;

public class GUITest {
JFrame myFrame ;

//method used for setting layout of GUI
public void initGUI ( ) {

//Step 2: setup the top level container

myFrame = new JFrame();

//Step 3: Get the component area of top-level container
Container c = myFrame.getContentPane();

//Step 4: Apply layouts
c.setLayout( new FlowLayout( ) );
```

```
//Step 5: create & add components
JTextField tf = new JTextField(10);
JButton b1 = new JButton("My Button");

c.add(tf);
c.add(b1);

//Step 6: set size of frame and make it visible
myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
myFrame.setSize(200,150);
myFrame.setVisible(true);
} //end initGUI method

public GUITest () {// default constructor
initGUI ();
}

public static void main (String args[ ]) {
GUITest gui = new GUITest();
}
} // end of class
```

10.7 Important Points to Consider

- *main* method (from where program execution starts) is written in the same class. The main method can be in a separate class instead of writing in the same class its your choice.
- Inside main, an object of GUI test class is created that results in calling of constructor of the class and from the constructor, *initGUI* method is called that is responsible for setting up the GUI.
- The following line of code is used to exit the program when you close the window

```
myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

If you delete this line and run your program, the desired GUI would be displayed. However if you close the window by using (X) button on top left corner of your window, you'll notice that the control doesn't return back to command prompt. The reason for this is that the java process is still running. However if you put this line in your code, when you exit your prompt will return.

10.8 References:

- Sun java tutorial: http://java.sun.com/docs/books/tutorial/java
- Thinking in java by Bruce Eckle
- Beginning Java2 by Ivor Hortan
- GUI creation steps are taken from the book Java A Lab Course by Umair Javed

10.9 Graphical User Interfaces - 2

10.9.1 Layout Managers

Layout Managers are used to form the appearance of your GUI. They are concerned with the arrangement of components of GUI. A general question is "why we cannot place components at our desired location (may be using the x,y coordinate position?"

The answer is that you can create your GUI without using Layout Managers and you can also do VB style positioning of components at some x,y co-ordinate in Java, but that is generally not advisable if you desire to run the same program on different platforms

The appearance of the GUI also depends on the underlying platform and to keep that same the responsibility of arranging layout is given to the LayoutManagers so they can provide the same look and feel across different platforms

Commonly used layout managers are

- Flow Layout
- Grid Layout
- Border Layout
- Box Layout
- Card Layout
- GridBag Layout and so on

Let us discuss the top three in detail one by one with code examples. These top three will meet most of your basic needs

10.9.1.1 Flow Layout

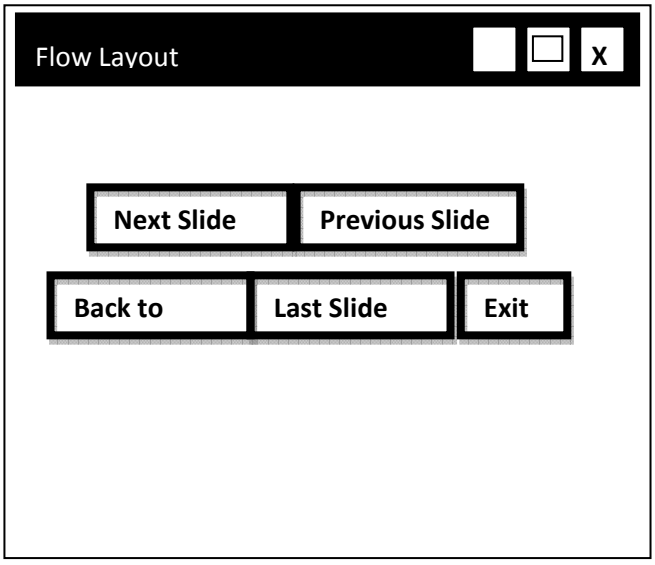
- Position components on line by line basis. Each time a line is filled, a new line is started.
- The size of the line depends upon the size of your frame. If you stretch your frame while your program is running, your GUI will be disturbed.

Example Code

```
// File FlowLayoutTest.
java import java.awt.*;
import javax.swing.*;
public class FlowLayoutTest {
JFrame myFrame ;
JButton b1, b2, b3, b4, b5;
//method used for setting layout of GUI
public void initGUI ( ) {
```

```
myFrame = new JFrame("Flow Layout");
Container c = myFrame.getContentPane();
c.setLayout( new FlowLayout( ) );
b1 =    new JButton("Next Slide");
b2 =    new JButton("Previous Slide");
b3 =    new JButton("Back to Start");
b4 =    new JButton("Last Slide");
b5 =    new JButton("Exit");
c.add(b1);
c.add(b2);
c.add(b3);
c.add(b4);
c.add(b5);
myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
myFrame.setSize(300,150);
myFrame.setVisible(true);
} //end initGUI method
public FlowLayoutTest () {    // default constructor
initGUI ();
}
public static void main (String args[ ]) {
FlowLayoutTest flTest = new FlowLayoutTest();
}
} // end of class
```

Output



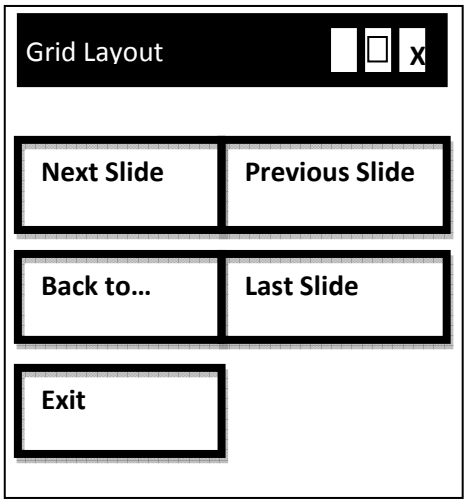
10.9.1.2 Grid Layout

- Splits the panel/window into a grid(cells) with given number of rows and columns.
- Forces the size of each component to occupy the whole cell. Size of each component is same .
- Components are added row wise. When all the columns of the first row are get filled the components are then added to the next row.
- Only one component can be added into each cell.

Example Code

```
// File GridLayoutTest.java
 import java.awt.*;
import javax.swing.*;
public class GridLayoutTest {
JFrame myFrame ;
JButton b1, b2, b3, b4, b5;
//method used for setting layout of GUI
public void initGUI ( ) {
myFrame = new JFrame("Grid Layout");
Container c = myFrame.getContentPane();
                                    // rows , cols
c.setLayout( new GridLayout( 3    , 2  ) );
b1 = new JButton("Next Slide");
b2 = new JButton("Previous Slide");
b3 = new JButton("Back to Start");
b4 = new JButton("Last Slide");
b5 = new JButton("Exit");
c.add(b1);
c.add(b2);
c.add(b3);
c.add(b4);
c.add(b5);
myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
myFrame.setSize(300,150);
myFrame.setVisible(true);
} //end initGUI method
public GridLayoutTest () {     // default constructor
initGUI ();
}
public static void main (String args[ ]) {
GridLayoutTest glTest = new GridLayoutTest();
}
} // end of class
```

Output

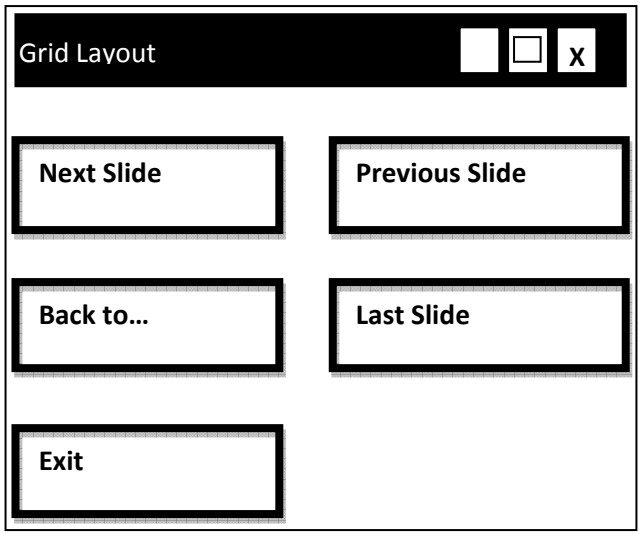


Modification

The grid layout also allows the spacing between cells. To achieve spacing between cells, modify the above program.
Pass additional parameters to the constructor of GridLayout, spaces between rows & spaces between columns as shown below

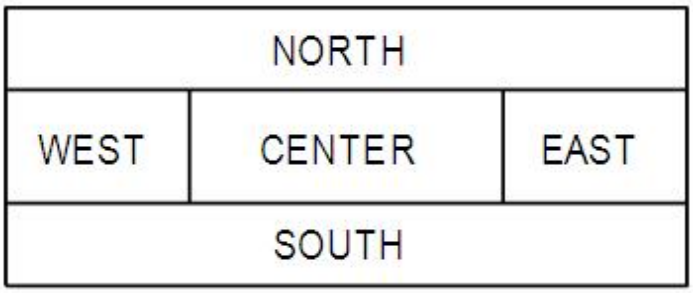
```
c.setLayout( new GridLayout( 3 , 2,10 ,  20)  );
```

The output is look similar to one given below.



10.9.1.3 Border Layout

- Divides the area into five regions. North, South, East, West and Center
- Components are added to the specified region
- If any region not filled, the filled regions will occupy the space but the center region will still appear as background if it contains no component.
- Only one component can be added into each region.



Example Code:

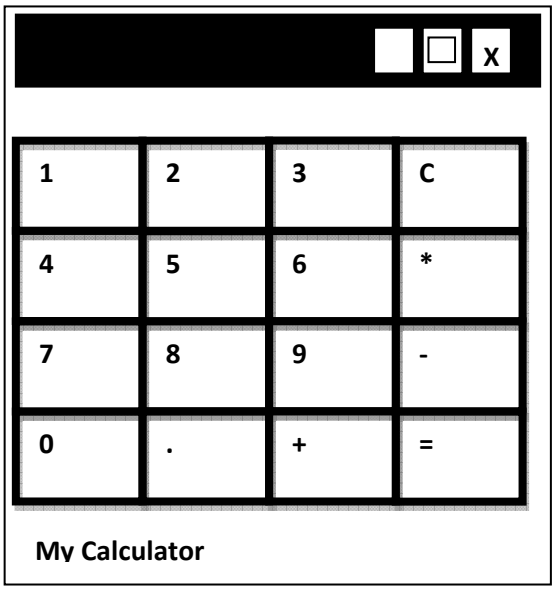
```
// File BorderLayoutTest.
java import java.awt.*;
import javax.swing.*;
public class BorderLayoutTest {
JFrame myFrame ;
JButton b1, b2, b3, b4, b5;
//method used for setting layout of GUI
public void initGUI ( ) {
myFrame = new JFrame("Border Layout");
Container c = myFrame.getContentPane();
c.setLayout( new BorderLayout( ) );
b1 =    new JButton("Next Slide");
b2 =    new JButton("Previous Slide");
b3 =    new JButton("Back to Start");
b4 =    new JButton("Last Slide");
b5 =    new JButton("Exit");
c.add( b1 , BorderLayout.NORTH );
c.add( b2 , BorderLayout.SOUTH );
c.add( b3 , BorderLayout.EAST );
c.add( b4 , BorderLayout.WEST );
c.add( b5 , BorderLayout.CENTER);
myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
myFrame.setSize(300,150);
myFrame.setVisible(true);
} //end initGUI method
public BorderLayoutTest () { // default constructor
initGUI ();
```

```
}
public static void main (String args[ ]) {
BorderLayoutTest glTest = new BorderLayoutTest();
}
} // end of class
```

Points to Remember

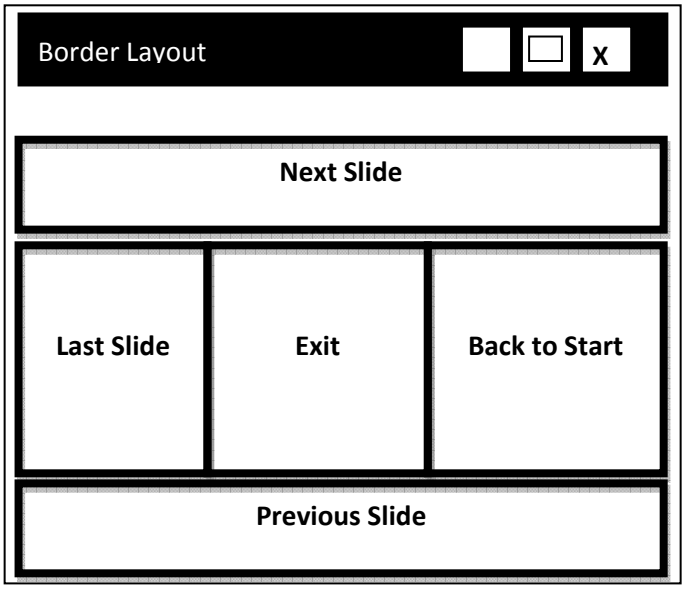
- Revisit the code of adding components, we specify the region in which we want to add component or otherwise they will not be visible.
- Consider the following segment of code: BorderLayout.NORTH, as you guessed correctly NORTH is a constant (final) defined in BorderLayout class public access modifier. Similarly the other ones are defined. Now you understand why so much emphasis has been made on following the naming conventions.

Output:



10.10 Making Complex GUIs

From the discussion above it seems that the basic Layout Managers may not help us in constructing complex GUIs, but generally a combination of these basic layouts can do the job. So let's try to create the calculator GUI given below



This GUI has 16 different buttons each of same size and text field on the top and a label 'my calculator' on the bottom.

So, how we can make this GUI? If Border Layout is selected, it has five regions (each region can have at most one component) but here we have more than five components to add. Lets try Grid Layout, but all the components in a Grid have same size and the text field at the top and label at the bottom has different size. Flow Layout cannot be selected because if we stretch our GUI it will destroy its shape.

Can we make this GUI? Yes, we can. Making of such GUI is a bit tricky business but General Purpose Containers are there to provide the solution.

10.10.1 JPanel

- It is general purpose container (can't exist alone, it has to be in some toplevel container) in which we can put in different components (JButton , JTextField etc even other JPanels)
- JPanel has its own layout that can be set while creating JPanel instance

```
JPanel myPanel = new JPanel ( new FlowLayout( ) );
```

- Add components by using *add* method like shown below.

```
myPanel.add (button);
```

- Must be added to a top level container (like JFrame etc) in order to be visible as they (general purpose containers) can't exist alone.

10.10.2 Solution

To make the calculator GUI shown above, take JFrame (top level container) and set its layout to border. Then take JPanel (general purpose container) and set its layout to Grid with 4 rows and 4 columns.

Add buttons to JPanel as they all have equal size and JPanel layout has been set to GridLayout. Afterthat, add text field to the north region, label to the south region and panel to the center region of the JFrame's container. The east and west regions are left blank and the center region will be stretched to cover up these. So, that's how we can build our calculator GUI.

Code for Calculator GUI

```
// File CalculatorGUI.java
 import java.awt.*;
import javax.swing.*;
public class CalculatorGUI {
JFrame fCalc;
JButton b1, b2, b3, b4, b5, b6, b7, b8, b9, b0;
JButton bPlus, bMinus, bMul, bPoint, bEqual, bClear;
JPanel pButtons;
JTextField tfAnswer;
JLabel lMyCalc;
//method used for setting layout of GUI
public void initGUI ( ) {
fCalc = new JFrame();
b0 = new JButton("0");
b1 = new JButton("1");
b2 = new JButton("2");
b3 = new JButton("3");
b4 = new JButton("4");
b5 = new JButton("5");
b6 = new JButton("6");
b7 = new JButton("7");
b8 = new JButton("8");
b9 = new JButton("9");
bPlus   = new JButton("+");
bMinus  = new JButton("-");
bMul    = new JButton("*");
bPoint  = new JButton(".");
bEqual  = new JButton("=");
bClear  = new JButton("C");
tfAnswer = new JTextField();
lMyCalc = new JLabel("My Clacualator");
//creating panel object and setting its layout
```

```
pButtons = new JPanel (new GridLayout(4,4));
//adding components (buttons) to panel
pButtons.add(b1);
pButtons.add(b2);
pButtons.add(b3);
pButtons.add(bClear);
pButtons.add(b4);
pButtons.add(b5);
pButtons.add(b6);
pButtons.add(bMul);
 pButtons.add(b7);
pButtons.add(b8);
 pButtons.add(b9);
 pButtons.add(bMinus);
 pButtons.add(b0);
pButtons.add(bPoint);
 pButtons.add(bPlus);
pButtons.add(bEqual);
// getting componenet area of JFrame
Container con = fCalc.getContentPane();
con.setLayout(new BorderLayout());
//adding components to container
con.add(tfAnswer, BorderLayout.NORTH);
con.add(lMyCalc, BorderLayout.SOUTH);
con.add(pButtons, BorderLayout.CENTER);
fCalc.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
fCalc.setSize(300, 300);
fCalc.setVisible(true);
} //end initGUI method
public CalculatorGUI () {       // default constructor
initGUI ();
}
public static void main (String args[ ]) {
CalculatorGUI calGUI = new CalculatorGUI ();
}
} // end of class
```

10.11 Reference:

- Sun java tutorial: http://java.sun.com/docs/books/tutorial/java
- Thinking in java by Bruce Eckle
- Beginning Java2 by Ivor Hortan
- Java A Lab Course by Umair Javed

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 11: Event Handling

One of the most important aspects of most non-trivial applications (especially UI type-apps) is the ability to respond to events that are generated by the various components of the application, both in response to user interactions and other system components such as client-server processing. In this handout we will look at how Java supports event generation and handling and how to create (and process) custom events.

GUIs generate events when the user interacts with GUI. For example,

- Clicking a button
- Moving the mouse
- Closing Window etc

Both AWT and swing components (not all) generate events

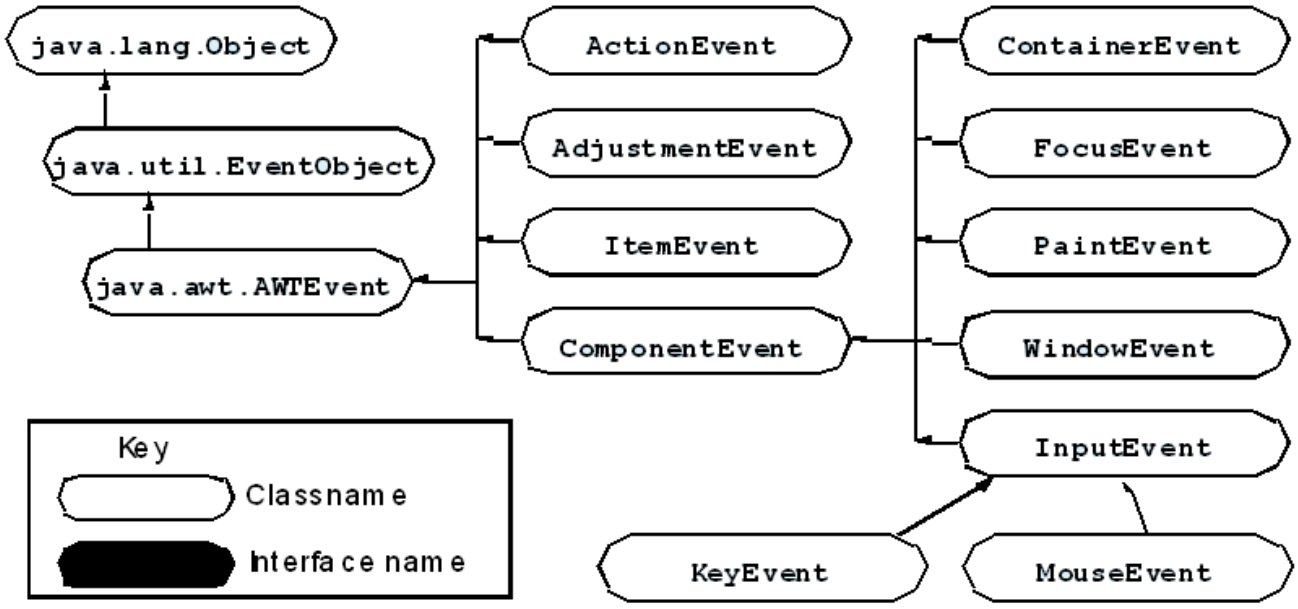
- java.awt.event.*;
- javax.swing.event.*;

In java, events are represented by Objects

These objects tell us about event and its source. Examples are:

- ActionEvent (Clicking a button)
- WindowEvent (Doing something with window e.g. closing , minimizing)

Some event classes of java.awt.event are shown in diagram below



11.1 Event Handling Model

In Java both AWT and Swing components use Event Delegation Model.

- In this model processing of an event is delegated to a particular object (handlers) in the program
- It's a Publish-Subscribe model. That is, event generating component publish an event and event handling components subscribe for that event. The publisher sends these events to subscribers. Similar to the way that you subscribe for newspaper and you get the newspaper at your home from the publisher.
- This model separates UI code from program logic, it means that we can create separate classes for UI components and event handlers and hence business/program logic is separated from GUI components.

11.2 Event Handling Steps

For a programmer the event Handling is a three step process in terms of code

- **Step 1:** Create components which can generate events (Event Generators)
- **Step 2:** Build component (objects) that can handle events (Event Handlers)
- **Step 3:** Register handlers with generators

11.3 Event Handling Process

11.3.1 Step 1: Event Generators

The first step is that you create an event generator. You have already seen a lot of event generators like:

- Buttons
- Mouse
- Key
- Window etc

Most of GUI components can be created by calling their constructors. For example

JButton b1 = new JButton("Hello");

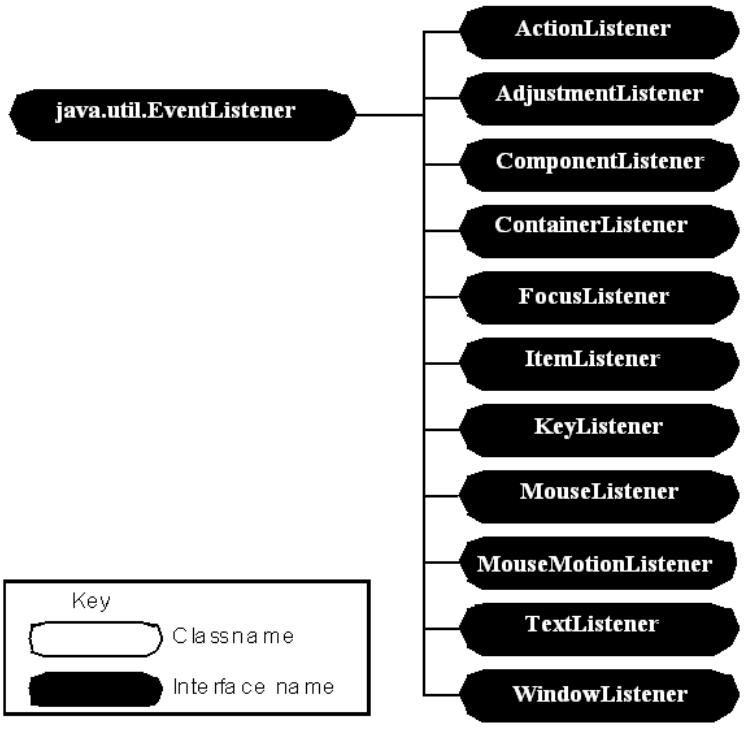
Now b1 can generate events

Note: We do not create Mouse/Keys etc as they are system components

11.3.2 Step 2: Event Handlers/ Event Listener

The second step is that you build components that can handle events

- First Technique - *By Implementing Listener Interfaces*
  - Java defines interfaces for every event type
  - If a class needs to handle an event. It needs to implement the corresponding listener interface
  - To handle "ActionEvent" a class needs to implement "ActionListener"
  - To handle "KeyEvent" a class needs to implement "KeyListener"
  - To handle "MouseEvent" a class needs to implement "MouseListener" and so on
  - Package java.awt.event contains different event Listener Interfaces which are shown in the following figure



Some Example Listeners, the way they are defined in JDK by Sun

```
public interface ActionListener {
public void actionPerformed(ActionEvent e);
}
```

```
public interface ItemListener {
public void itemStateChanged(ItemEvent e);
}
```

```
public interface ComponentListener {
public void componentHidden(ComponentEvent e);
 public void componentMoved(ComponentEvent e);
public void componentResized(ComponentEvent e);
public void componentShown(ComponentEvent e);
}
```

- By implementing an interface the class agrees to implement all the methods that are present in that interface. Implementing an interface is like *signing a contract*.
- Inside the method the class can do whatever it wants to do with that event
- Event Generator and Event Handler can be the same or different classes
- To handle events generated by Button. A class needs to implement ActionListener interface and thus needs to provide the definition of actionPerformed()method which is present in this interface.

```
public class Test implements ActionListener{
```

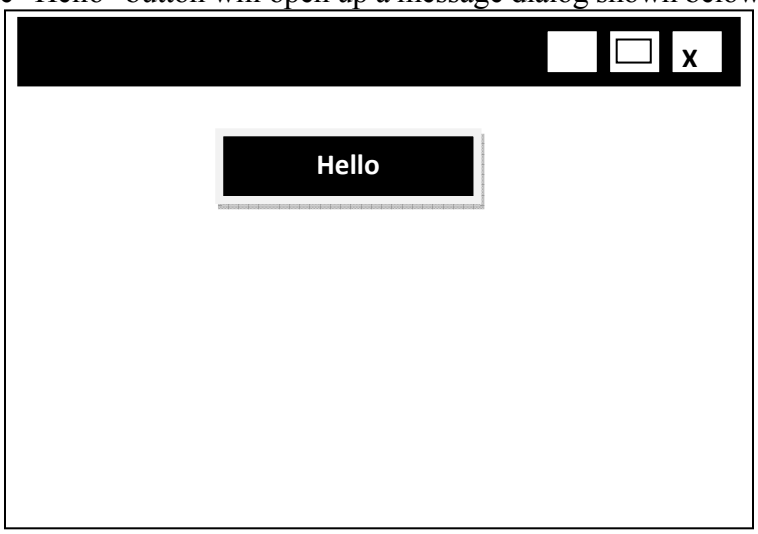
```
public void actionPerformed(ActionEvent ae) {
// do something
}
}
```

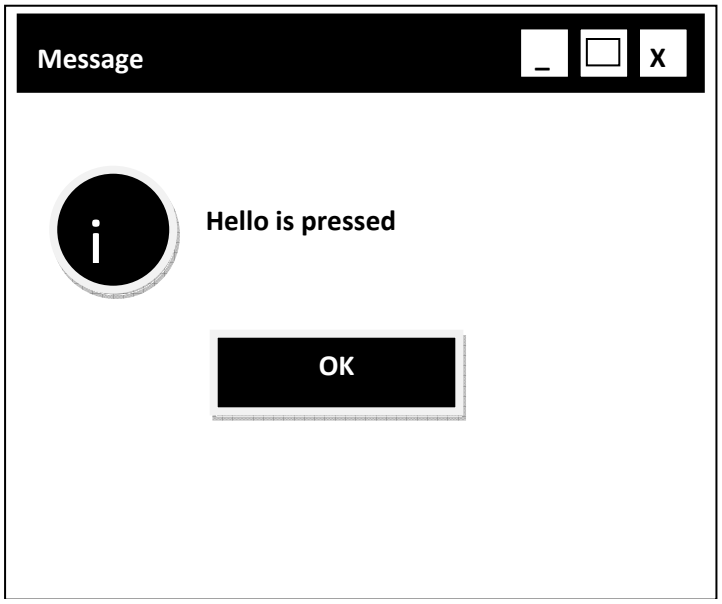
11.3.3 Step 3: Registering Handler with Generator

- The event generator is told about the object which can handle its events
- Event Generators have a method
  - addXXXListener(_reference to the object of Handler class_)
- For example, if b1 is JButtonthen
  - b1.addActionListener(this); // if listener and generator are same class

Event Handling Example

Clicking the "Hello" button will open up a message dialog shown below.





We will take the simplest approach of creating handler and generator in a single class. Button is our event generator and to handle that event our class needs to implement ActionListener Interface and to override its actionPerformedmethod and in last to do the registration.

```
1. import java.awt.*;
2. import javax.swing.*;
3. import java.awt.event.*;
/* Implementing the interface according to the type of the event, i.e. creating event handler (first part of step 2 of our process)
*/

4. public class ActionEventTest implements ActionListener{
5. JFrame frame;
6. JButton hello;
// setting layout components
7. public void initGUI ( ) {
8. frame = new JFrame();
9. Container cont =     frame.getContentPane();
10. cont.setLayout(new FlowLayout());
//Creating event generator step-1 of our process
11. hello = new JButton("Hello");
/* Registering event handler with event generator. Since event handler is in same object that contains button, we have used this to pass the reference.(step 3 of the process) */
12. hello.addActionListener(this);
13. cont.add(hello);
14. frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15. frame.setSize(150, 150);
16. frame.setVisible(true);
17. }
//constructor
18. public ActionEventTest ( ) {
19. initGUI();
20. }
/* Override actionPerformed method of ActionListener's interfacemethod of which will be called when event takes place (second part of step 2 of our process) */
21.       public void actionPerformed(ActionEvent event) {
22. JOptionPane.showMessageDialog(null,"Hello is pressed");
23. }
24.       public static void main(String args[]) {
25. ActionEventTest aeTest = new ActionEventTest();
26. }
27.} // end class
```

11.4 How Event Handling Participants Interact Behind the Scenes?

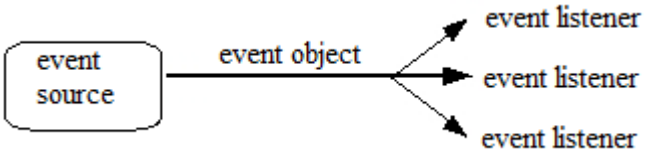
We have already seen that what a programmer needs to do handle events. Let's see what takes place behind the scenes, i.e How JVM handles event. Before doing that lets revisit different participants of Event Handling Process and briefly what they do.

11.4.1 Event Generator / Source

- Swing and awt components
- For example, JButton, JTextField, JFrame etc
- Generates an event object
- Registers listeners with itself

11.4.2 Event Object

- Encapsulate information about event that occurred and the source of that event
- For example, if you click a button, ActionEvent object is created



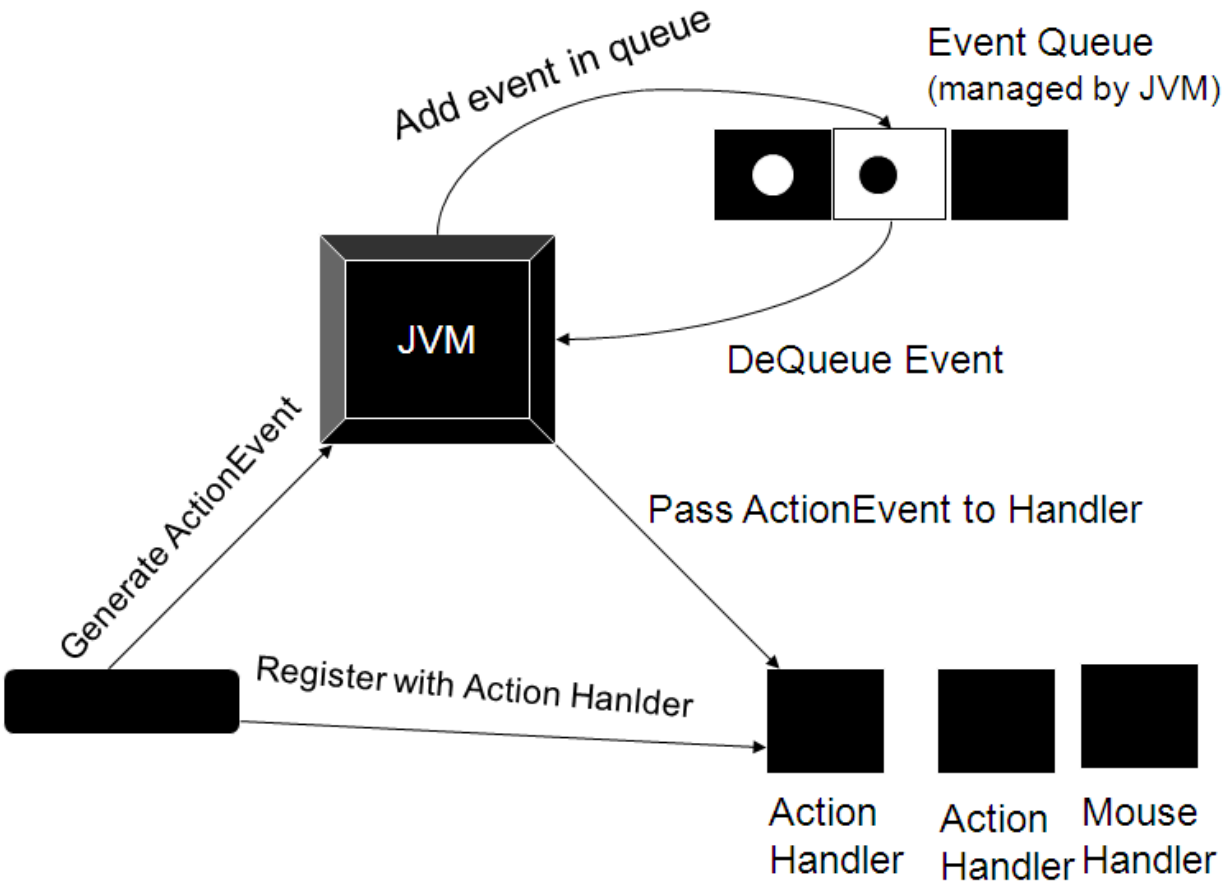
11.4.3 Event Listener/handler

- Receives event objects when notified, then responds
- Each event source can have multiple listeners registered on it
- Conversely, a single listener can register with multiple event sources

11.4.4 JVM

- Receives an event whenever one is generated
- Looks for the listener/handler of that event
- If exist, delegate it for processing
- If not, discard it (event).

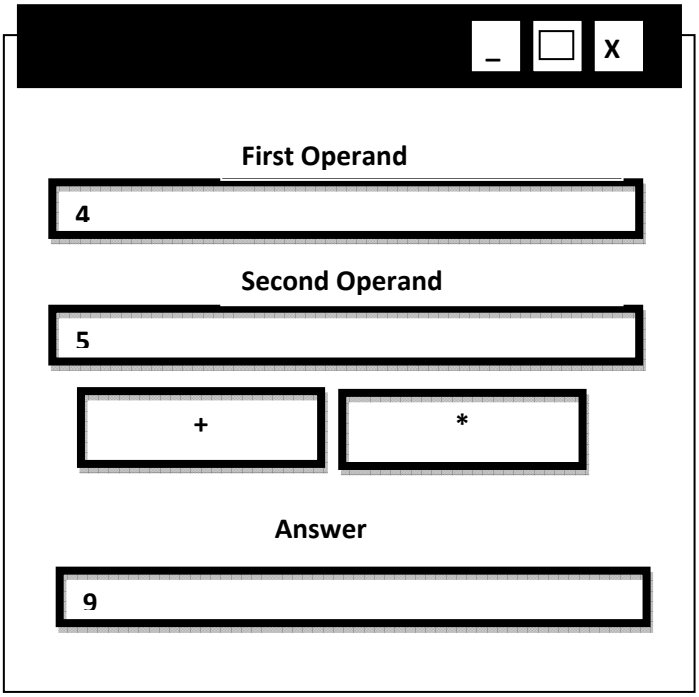
When button generates an ActionEvent it is sent to JVM which puts it in an event queue. After that when JVM find it appropriate it de-queue the event object and send it to all the listeners that are registered with that button. This is all what we shown in the pictorial form below:

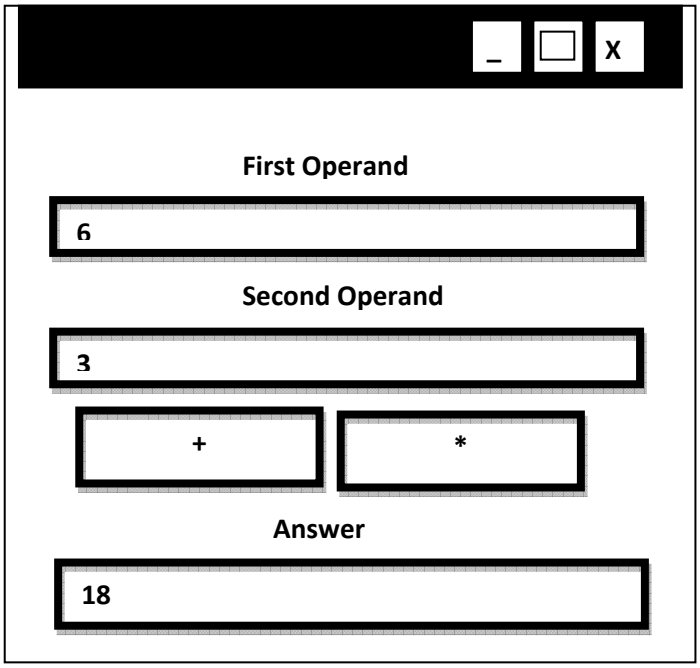


(Figure from JAVA A Lab Course)

Making Small Calculator

- User enters numbers in the provided fields
- On pressing "+" button, sum would be displayed in the answer field
- On pressing "*" button, product would be displayed in the answer field





Example Code: Making Small Calculator

```
1. import java.awt.*;
2. import javax.swing.*;
3. import java.awt.event.*;
4. public class SmallCalcApp implements ActionListener{
5. JFrame frame;
6. JLabel firstOperand, secondOperand, answer;
7. JTextField op1, op2, ans;
8. JButton plus, mul;
9. // setting layout
10. public void initGUI ( ) {
11. frame = new JFrame();
12. firstOperand         = new JLabel("First Operand");
13. secondOperand = new JLabel("Second Operand");
14. answer       = new JLabel("Answer");
15. op1 = new JTextField (15);
16. op2 = new JTextField (15);
17. ans = new JTextField (15);
18. plus = new JButton("+");
19. plus.setPreferredSize(new Dimension(70,25));
20. mul = new JButton("*");
21. mul.setPreferredSize(new Dimension(70,25));
22. Container cont =   frame.getContentPane();
23. cont.setLayout(new FlowLayout());
24. cont.add(firstOperand);
25. cont.add(op1);
```

```
26. cont.add(secondOperand);
27. cont.add(op2);
28. cont.add(plus);
29. cont.add(mul);
30. cont.add(answer);
31. cont.add(ans);
32. plus.addActionListener(this);
33. mul.addActionListener(this);
34. frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
35. frame.setSize(200, 220);
36. frame.setVisible(true);
37. }
38. //constructor
39. public SmallCalcApp ( ) {
40. initGUI();
41. }
42. public void actionPerformed(ActionEvent event) {
43. String oper, result;
44. int num1, num2, res;
/* All the information regarding an event is contained inside the event object. Here we are calling the
getSource() method on the event object to figure out the button that has generated that event.      */
45. if (event.getSource() == plus) {
46. oper = op1.getText();
47. num1 = Integer.parseInt(oper);
48. oper = op2.getText();
49. num2 = Integer.parseInt (oper);
50. res = num1+num2;
51. result = res+"";
52. ans.setText(result);
53. }
54. else if (event.getSource() == mul) {
55. oper = op1.getText();
56. num1 = Integer.parseInt(oper);
57. oper = op2.getText();
58. num2 = Integer.parseInt (oper);
59. res = num1*num2;
60. result = res+"";
61. ans.setText(result);
62. }
63.      }
64. public static void main(String args[]) {
65. SmallCalcApp scApp = new SmallCalcApp();
66. }
67. }// end class
```

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 12: More Examples of Handling Events

12.1 Handling Mouse Event

Mouse events can be trapped for any GUI component that inherits from Component class. For example, JPanel, JFrame & JButton etc.

To handle Mouse events, two types of listener interfaces are available.

- MouseMotionListener
- MouseListener

The class that wants to handle mouse event needs to implement the corresponding interface and needs to provide the definition of all the methods in that interface.

12.1.1 MouseMotionListener interface

- Used for processing mouse motion events
- Mouse motion event is generated when mouse is moved or dragged

A MouseMotionListener interface isdefined in JDK as follows:

```
public interface MouseMotionListener {
public void mouseDragged (MouseEvent me);
public void mouseMoved (MouseEvent me);
}
```

12.1.2 MouseListener interface

- Used for processing “interesting” mouse events like when mouse is:
    - Pressed
    - Released
    - Clicked (pressed & released without moving the cursor)
    - Enter (mouse cursor enters the bounds of component)
    - Exit (mouse cursor leaves the bounds of component)

MouseListenerinterfaces are defined in JDK as follows:

```
public interface MouseListener {
public void mousePressed (MouseEvent me);
```

```
public void mouseClicked (MouseEvent me);
public void mouseReleased (MouseEvent me);
public void mouseEntered (MouseEvent me);
public void mouseExited (MouseEvent me);
}
```

Example Code: Handling Mouse Events

Example to show Mouse Event Handling .Every time mouse is moved, the coordinates for a new place is shown in a label.

```
1. import java.awt.*;
2. import javax.swing.*;
3. import java.awt.event.*;
4. public class EventsEx implements MouseMotionListener{
5. JFrame frame;
6. JLabel coordinates;
7. // setting layout
8. public void initGUI ( ) {
9. // creating event generator
10. frame = new JFrame();
11. Container cont =frame.getContentPane();
12. cont.setLayout(new BorderLayout( ) );
13. coordinates = new JLabel ();
14. cont.add(coordinates, BorderLayout.NORTH);
15. // registring mouse event handler with generator
16. frame.addMouseMotionListener(this);
17. frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18. frame.setSize(350, 350);
19. frame.setVisible(true);
20. } // end initGUI method
21. //default constructor
22.       public EventsEx ( ) {
23. initGUI();
24. }
// MouseMotionListener event hadler handling dragging
25.       public void mouseDragged (MouseEvent me) {
26. int x = me.getX();
27. int y = me.getY();
28. coordinates.setText("Dragged at [" + x + "," + y + "]");
29. }
// MouseMotionListener event handler handling motion
30.       public void mouseMoved (MouseEvent me) {
```

```
32. int y = me.getY();
33. coordinates.setText("Moved at [" + x + "," + y + "]");
34. }
35.       public static void main(String args[]) {
36. EventsEx ex = new EventsEx();
37. }
38. } // end class
```

Another Example: Handling Window Events

Task

We want to handle Window Exit event only

Why?

- When window is closed, control should return back to command prompt.
- But we have already achieved this functionality through following line of code

  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

- But, what if we want to display some message (*Good Bye*) before exiting?



When user close the window, Message would be displayed



After pressing Ok button program will exit

How?

- To handle window events, we need to implement "WindowListner" interface.
- WindowListner" interface contains *7* methods We require only one i.e. windowClosing

- But, We have to provide definitions of all methods to make our class a concrete class
- WindowListener interface is defined in the JDK as follows

public interface **WindowListener** {
public void windowActivated(WindowEvent we); public void windowClosed(WindowEvent we); **public void windowClosing(WindowEvent we);** public void windowDeactivated(WindowEvent we); public void windowDeiconified(WindowEvent we); public void windowIconified(WindowEvent we); public void windowOpened(WindowEvent we);
}

- public void windowClosing(WindowEvent we)is our required method

Example Code: WindowExitHandler

This example code is modification of the last code example i.e. EventsEx.java

```
1. import java.awt.*;
2. import javax.swing.*;
3. import java.awt.event.*;
4. public class EventsEx implements MouseMotionListener , WindowListener {
5. JFrame frame;
6. JLabel coordinates;
// setting layout
7. public void initGUI ( ) {
// creating event generator
8. frame = new JFrame();
9. Container cont =     frame.getContentPane();
10. cont.setLayout(new BorderLayout( ) );
11. coordinates = new JLabel ();
12. cont.add(coordinates, BorderLayout.NORTH);
// registring mouse event handler with generator
13. frame.addMouseMotionListener(this);
// registering window handler with generator
14. frame.addWindowListener(this);
15. frame.setSize(350, 350);
16. frame.setVisible(true);
17. } // end initGUI method
//default constructor
18. public EventsEx ( ) {
```

```
19. initGUI();
20. }

// MouseMotionListener event hadler handling dragging
21. public void mouseDragged (MouseEvent me) {
22. int x = me.getX();
23. int y = me.getY();
24. coordinates.setText("Dragged at [" + x + "," + y + "]");
25. }
// MouseMotionListener event handler handling motion
26. public void mouseMoved (MouseEvent me) {
27. int x = me.getX();
28. int y = me.getY();
29.
30. coordinates.setText("Moved at [" + x + "," + y + "]");
31. }
// window listener event handler
32. public void windowActivated (WindowEvent we) {      }
33. public void windowClosed (WindowEvent we) {         }
34. public void windowClosing (WindowEvent we) {
35. JOptionPane.showMessageDialog(null, “Good Bye”);
36. System.exit(0);
37. }
38. public void windowDeactivated (WindowEvent we) {   }
39. public void windowDeiconified (WindowEvent we) {   }
40. public void windowIconified (WindowEvent we) {      }
41. public void windowOpened (WindowEvent we) {         }
42. public static void main(String args[]) {
43. EventsEx ex = new EventsEx();
44. }
45. } // end class
```

Note: Coding exercises in working condition for this lecture are also available on **“Downloads”** section of LMS.

Lecture 13: Adapter Classes

Problem in Last Code Example

Problem

- We were interested in *windowClosing()* method only
- But have to provide definitions of all the methods, Why?
- Because a class implementing an interface has to provide definitions of all methods present in that interface.

Solution

- To avoid giving implementations of all methods of an interface when we are not using these methods we use Event *Adapter* classes

13.1 Adapter Classes

- For listener interfaces containing more than one event handling methods, jdk defines adapter classes. Examples are
  - For WindowListener WindowAdapter
  - For MouseMotionListener MouseMotionAdapter
  - and many more
- Adapter classes provide definitions for all the methods (empty bodies) of their corresponding Listener interface
- It means that WindowAdapter class implements WindowListener interface and provide the definition of all methods inside that Listener interface
- Consider the following example of MouseMotionAdapter and its corresponding MouseMotionListener interface

```
public interface MouseMotionListener {
public void mouseDragged (MouseEvent me);
public void mouseMoved (MouseEvent me); }
public class MouseMotionAdapter implements MouseMotionListener{
public void mouseDragged (MouseEvent me) {       }
public void mouseMoved (MouseEvent me) {         }
}
```

13.2 Available Adapter classes

Listener	Adapter Class (If Any)	Registration Method
ActionListener		addActionListener
AdjustmentListener		addAdjustmentListener
ComponentListener	ComponentAdapter	addComponentListener
ContainerListener	ContainerAdapter	addContainerListener
FocusListener	FocusAdapter	addFocusListener
ItemListener		addItemListener
KeyListener	KeyAdapter	addKeyListener
MouseListener	MouseAdapter	addMouseListener
MouseMotionListener	MouseMotionAdapter	addMouseMotionListener
TextListener		addTextListener
WindowListener	WindowAdapter	addWindowListener

13.2.1 How to use Adapter Classes

- previously handler class need to implement interface

  public class EventsEx **implements MouseMotionListener**{...}
- Therefore it has to provide definitions of all the methods inside that interface now our handler class will inherit from adapter class

  public class EventsEx **extends MouseMotionAdapter**{...}
- Due to inheritance, all the methods of the adapter class will be available inside our handler class since adapter classes has already provided definitions with empty bodies.
- we do not have to provide implementations of all the methods again
- We only need to override our method of interest.

Example Code 13.1: Handling Window Events using Adapter Classes

Here we are modifying the window event code in the last example to show the use of WindowAdapter instead of WindowListener. Code related to MouseMotionListener is deleted to avoid cluttering of code.

```
1. import java.awt.*;
```

```
2. import javax.swing.*;
3. import java.awt.event.*;
4. public class EventsEx extends WindowAdapter {
5. JFrame frame;
6. JLabel coordinates;
// setting layout
7. public void initGUI ( ) {
// creating event generator
8. frame = new JFrame();
9. Container cont =     frame.getContentPane();
10. cont.setLayout(new BorderLayout( ) );
11. coordinates = new JLabel ();
12. cont.add(coordinates, BorderLayout.NORTH);
// registering window handler with generator
13. frame.addWindowListener(this);
14. frame.setSize(350, 350);
15. frame.setVisible(true);
16. } // end initGUI method
//default constructor
17. public EventsEx ( ) {
18. initGUI();
19. }
//          As you can see that we have only implemented
//          our required method
20. public void windowClosing (WindowEvent we) {
21. JOptionPane.showMessageDialog(null, “Good Bye”);
22. System.exit(0);
23. }

24. public static void main(String args[]) {
25. EventsEx ex = new EventsEx();
26. }
27. } // end class
```

Problem in Last Code Example

- We have inherited from WindowAdapter
- What if we want to use MouseMotionAdpater as well? Or what if our class already inherited from some other class?

Problem

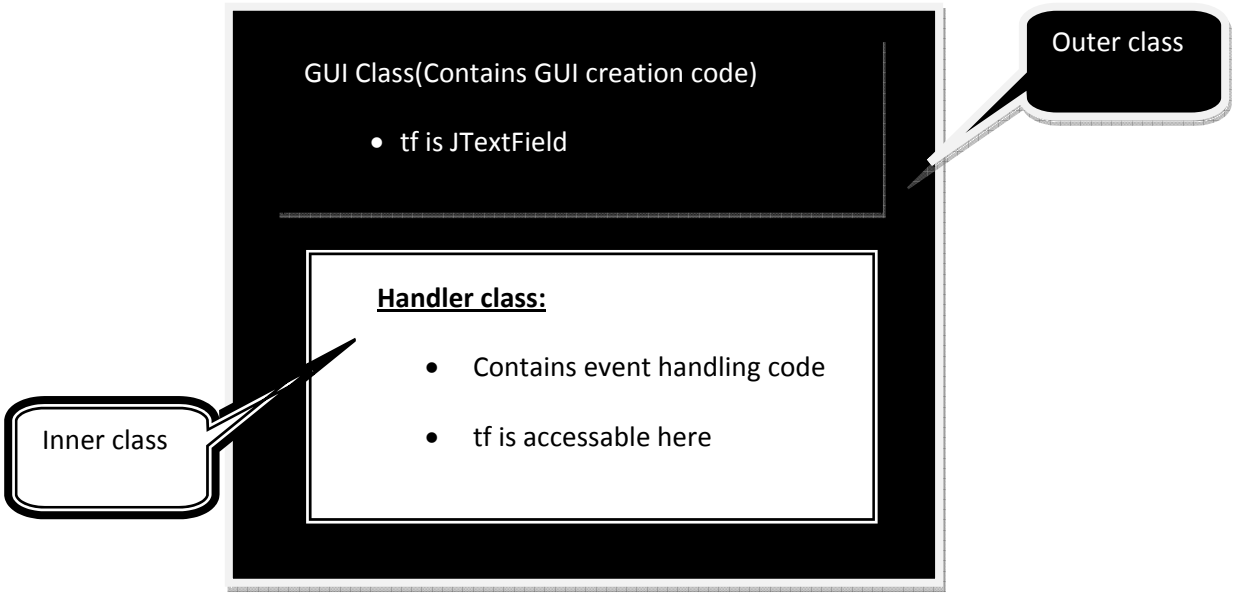
- Java allows single inheritance

Solution

- Use *Inner* classes

13.3 Inner Classes

- A class defined inside another class
- Inner class can access the instance variables and members of outer class
- It can have constructors, instance variables and methods, just like a regular class
- Generally used as a private utility class which does not need to be seen by others classes



Example Code13.2: Handling Window Event with Inner Class

Here we are modifying the window event code in the last example to show the use of WindowAdapter as an inner class.

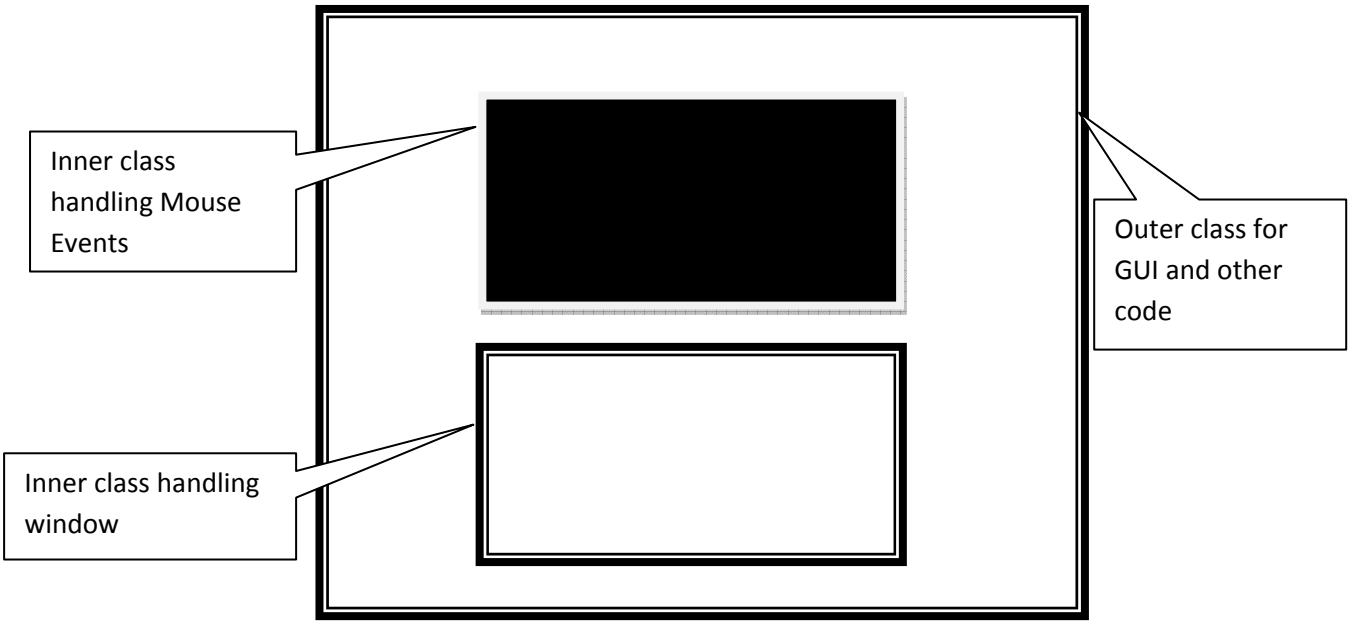
```
1.      import java.awt.*;
2.      import javax.swing.*;
3.      import java.awt.event.*;
4.      public class EventEx {

5. JFrame frame;
6. JLabel coordinates;
7. // setting layout
8. public void initGUI ( ) {
9. frame = new JFrame();
10. Container cont =frame.getContentPane();
11. cont.setLayout(new BorderLayout( ));
12. coordinates = new JLabel ();
```

```
13. cont.add(coordinates, BorderLayout.NORTH);
/* Creating an object of the class which is handling our
window events and registering it with generator        */
14. WindowHandler handler = new WindowHandler ();
15. frame.addWindowListener(handler);
16. frame.setSize(350, 350);
17. frame.setVisible(true);
18. } // end initGUI
//default constructor
19. public EventEx ( ) {
20. initGUI();
21. }
/* Inner class implementation of window adapter. Outer
class is free to inherit from any other class. */
22. private class WindowHandler extends WindowAdapter {
// Event Handler for WindowListener
23. public void windowClosing (WindowEvent we) {
24. JOptionPane.showMessageDialog(null, “Good Bye”);
25. System.exit(0);
26. }
27. } // end of WindowHandler class
28. public static void main(String args[]) {
29. EventEx e = new EventEx();
30. }
31. } // end class
```

Example Code 13.3: Handling Window and Mouse Events with Inner Class

Here we are modifying the window event code of the last example to handle window and mouse events using inner classes. The diagram given below summarizes the approach.

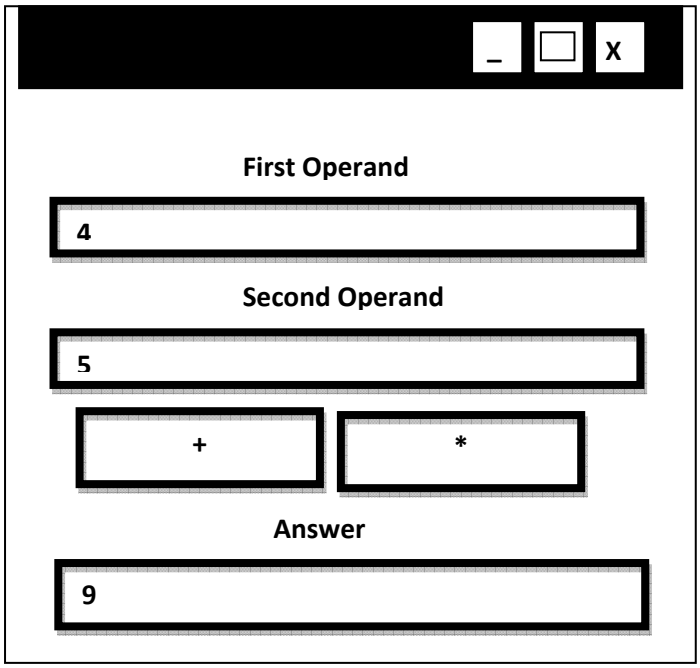


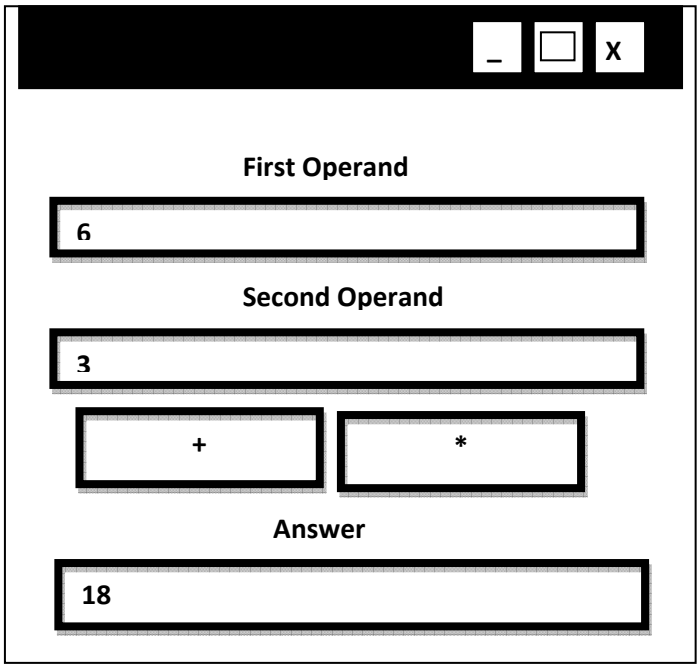

```
1.      import java.awt.*;
2.      import javax.swing.*;
3.      import java.awt.event.*;
4.      public class EventEx {
5. JFrame frame;
6. JLabel coordinates;
7. // setting layout
8. public void initGUI ( )
{
9. frame = new JFrame();
10. Container cont =frame.getContentPane();
11. cont.setLayout(new BorderLayout( ) );
12. coordinates = new JLabel ();
13. cont.add(coordinates, BorderLayout.NORTH);
/* Creating an object of the class which is handling our window events and registering it with
generator */
14. WindowHandler whandler = new WindowHandler ();
15. frame.addWindowListener(whandler);
/* Creating an object of the class which is handling our MouseMotion events & registering it with
generator */
16. MouseHandler mhandler = new MouseHandler ();
17. frame.addMouseMotionListener(mhandler);
18. frame.setSize(350, 350);
```

```
20. }
//default constructor
21. public EventEx ( ) {
22. initGUI();
23. }
/* Inner class implementation of WindowAdapter. Outer class
is free to inherit from any other class. */
24. private class WindowHandler extends WindowAdapter {
// Event Handler for WindowListener
25. public void windowClosing (WindowEvent we) {
26. JOptionPane.showMessageDialog(null, “Good Bye”);
27. System.exit(0);
28. }
29. } // end of WindowHandler
//Inner class implementation of MouseMotionAdapter
30. private class MouseHandler extends MouseMotionAdapter {
// Event Handler for mouse motion events
31. public void mouseMoved (MouseEvent me) {
32. int x = me.getX();
33. int y = me.getY();
34. coordinates.setText(“Moved at [" + x + "," + y + "]” );
35. }
36. } // end of MouseHandler
37. public static void main(String args[]) {
38. EventEx e = new EventEx();
39. }
40. } // end clas
```

Example Code: Making Small Calculator using Inner classes

- User enters numbers in the provided fields
- On pressing “+” button, sum would be displayed in the answer field
- On pressing “*” button, product would be displayed in the answer field






```
1. import java.awt.*;
2. import javax.swing.*;
3. import java.awt.event.*;
4. public class SmallCalcApp{
5. JFrame frame;
6. JLabel firstOperand, secondOperand, answer;
7. JTextField op1, op2, ans;
8. JButton plus, mul;
9. // setting layout
10. public void initGUI ( ) {
11. frame = new JFrame();
12. firstOperand         = new JLabel("First Operand");
13. secondOperand = new JLabel("Second Operand");
14. answer       = new JLabel("Answer");
15. op1 = new JTextField (15);
16. op2 = new JTextField (15);
17. ans = new JTextField (15);
18. plus = new JButton("+");
19. plus.setPreferredSize(new Dimension(70,25));
20. mul = new JButton("*");
21. mul.setPreferredSize(new Dimension(70,25));
22. Container cont =    frame.getContentPane();
23. cont.setLayout(new FlowLayout());
```

```
24. cont.add(firstOperand);
25. cont.add(op1);
26. cont.add(secondOperand);
27. cont.add(op2);
28. cont.add(plus);
29. cont.add(mul);
30. cont.add(answer);
31. cont.add(ans);
/* Creating an object of the class which is handling
button events & registering it with generators */
32. ButtonHandler bHandler = new ButtonHandler();
33. plus.addActionListener(bHandler);
34. mul.addActionListener(bHandler);
35. frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
36. frame.setSize(200, 220);
37. frame.setVisible(true);
38. }
39. //constructor
40. public SmallCalcApp ( ) {
41. initGUI();
42. }
//Inner class implementation of ActionListener
43. private class ButtonHandler implements ActionListener{
44. public void actionPerformed(ActionEvent event) {
45. String oper, result;
46. int num1, num2, res;
47. if (event.getSource() == plus) {
48. oper = op1.getText();
49. num1 = Integer.parseInt(oper);
50. oper = op2.getText();
51. num2 = Integer.parseInt (oper);
52. res = num1+num2;
53. result = res+"";
54. ans.setText(result);
55 }
56. else if (event.getSource() == mul) {
57. oper = op1.getText();
58. num1 = Integer.parseInt(oper);
59. oper = op2.getText();
60. num2 = Integer.parseInt (oper);
61. res = num1*num2;
62. result = res+"";
63. ans.setText(result);
64 }
65. } // end actionPerformed method
66. } // end inner class ButtonHandler
```

```
67. public static void main(String args[]) {
68. SmallCalcApp scApp = new SmallCalcApp();
69. }
70. }// end class
```

13.4 Anonymous Inner Classes

- has no name
- same as inner class in capabilities
- much shorter
- Difficult to understand

13.5 Named vs. Anonymous Objects

13.5.1 Named

- String s = "hello";
  System.out.println(s);
- "hello" has a named reference s.

13.5.2 Anonymous

- System.out.println("hello");

We generally use anonymous object when there is just a onetime use of a particular object but in case of a repeated use we generally used named objects and use that named reference to use that objects again and again.

Example Code 13.4 Handling Window Event with Anonymous Inner Class

Here we are modifying the window event code of 13.3 to show the use of anonymous inner class.

```
28.import java.awt.*;
29.import javax.swing.*;
30.import java.awt.event.*;
31.public class EventsEx extends WindowAdapter {
32. JFrame frame;
33. JLabel coordinates;
// setting layout
34.public void initGUI ( ) {
// creating event generator
35. frame = new JFrame();
36. Container cont =frame.getContentPane();
37. cont.setLayout(new BorderLayout( ) );
38. coordinates = new JLabel ();
39. cont.add(coordinates, BorderLayout.NORTH);
// registering event handler (anonymous inner class)
// with generator by using
40. frame.addWindowListener (
41. new WindowAdapter ( ) {
```

```
42. public void windowClosing (WindowEvent we) {
43. JOptionPane.showMessageDialog(null, “Good Bye”);
44. System.exit(0);
45. } // end window closing
46. } // end WindowAdapter
47. ); // end of addWindowListener
48. frame.setSize(350, 350);
49. frame.setVisible(true);
50. } // end initGUI method
//default constructor
51.       public EventsEx ( ) {
52. initGUI();
53. }
54.       public static void main(String args[]) {
55. EventsEx ex = new EventsEx();
56. }
57. } // end class
```

13.6 Summary of Approaches for Handling Events

- By implementing Interfaces
- By extending from Adapter classes

To implement the above two techniques we can use

- **Same class**
  - putting event handler & generator in one class
- **Separate class**
  - Outer class
    - Putting event handlers & generator in two different classes
  - Inner classes
  - Anonymous Inner classes

13.7 References

Java A Lab Course by Umair Javed

Note: Coding exercises in working condition for this lecture are also available on **“Downloads”** section of LMS.

Lecture 14: Java Database Connectivity

14.1 Introduction

Java Database Connectivity (JDBC) provides a standard library for accessing databases. The JDBC API contains number of interfaces and classes that are extensively helpful while communicating with a database.

14.2 The java.sql package

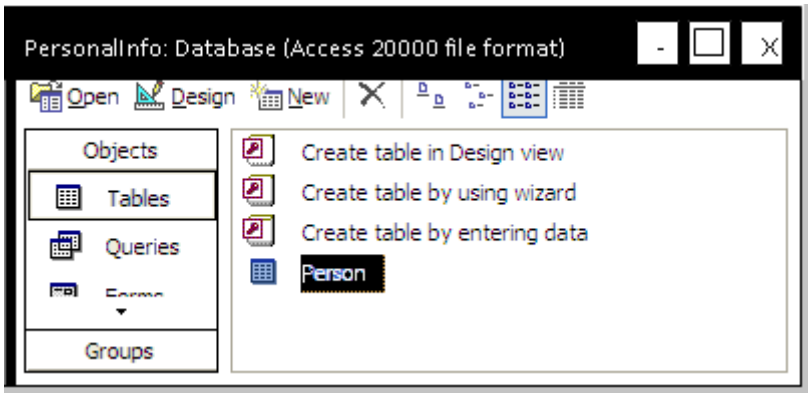
The java.sql package contains basic & most of the interfaces and classes. You automatically get this package when you download the J2SE™. You have to import this package whenever you want to interact with a relational database.

14.3 Connecting With Microsoft Access

In this handout, we will learn how to connect & communicate with Microsoft Access Database. We chooses Access because most of you are familiar with it and if not than it is very easy to learn.

14.3.1 Create Database

In start create a database "PersonInfo" using Microsoft Access. Create one table named "Person". The schema of the table is shown in the picture.



Person: Table

	Field Name	Data Type
🔑	id	AutoNumber
	name	Text
	address	Text
	phoneNum	Text

Add the following records into Person table as shown below:

Person: Table

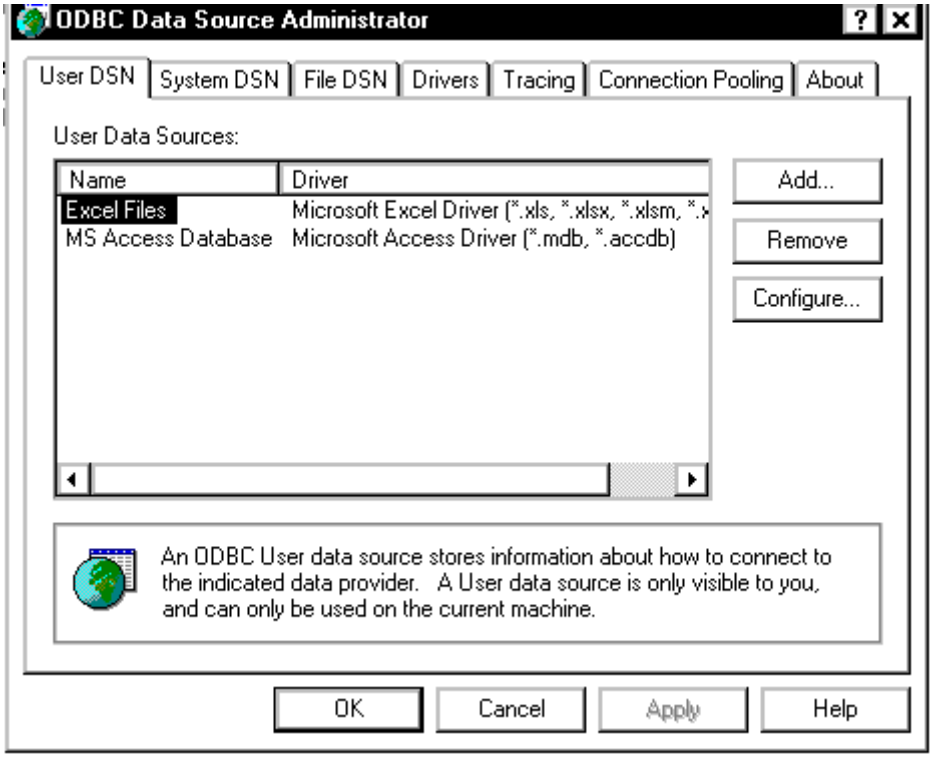
	id	name	address	phoneNum
▶	1	ali	model town	9203256
	2	usman	gulberg	8219065
	3	raza	defence	5173946
*	0			

Record: 1 of 3

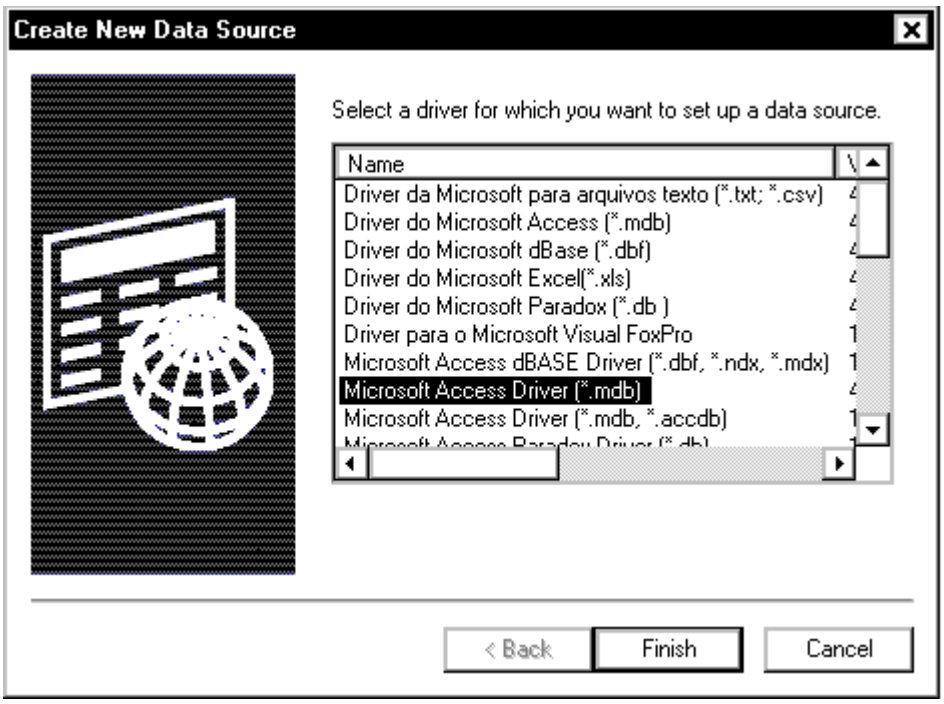
Save the data base in some folder. (Your database will be saved as an .mdb file)

14.3.2 Setup System DSN

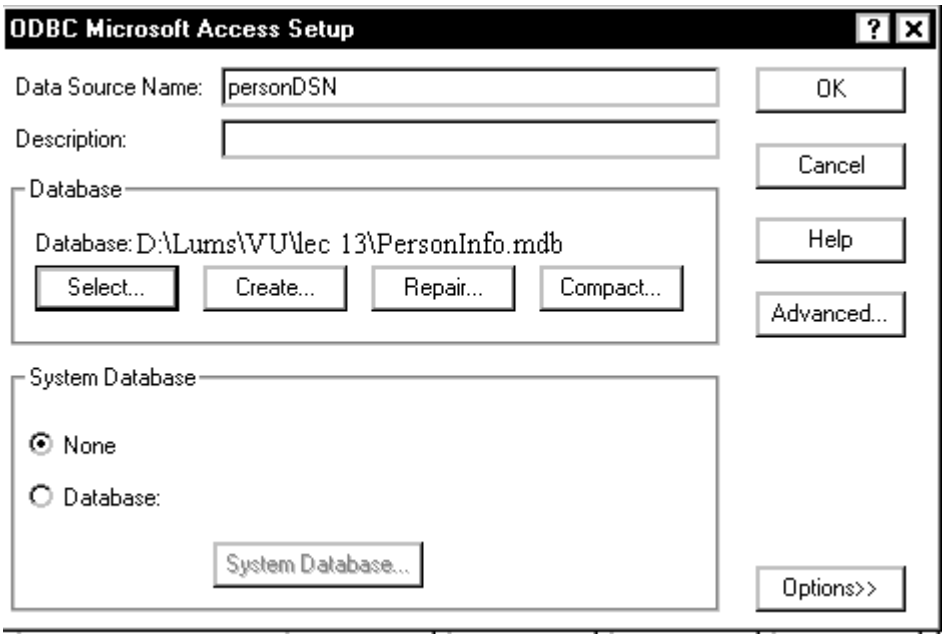
- After creating database, you have to setup a system Data Source Name (DSN). DSN is a name through which your system recognizes the underlying data source.
- Select Start -> Settings -> Control Panel -> Administrative Tools -> Data Sources (ODBC).
- The ODBC Data Source Administrator window would be opened as shown below: Select System DSN tab. (If you are unable to use System DSN tab due to security restrictions on your machine, you can use the User DSN tab)



- Press Add… button and choose Microsoft Access Driver (*.mdb) from Create New Data Source window and press Finish button as shown in diagram:



- After that, ODBC Microsoft Access Setup window would be opened as shown in following diagram:



- Enter the Data Source Name *personDSN* and select the database by pressing Select button. The browsing window would be opened, select the desired folder that contains the database (The database .mdb file you have created in the first step) Press Ok button.

14.4 Basic Steps in Using JDBC

There are eight (8) basic steps that must be followed in order to successfully communicate with a database. Let's take a detail overview of all these one by one.

14.4.1 Import Required Package

- Import the package java.sql.* that contains useful classes and interfaces to access & work with database.

14.4.2 Load Driver

```
import java.sql.*;
```

- Need to load suitable driver for underlying database.

- Different drivers & types for different databases are available.
- For MS Access, load following driver available with j2se.
- Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
- For Oracle, load the following driver. You have to download it explicitly.
- Class.forName("oracle.jdbc.driver.OracleDriver");

14.4.3 Define Connection URL

- To get a connection, we need to specify the URL of a database (Actually we need to specify the address of the database which is in the form of URL)
- As we are using Microsoft Access database and we have loaded a JDBC-ODBC driver. Using JDBC-ODBC driver requires a DSN which we have created earlier and named it personDSN. So the URL of the database will be

String conURL = "jdbc:odbc:personDSN";

14.4.4 Establish Connection With DataBase

- Use DriverManager to get the connection object.
- The URL of the database is passed to the getConnection method.
- **Connection con = DriverManager.getConnection(conURL);**
- If DataBase requires username & password, you can use the overloaded version of getConnection method as shown below:

```
String usr = "umair";
String pwd = "vu";
Connection con = null;
con = DriverManager.getConnection(conURL, usr, pwd);
```

14.4.5 Create Statement

- A Statement object is obtained from a Connection object.

Statement stmt = con.createStatement( );

- Once you have a statement, you can use it for various kinds of SQL queries.

14.4.6 Execute a Query

- The next step is to pass the SQL statements & to execute them.
- Two methods are generally used for executing SQL queries. These are:

- **executeQuery(sql) method**
  - Used for SQL SELECT queries.
  - Returns the ResultSET object that contains the results of the query and can be used to access the query results.

```
String sql = "SELECT * from sometable";
ResultSet rs = stmt.executeQuery(sql);
```

- **executeUpdate(sql) method**
  - This method is used for executing an update statement like INSERT, UPDATE or DELETE
  - Returns an Integer value representing the number of rows updated

```
String sql = "INSERT INTO tablename " +
                "(columnNames) Values (values)" ;
int count = stmt.executeUpdate(sql);
```

14.4.7 Process Results of the Query

- The ResultSet provides various getXXX methods that takes a column index or name and returns the data
- The ResultSet maintains the data in the form tables (rows & columns)
- First row has index 1, not 0.
- The next method of ResultSet returns true or false depending upon whether the next row is available (exist) or not and moves the cursor
- Always remember to call next()method at-least once
- To retrieve the data of the column of the current row you need to use the various getters provided by the ResultSet.
- For example, the following code snippet will iterate over the whole ResultSetand illustrates the usage of getters methods

```
while ( rs.next() ){
String name = rs.getString("columnName");   //by using column name
String name = rs.getString(1); // or by using column index }
```

14.4.8 Close the Connection

- An opening connection is expensive, postpone this step if additional database operations are expected

```
con.close();
```

Example Code 14.1: Retrieving Data from ResultSet

The JdbcEx.java demonstrates the usage of all above explained steps. In this code example, we connect with the *PersonInfo* database, the one we have created earlier, and then execute the simple SQL SELECT query on *Person* table, and then process the query results.

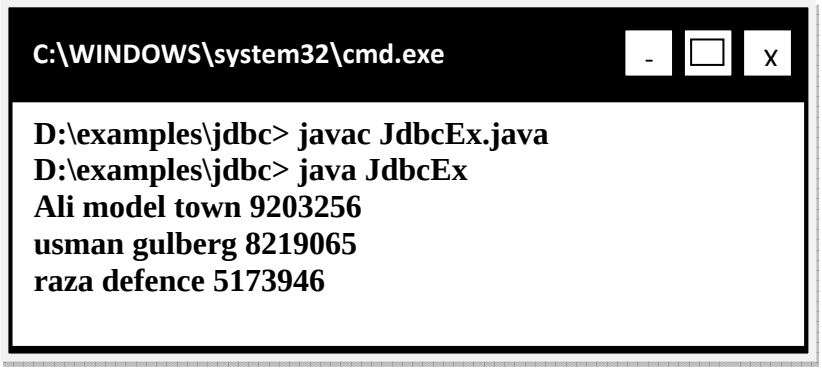
```
// File JdbcEx.java
//Step 1: Import package
import java.sql.*;
public class JdbcEx {
public static void main (String args[ ]) {
try {
//Step 2: load driver
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
//Step 3: define the connection URL
String url = "jdbc:odbc:personDSN";
//Step 4: establish the connection
Connection con = DriverManager.getConnection(url);
//Step 5: create Statement
Statement st = con.createStatement();
//Step 6: preapare & execute the query
String sql = "SELECT * FROM Person";
ResultSet rs = st.executeQuery(sql);
//Step 7: process the results
while(rs.next()){
// The row name is "name" in database "PersonInfo,
// hence specified in the getString() method.
String name = rs.getString("name");
String add      = rs.getString("address");
String pNum = rs.getString("phoneNum");
System.out.println(name + " " + add + " " + pNum);
}
//Step 8: close the connection
con.close();
}catch(Exception sqlEx){
System.out.println(sqlEx);
}
} // end main
} // end class
```

The important thing you must notice that we have put all code inside tryblock and then handle (in the above example, only printing the name of the exception raised) exception inside catch block.

Why? Because we are dealing with an external resource (database). If you can recall all IO related operations involving external resources in java throw exceptions. These exceptions are checked exceptions and we must need to handle these exceptions.

Compile & Execute

Since the *Person* table contains only three records, so the following output would be produced on executing the above program.



14.5 References:

- Java - A Lab Course by Umair Javed
- Java tutorial by Sun: http://java.sun.com/docs/books/turorial
- Beginning Java2 by Ivor Hortan

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 15: More On JDBC

In the previous handout, we have discussed how to execute SQL statements. In this handout, we'll learn how to execute DML (insert, update, delete) statements as well some useful methods provided by the JDBC API.

Before jumping on to example, let's take a brief overview of executeUpdate() method that is used for executing DML statements.

15.1 Useful Statement Methods:

15.1.1 executeUpdate( )

- Used to execute for INSERT, UPDATE, or DELETE SQL statements.
- This method returns the number of rows that were affected in the database.
- Also supports DDL (Data Definition Language) statements CREATE TABLE, DROP TABLE, and ALERT TABLE etc.
- For example,

```
int num = stmt.executeUpdate("DELETE from Person WHERE id
= 2" );
```

Example Code 15.1 : Executing SQL DML Statements

This program will take two command line arguments that are used to update records in the database. executeUpdate( ) method will be used to achieve the purpose stated above.

```
// File JdbcDmlEx.java
//step 1:
import package import java.sql.*;
public class JdbcDmlEx {
public static void main (String args[ ]) {
try {
//Step 2: load driver
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

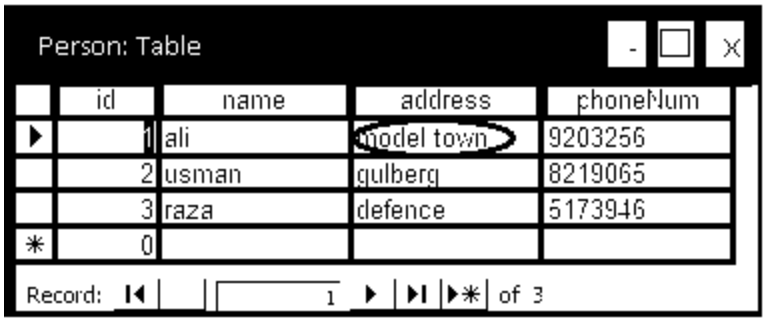
//Step 3: define the connection URL
```

```
String url = "jdbc:odbc:personDSN";
//Step 4: establish the connection
Connection con = DriverManager.getConnection(url);
//Step 5: create Statement
Statement st = con.createStatement();
// assigning first command line argument value
String addVar  = args[0];
// assigning second command line argument value

String nameVar = args[1];
// preparing query - nameVar & addVar strings are embedded
// into query within '" + string + "'
String sql =     "UPDATE Person SET address = '"+addVar+"'" +" WHERE
name = '"+nameVar+"' ";
// executing query
int num = st.executeUpdate(sql);
// Step 7: process the results of the query
// printing number of records affected
System.out.println(num + " records updated");
//Step 8: close the connection
con.close();
}catch(Exception sqlEx){
System.out.println(sqlEx);
} // end main
} // end class
```

Compile & Execute

The *Person* table is shown in the following diagram before execution of the program. We want to update first row i.e. address of the person ali.



The next diagram shows how we have executed our program. We passed it two arguments. The first one is the address (defence) and later one is the name (ali) of the person against whom we want to update the address value.

```
C:\WINDOWS\system32\cmd.exe
D:\examples\jdbc> javac JdbcEx.java
D:\examples\jdbc> java JdbcEx defence ali
1 record updated
```

The Person table is shown in the following diagram after the execution of the program. Notice that address of the ali is now changed to defence.

Person: Table

	id	name	address	phoneNum
▶	1	ali	defence	9203256
	2	usman	gulberg	8219065
	3	raza	defence	5173946
*	0			

Record: 1 of 3

Note

When we execute DML statements (insert, update, delete) we have to commit it in the database explicitly to make the changes permanent or otherwise we can rollback the previously executed statements.

But in the above code, you have never seen such a statement. This is due to the fact that java will implicitly commit the changes. However, we can change this java behavior to manual commit. We will cover these in some later handout.

15.1.2 getMaxRows / setMaxRows(int)

- Used for determines the number of rows a ResultSetmay contain
- By default, the number of rows are unlimited (return value is 0), or by using setMaxRows(int), the number of rows can be specified.

15.1.3 getQueryTimeOut / setQueryTimeOut (int)

- Retrieves the number of seconds the driver will wait for a Statement object to execute.
- The current query time out limit in seconds, zero means there is no limit

- If the limit is exceeded, a SQLException is thrown

15.2 Different Types of Statements

- As we have discussed in the previous handout that through Statement objects, SQL queries are sent to the databases.
- Three types of Statementobjects are available. These are:

15.2.1Statement

- The Statementobjects are used for executing simple SQL statements.

- We have already seen its usage in the code examples.

15.2.2PreparedStatement

- The PreparedStatement are used for executing precompiled SQL statements and passing in different parameters to it.
- We will talk about it in detail shortly.

15.2.3CallableStatement

- These are used for executing stored procedures.
- We are not covering this topic; See the Java tutorial on it if you are interested in learning it.

15.2.4 Prepared Statements

- What if we want to execute same query multiple times by only changing parameters.
- PreparedStatement object differs from Statement object as that it is used to create a statement in standard form that is sent to database for compilation, before actually being used.
- Each time you use it, you simply replace some of the marked parameters (?) using some setter methods.
- We can create PreparedStatement object by using prepareStatement method of the connection class. The SQL query is passed to this method as an argument as shown below.

```
PreparedStatement pStmt = con.prepareStatement (
                                        "UPDATE tableName SET columnName =
                                        ? " + "WHERE columnName = ? " );
```

- Notices that we used marked parameters (?) in query. We will replace them later on

by using various setter methods.

- If we want to replace *first* **?** with String value, we use setString method and to replace *second* **?** with int value, we use setInt method. This is shown in the following code snippet:

```
pStmt.setString (1 , stringValue);
pStmt.setInt (2 , intValue)
```

Note: The first market parameter has index 1.

- Next, we can call executeUpdate (for INSERT, UPDATE or DELETE queries) or executeQuery (for simple SELECT query) method**.**

```
pStmt.executeUpdate();
```

Modify Example Code: Executing SQL DML using Prepared Statements

This example code is modification to the last example code (JdbcDmlEx.java).The modifications are highlighted as bold face.

```
// File JdbcDmlEx.java
//step 1: import package
import java.sql.*;
public class JdbcDmlEx {
public static void main (String args[ ]) {
try {

//Step 2: load driver
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

//Step 3: define the connection URL
String url = "jdbc:odbc:personDSN";

//Step 4: establish the connection
Connection con = DriverManager.getConnection(url, "", "");

// make query and place ? where values are to
//be inserted later
String sql =     "UPDATE Person SET address = ? " +
" WHERE name = ? ";
// creating statement using Connection object and passing
// sql statement as parameter
PreparedStatement pStmt = con.prepareStatement(sql);
// assigning first command line argument value
String addVar = args[0];
```

```
// assigning second command line argument value
String nameVar = args[1];

// setting first marked parameter (?) by using setString()
// method to address.
pStmt.setString(1 , addVar);

// setting second marked parameter(?) by using setString()
// method to name
pStmt.setString(2 , nameVar);

// suppose address is “defence” & name is “ali”
// by setting both marked parameters, the query will look
// like:
//        sql = “UPDATE Person SET address = “defence”
//        WHERE name = “ali” ”


// executing update statemnt
int num = pStmt.executeUpdate();

// Step 7: process the results of the query
// printing number of records affected
System.out.println(num + “ records updated”);

//Step 8: close the connection
con.close();
}catch(Exception sqlEx){
System.out.println(sqlEx);
}
} // end main
} // end class
```

Compile & Execute

Execute this code in a similar way as we showed you in execution of the last program. Don’t forget to pass the address & name values as the command line arguments.

15.3 References:

- Entire material for this handout is taken from the book **JAVA A Lab Course** by **Umair Javed**. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose without the consent of author.

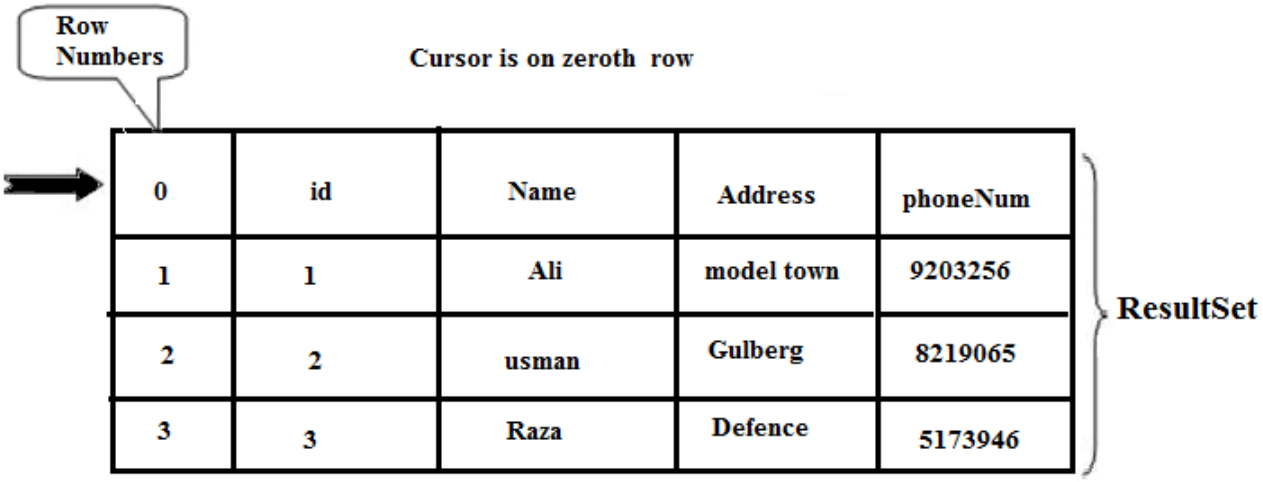
Note: Coding exercises in working condition for this lecture are also available on **“Downloads”** section of LMS.

Lecture 16: Result Set

This handout will familiarize you with another technique of inserting, updating & deleting rows. Before moving on, first we look at ResultSet.

16.1 ResultSet

- A ResultSet contains the results of the SQL query
  - Represented by a table with rows and columns
  - Maintains a cursor pointing to its current row of data.
  - Initially the cursor positioned before the row (0).
  - First row has index 1



16.1.1 Default ResultSet

- A default ResultSet object is not updatable and has a cursor that moves *forward only*.
- You can iterate over through it only *once* and only from the *first row to last row*.
- Until now, we have worked & used it in various examples.
- For a quick overview, here how we create a *default ResultSet* object. String sql = "SELECT * FROM Person"; PreparedStatement pStmt = con.prepareStatement(sql); **ResultSet rs** = pStmt.executeQuery( );

16.1.2 Useful ResultSet's Methods

Following methods are used often to work with *default* ResultSetobject. We already seen and used some of them in code examples.

16.1.2.1 next( )

- Attempts to move to the next row in the ResultSet, if available
- The next() method returns true or false depending upon whether the next row is available (exist) or not.
- Before retrieving any data from ResultSet, always remember to call next()at least once because initially cursor is positioned before first row.

16.1.2.2 getters

- To retrieve the data of the column of the current row you need to use the various getters provided by the ResultSet
- These getters return the value from the column by specifying column name or column index.
- For example, if the column name is "Name" and this column has index 3 in the ResultSet object, then we can retrieve the values by using one of the following methods:
- String name = rs.getString("Name");
- String name = rs.getString(3);
- These getter methods are also available for other types like getInt( ), getDouble( ) etc. Consult the Java API documentation for more references.

Note: Remember that first column has an index 1, NOT zero (0).

16.1.2.3 close( )

- Used to release the JDBC and database resources
- The ResultSet is implicitly closed when the associated Statement object executes a new query or closed by method call.

16.1.2.4 Updatable and/or Scrollable ResultSet

- It is possible to produce ResultSet objects that are scrollable and/or updatable (since JDK 1.2)
- With the help of such ResultSet, it is possible to move forward as well as backward with in ResultSet object.
- Another advantage is, rows can be inserted, updated or deleted by using updatable ResultSet object.

16.1.2.5 Creating Updatable & Scrollable ResultSet

The following code fragment, illustrates how to make a ResultSet object that is scrollable and updatable.

String sql = “SELECT * FROM Person”; PreparedStatement pStmt = con.prepareStatement(sql, **ResultSet.TYPE_SCROLL_INSENSITIVE**, **ResultSet.CONCUR_UPDATABLE**);
ResultSet rs = pStmt.executeQuery( );

Two constants have been used of ResultSet class for producing a ResultSet rs that is scrollable, will not show changes made by others and will be updatable

16.1.2.6 previous( )

- Moves the cursor to the previous row in the ResultSet object, if available
- Returns true if cursor is on a valid row, false it is off the result set.
- Throws exception if result type is TYPE_FORWARD_ONLY.

Example Code 16.1: Use of previous ( ), next( ) & various getters methods

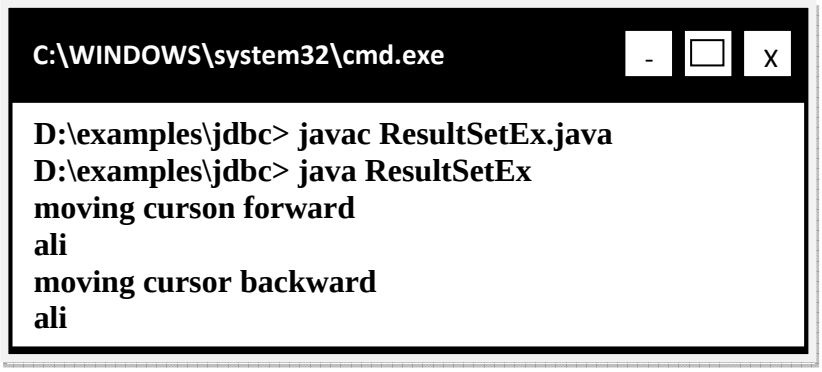
The ResultSetEx.javashows the use of previous, next and getters methods. We are using the same Persontable of *PersonInfo* database, the one we had created earlier in this example and later on.

```
1. // File ResultSetEx.java
2. import java.sql.*;
3. public class ResultSetEx {
4. public static void main (String args[ ]) {
5. try {
6. //Step 2: load driver
7. Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);
8. //Step 3: define the connection URL
9. String url = “jdbc:odbc:personDSN”;
10. //Step 4: establish the connection
11. Connection con = DriverManager.getConnection(url);
12.//Step 5: creating PrepareStatement by passing sql and
13.//ResultSet’s constants so that the ResultSet that will
14.//produce as a result of executing query will be
15.//scrollable & updatable
16.String sql = “SELECT * FROM Person”;
17.PreparedStatement pStmt = con.prepareStatement(sql, ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
20 //Step 6: execute the query
21 ResultSet rs = pStmt.executeQuery();
22 // moving cursor forward i.e. first row
23 rs.next( );
24 // printing column “name” value of current row (first)
```

```
25 System.out.println("moving cursor forward");
26 String name = rs.getString("Name");
27 System.out.println(name);
28 // moving cursor forward i.e. on to second row
29 rs.next( );
   30 // moving cursor backward i.e to first row
31 rs.previous( );
32.// printing column "name" value of current row (first)
33.System.out.println("moving cursor forward");
34.name = rs.getString("Name");
35.System.out.println(name);
36.//Step 8: close the connection
37.con.close();
38.}catch(Exception sqlEx){
39.System.out.println(sqlEx);
40.}
41.} // end main
42.} // end class
```

Compile & Execute:

The sample output is given below:



16.1.2.7 absolute(int)

- Moves the cursor to the given row number in the ResultSet object.
- If given row number is positive, moves the cursor forward with respect to beginning of the result set.
- If the given row number is negative, the cursor moves to the absolute row position with respect to the end of the result set.
- For example, calling absolute(-1) positions the cursor on the last row; calling absolute(-2) moves the cursor to next-to-last row, and so on.
- Throws Exception if ResultSet type is TYPE_FORWARD_ONLY

16.1.2.8 updaters (for primitives, String and Object)

- Used to update the column values in the current row or in insert row (discuss later)
- Do not update the underlying database

- Each update method is overloaded; one that takes column name while other takes column index. For example String updater are available as:

updateString(String columnName, String value)
updateString(String columnIndex, String value)

16.1.2.9 updateRow( )

- Updates the underlying database with new contents of the current row of this ResultSet object

Example Code 16.2: Updating values in existing rows

The following code example updates the *Name* column in the second row of the ResultSet object rs and then uses the method updateRow to update the *Person* table in database.

This code is the modification of the last one. Changes made are shown in bold face.

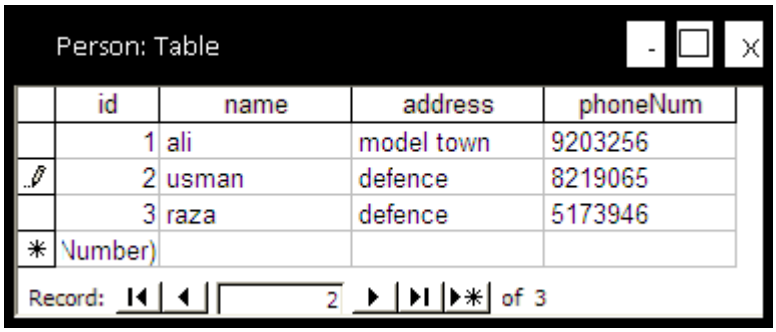
```
1. // File ResultSetEx.java
2. import java.sql.*;
3. public class ResultSetEx {
4. public static void main (String args[ ]) {
5. try {
6. //Step 2: load driver
7. Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
8. //Step 3: define the connection URL
9. String url = "jdbc:odbc:personDSN";
10.//Step 4: establish the connection
11.Connection con = DriverManager.getConnection(url);
12.//Step 5: create PrepareStatement by passing sql and
13.// ResultSet appropriate fields
14.String sql =      "SELECT * FROM Person";
15.PreparedStatement pStmt = con.prepareStatement(sql,
16.ResultSet.TYPE_SCROLL_INSENSITIVE,
17.ResultSet.CONCUR_UPDATABLE);
18.//Step 6: execute the query
19.ResultSet rs = pStmt.executeQuery();
20.// moving cursor to second row
21.rs.absolute(2);
```

```
22.// update address column of 2nd row in rs
23.rs.updateString("Address", "model town");
24.// update the row in database
25.rs.updateRow( );
26.//Step 8: close the connection
27.con.close();
28.}catch(Exception sqlEx){
29.System.out.println(sqlEx);
30.}
31.} // end main
32.} // end class
```

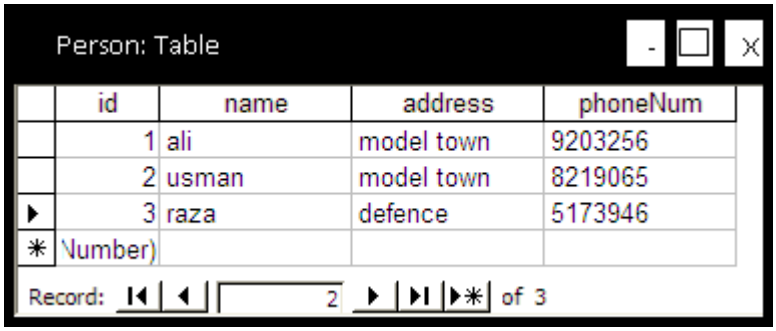
Compile & Execute

Given below are two states of *Person* table. Notice that address of 2nd row is updated.

Person table: Before execution

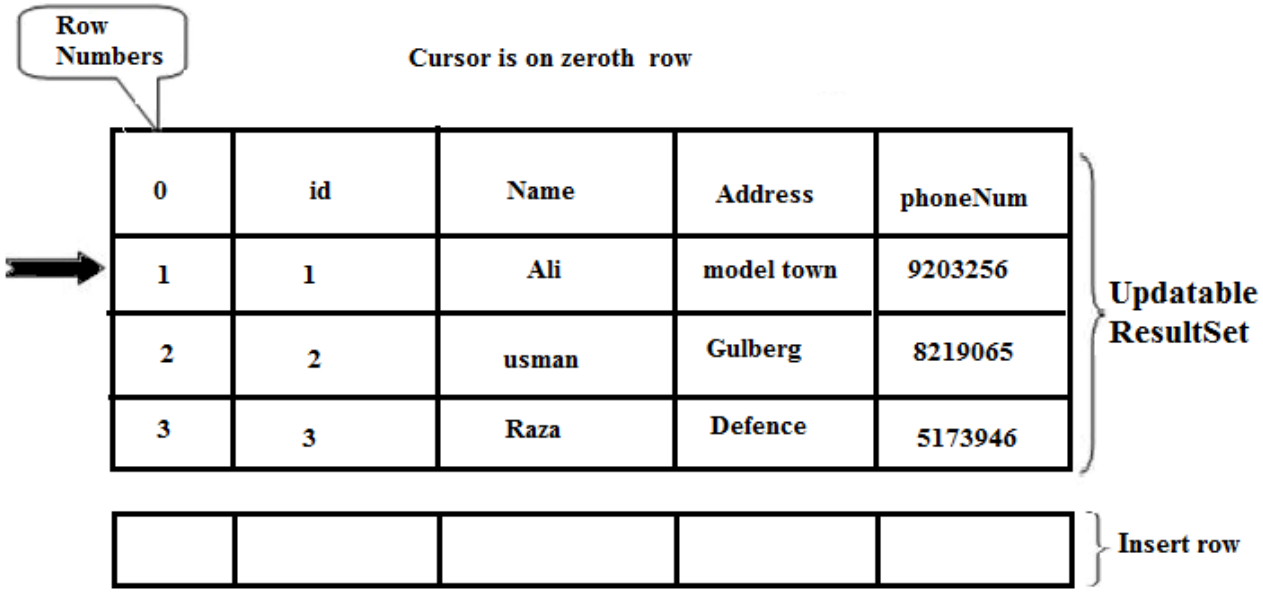


Person table: After execution

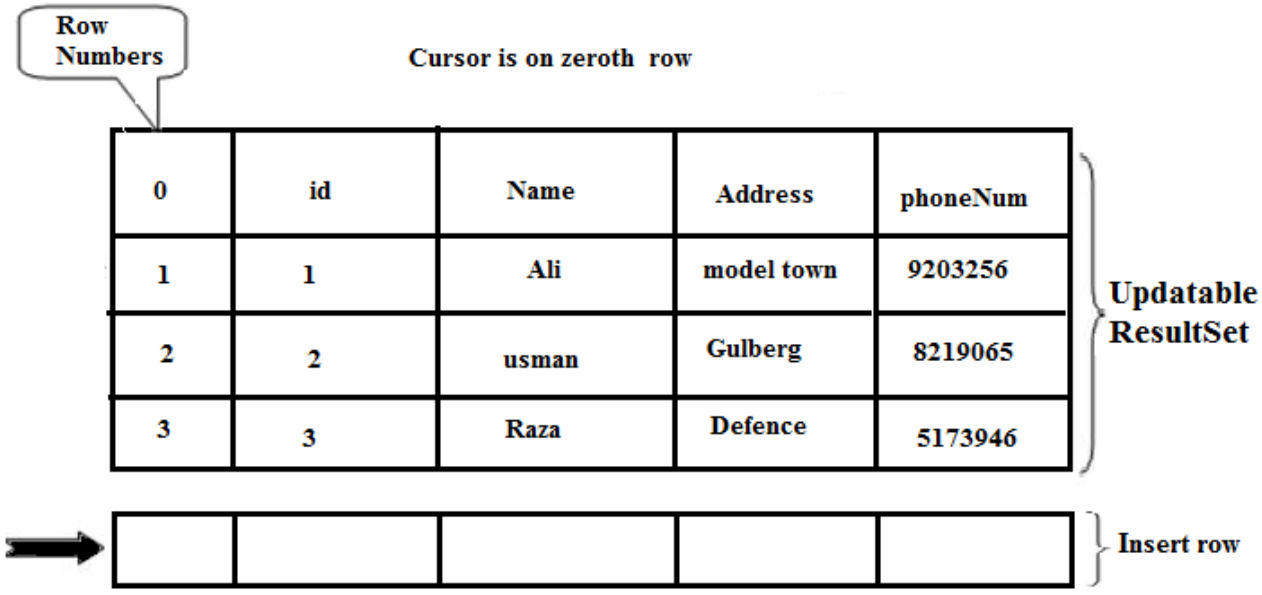


16.1.2.10moveToInsertRow(int)

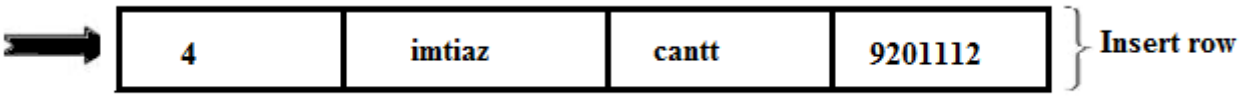
- An updatable resultset object has a special row associate with it i.e. insert row
- Insert row - a buffer, where a new row may be constructed by calling updater methods.
- Doesn't insert the row into a result set or into a database.
- For example, initially cursor is positioned on the first row as shown in the diagram:



- By calling moveToInsertRow( ), the cursor is moved to insert row as shown below:

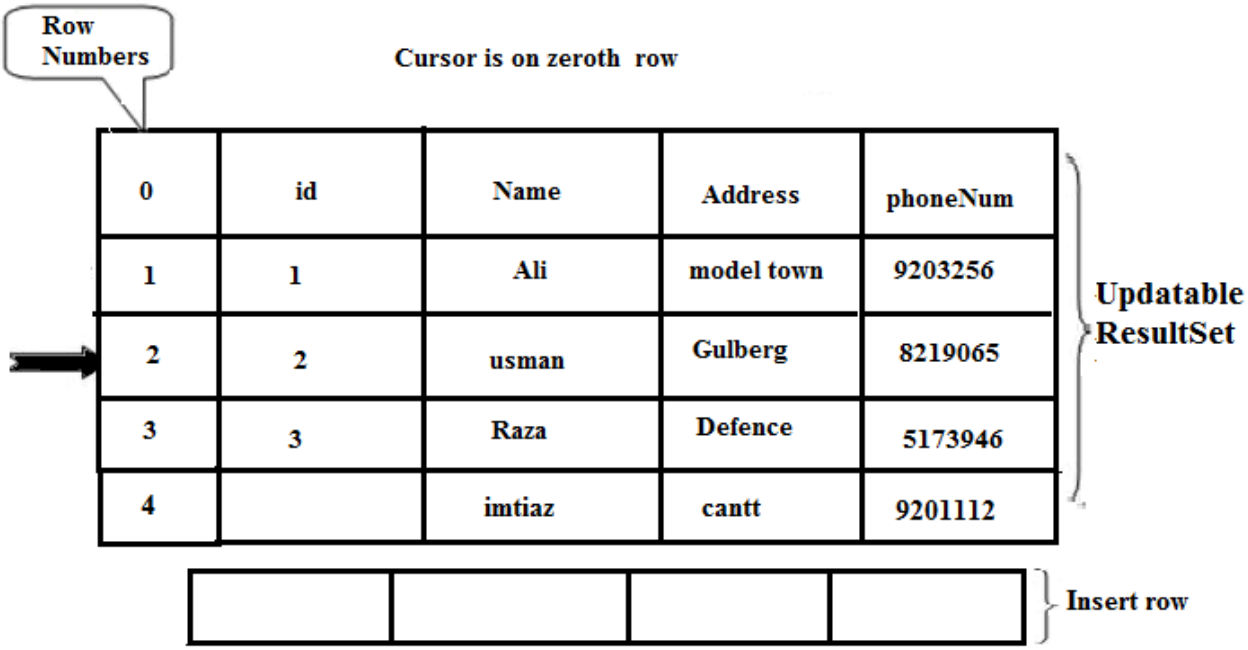


- Now, by calling various updaters, we can insert values into the columns of insert row as shown below.



16.1.2.11 insertRow( )

- Inserts the contents of the current row into this ResultSet object and into the database too.
- Moves the cursor back to the position where it was before calling moveToInsertRow()
- This is shown in the given below diagram



Note: The cursor must be on the insert row before calling this method or exception would be raised.

Example Code 16.3: Inserting new row

The following code example illustrates how to add/insert new row into the ResultSet as well into the database.

This code is the modification of the last one. Changes made are shown in bold face.

```
1. // File ResultSetEx.java
```

```
2. import java.sql.*;
3. public class ResultSetEx {
4. public static void main (String args[ ]) {
5. try {
6. //Step 2: load driver
7. Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

8. //Step 3: define the connection URL
9. String url = "jdbc:odbc:personDSN";

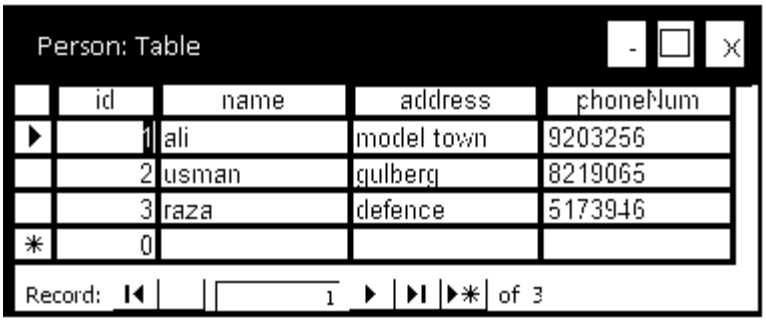
10.//Step 4: establish the connection
11.Connection con = DriverManager.getConnection(url);

12.//Step 5: create PrepareStatement by passing sql and
13.// ResultSet appropriate fields
14.String sql = "SELECT * FROM Person";
15.PreparedStatement pStmt = con.prepareStatement(sql, ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
18.//Step 6: execute the query

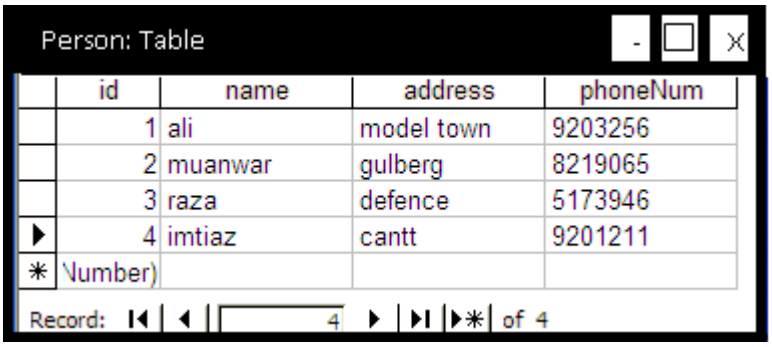
19.ResultSet rs = pStmt.executeQuery();
20.// moving cursor to insert row
21.rs.moveToInsertRow();
22.// updating values in insert row
23.rs.updateString( "Name"     ,          "imitiaz" );
24.rs.updateString( "Address" ,          "cantt" );
25.rs.updateString( "phoneNum" ,      "9201211" );
26.// inserting row in resultset & into database
27.rs.insertRow( );
28.//Step 8: close the connection
29.con.close();
30.}catch(Exception sqlEx){
31.System.out.println(sqlEx);
32.}
33.} // end main
34.} // end class
```

Compile & Execute

Given below are two states of *Person* table. Note that after executing program, a newly added row is present.



Person table: Before execution



Person table: After execution

16.1.2.12 last( ) & first( )

- Moves the cursor to the last & first row of the ResultSet object respectively.
- Throws exception if the ResultSet is TYPE_FORWARD_ONLY

16.1.2.13 getRow( )

- Returns the current row number
- As mentioned earlier, the first row has index 1 and so on.

16.1.2.14 deleteRow( )

- Deletes the current row from this ResultSet object and from the underlying database.
- Throws exception if the cursor is on the insert row.

Example Code 16.4: Deleting existing row

The given below example code shows the usage of last( ), getRow( )and

deleteRow( )method.

This code is also the modification of the last one. Changes made are shown in bold face.

```
1. // File ResultSetEx.java
2. import java.sql.*;
3. public class ResultSetEx {
4. public static void main (String args[ ]) {
5.       try {

6.       //Step 2: load driver
7.       Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

8.       //Step 3: define the connection URL
9.       String url = "jdbc:odbc:personDSN";

10.      //Step 4: establish the connection
11.      Connection con = DriverManager.getConnection(url);

12.      //Step 5: create PrepareStatement by passing sql and

13.      //         ResultSet appropriate fields
14.      String sql =     "SELECT * FROM Person";

15.      PreparedStatement pStmt = con.prepareStatement(sql,

16.      ResultSet.TYPE_SCROLL_INSENSITIVE,
17.      ResultSet.CONCUR_UPDATABLE);

18.      //Step 6: execute the query
19.      ResultSet rs = pStmt.executeQuery();

20.      // moves to last row of the resultset
21.      rs.last();

22.      // retrieving the current row number
23.      int rNo = rs.getRow();

24.      System.out.println("current row number" + rNo);

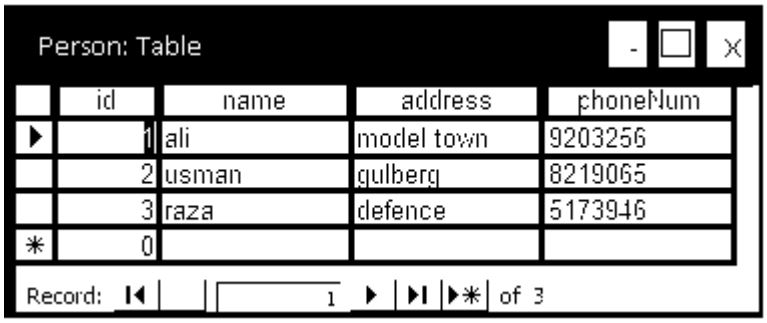
25.      // delete current row from rs & db i.e. 4 because
26.      // previously we have called last() method
27.      rs.deleteRow( );
28.      //Step 8: close the connection
29.      con.close();
30.      }catch(Exception sqlEx){
31.      System.out.println(sqlEx);
```

```
32.     }

33.     } // end main
34.     } // end class
```

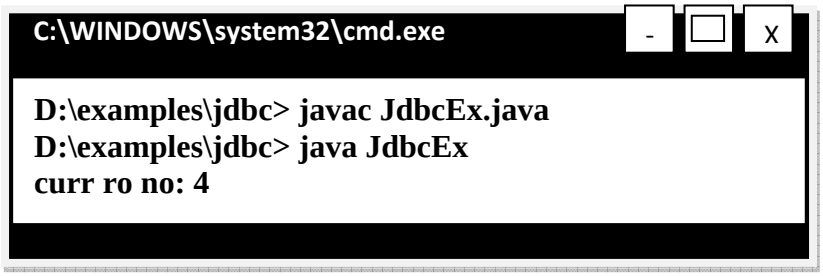
Compile & Execute

The first diagram shows the *Person* table before execution.



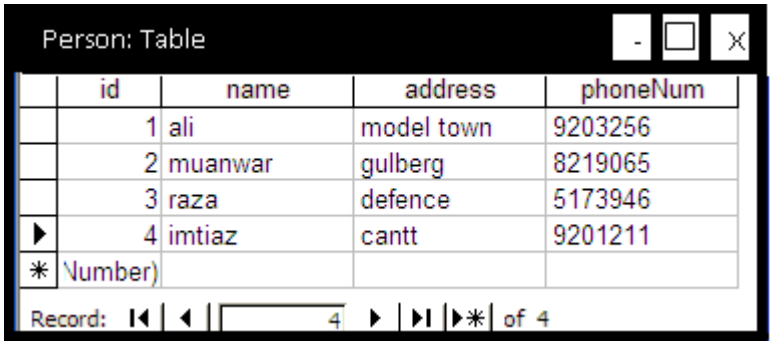
Person table: Before execution

Execution program from command prompt will result in displaying current row number on console. This can be confirmed from following diagram.



Executing Program from Command Prompt

After execution, the last row (4) is deleted from ResultSet as well as from database. The *Person* table is shown after execution



Person table: After execution

16.2 References:

- Entire material for this handout is taken from the book **JAVA A Lab Course** by **Umair Javed**. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose without the consent of author.

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

© Virtual University of Pakistan

Lecture 17: Meta Data

In simple terms, Meta Data is *data (information) about data*. The actual data has no meaning without existence of Meta data. To clarify this, let's look at an example. Given below are listed some numeric values

1000
2000
4000

What this information about? We cannot state accurately. These values might be representing some one's salaries, price, tax payable & utility bill etc. But if we specify Meta data about this data like shown below:

Salary
1000
2000
4000

Now, just casting a glance on these values, you can conclude that it's all about some ones salaries.

17.1 ResultSet Meta data

ResultSet Meta Data will help you in answering such questions

- How many columns are in the ResultSet?
- What is the name of given column?
- Are the column name case sensitive?
- What is the data type of a specific column?
- What is the maximum character size of a column?
- Can you search on a given column?

17.1.1 Creating ResultSetMetaData object

From a ResultSet (the return type of executeQuery()), derive a ResultSetMetaDataobject by calling getMetaData()method as shown in the given code snippet (here rsis a valid ResultSetobject):

Now, rsmd can be used to look up number, names & types of columns.

17.1.2 Useful ResultSetMetaData methods

17.1.2.1 getColumnCount ( )

- Returns the number of columns in the result set

17.1.2.2 getColumnDisplaySize (int)

- Returns the maximum width of the specified column in characters

17.1.2.3 getColumnName(int) / getColumnLabel (int)

- The getColumnName() method returns the database name of the column
- The getColumnLabel() method returns the suggested column label for printouts

17.1.2.4 getColumnType (int)

- Returns the SQL type for the column to compare against types in java.sql.Types

Example Code 17.1: Using ResultSetMetaData

The *MetaDataEx.java* will print the column names by using ResultSetMetaData object and column values on console. This is an excellent example of the scenario where we have no idea about the column names in advance.

Note: For this example code and for the coming ones, we are using the same database (*PersonInfo*) the one we created earlier and repeatedly used. Changes are shown in bold face

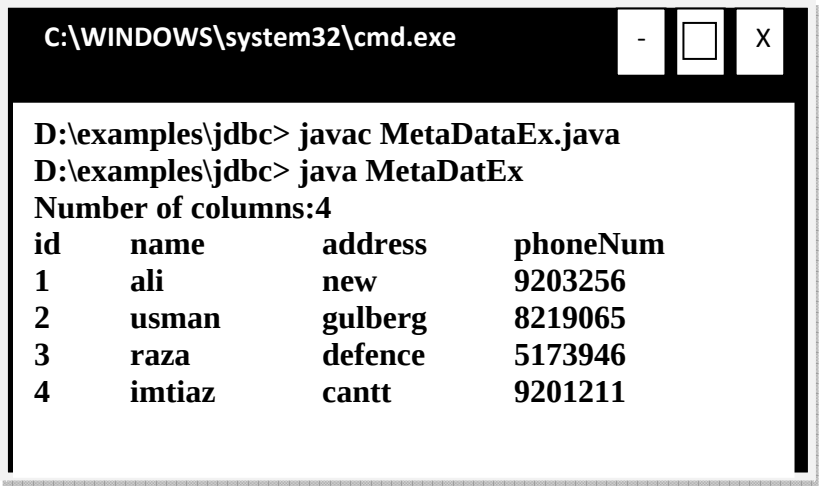
```
44. // File MetaDataEx.java
45. import java.sql.*;
46. public class MetaDataEx {
47. public static void main (String args[ ]) {
```

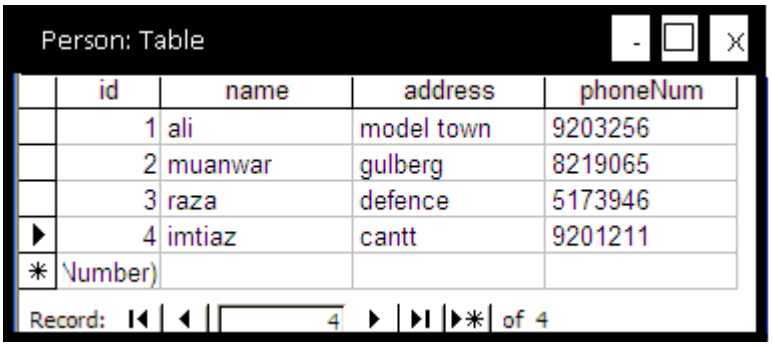
```
48. try {
49. //Step 2: load driver
50. Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
51. //Step 3: define the connection URL
52. String url = "jdbc:odbc:personDSN";
53. //Step 4: establish the connection
54. Connection con = null;
55. con = DriverManager.getConnection(url, "", "");
56. //Step 5: create PrepareStatement by passing sql and
57. //          ResultSet appropriate fields
58. String sql =      "SELECT * FROM Person";
59. PreparedStatement pStmt = con.prepareStatement(sql,
60. ResultSet.TYPE_SCROLL_INSENSITIVE,
61. ResultSet.CONCUR_UPDATABLE);
62. //Step 6: execute the query
63. ResultSet rs = pStmt.executeQuery();
64. // get ResultSetMetaData object from rs
65. ResultSetMetaData rsmd = rs.getMetaData( );
66. // printing no. of column contained by rs
67. int numColumns = rsmd.getColumnCount();
68. System.out.println("Number of Columns:" + numColumns);
69. // printing all column names by using for loop
70. String cName;
71. for(int i=1; i<= numColumns; i++) {
72. cName = rsmd.getColumnName(i);
73. System.out.println(cName);
74. System.out.println("\t");
75. }
76. // changing line or printing an empty string
77. System.out.println(" ");
78. // printing all values of ResultSet by iterating over it
79. String id, name, add, ph;
80. while( rs.next() )
81. {
82. id= rs.getString(1);
83. name = rs.getString(2);
84. add= rs.getString(3);
85. ph= rs.getString(4);
86. System.out.println(id);
87. System.out.println("\t");
88. System.out.println(name);
89. System.out.println("\t");
90. System.out.println(add);
91. System.out.println("\t");
92. System.out.println(ph);
93. System.out.println(" ");
```

```
94. }
95. //Step 8: close the connection
96. con.close();
97. }catch(Exception sqlEx){
98. System.out.println(sqlEx); }
99. } // end main
100.} // end class
```

Compile & Execute:

The database contains the following values at the time of execution of this program. The database and the output are shown below:





7.2 DatabaseMetaData

Database Meta Data will help you in answering such questions

- What SQL types are supported by DBMS to create table?
- What is the name of a database product?
- What is the version number of this database product?
- What is the name of the JDBC driver that is used?

17.2.1 Creating DatabaseMetaData object

From a Connection object, a DatabaseMetaData object can be derived. The following code snippet demonstrates how to get DataBaseMetaDataobject.

```
Connectioncon= DriverManager.getConnection(url, usr, pwd);
DatabaseMetaData dbMetaData = con.getMeataData();
```

Now, you can use the dbMetaData to gain information about the database.

17.2.2 Useful ResultSetMetaData methods

17.2.2.1 getDatabaseProductName( )

- Returns the name of the database's product name

17.2.2.2 getDatabaseProductVersion( )

- Returns the version number of this database product

17.2.2.3 getDriverName( )

- Returns the name of the JDBC driver used to established the connection

17.2.2.4 isReadOnly( )

- Retrieves whether this database is in read-only mode
- Returns true if so, false otherwise

Example Code 17.2: using DatabaseMetaData

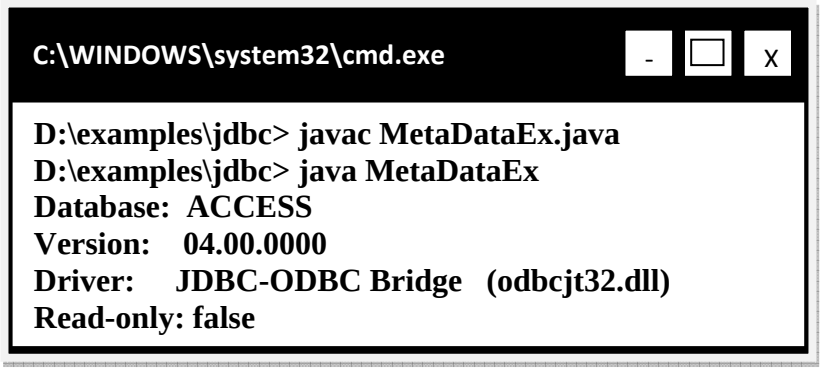
This code is modification of the example code 17.1. Changes made are shown in bold face.

```
102. // File MetaDataEx.java
103. import java.sql.*;
104. public class MetaDataEx {
105.         public static void main (String args[ ]) {
106.         try {
107.         //Step 2: load driver
```

```
108.        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
109.        //Step 3: define the connection URL
110.        String url = "jdbc:odbc:personDSN";
111.        //Step 4: establish the connection
112.        Connection con = null;
113.        con = DriverManager.getConnection(url, "", "");
114.        // getting DatabaseMetaDat object
115.        DatabaseMetaData dbMetaData = con.getMetaData();
116.        // printing database product name
117.        String pName = dbMetaData.getDatabaseProductName();
118.        System.out.println("DataBase: " + pName);
119.        // printing database product version
120.        String pVer = dbMetaData.getDatabaseProductVersion();
121.        System.out.println("Version: " + pVer);
122.        // printing driver name used to establish connection &
123.        // to retrieve data
124.        String dName = dbMetaData.getDriverName();
125.        System.out.println("Driver: " + dName);
126.        // printing whether database is read-only or not
127.        boolean rOnly = dbMetaData.isReadOnly();
128.        System.out.println("Read-Only: " + rOnly);
129.        // you can create & execute statements and can
130.        // process results over here if needed
131.        //Step 8: close the connection
132.        con.close();
133.        }catch(Exception sqlEx){
134.        System.out.println(sqlEx);
135.        }
136.        } // end main
137.} // end class
```

Compile & Execute

On executing the above program, the following output will produce:



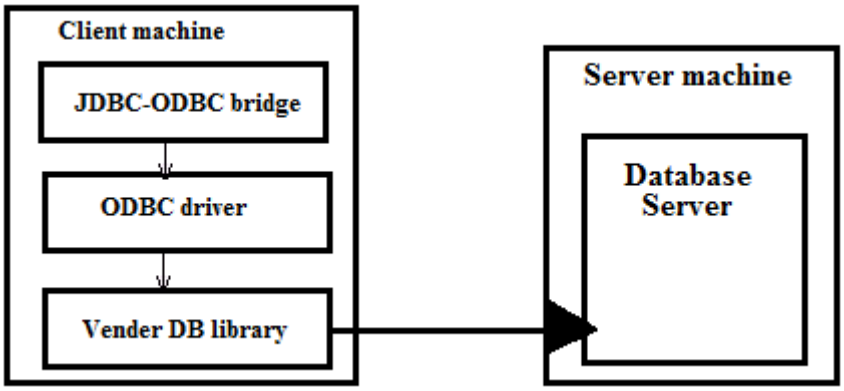
17.3 JDBC Driver Types

- JDBC Driver Types are divided into four types or levels.
- Each type defines a JDBC driver implementation with increasingly higher level of platform independence, performance, deployment and administration.
- The four types are:
  - Type - 1: JDBC - ODBC Bridge
  - Type 2: Native - API/partly Java driver
  - Type 3: Net - protocol/all-Java driver
  - Type 4: Native - protocol/all-Java driver

Now, let's look at each type in more detail

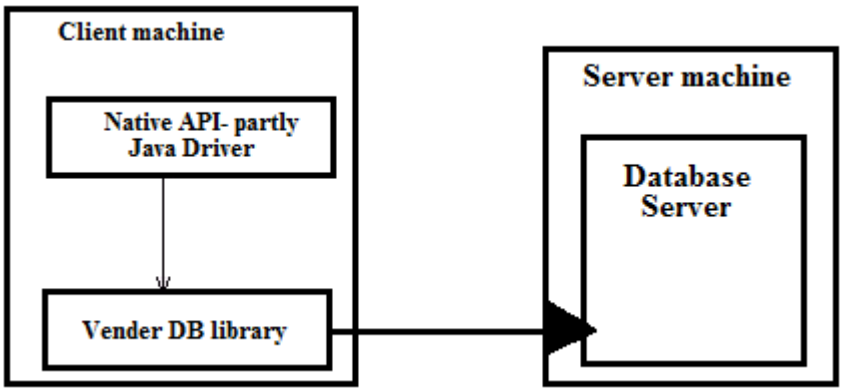
17.3.1 Type - 1: JDBC - ODBC Bridge

- Translates all JDBC calls into ODBC (Open Database Connectivity) calls and send them to the ODBC Driver
- Generally used for Microsoft database.
- Performance is degraded



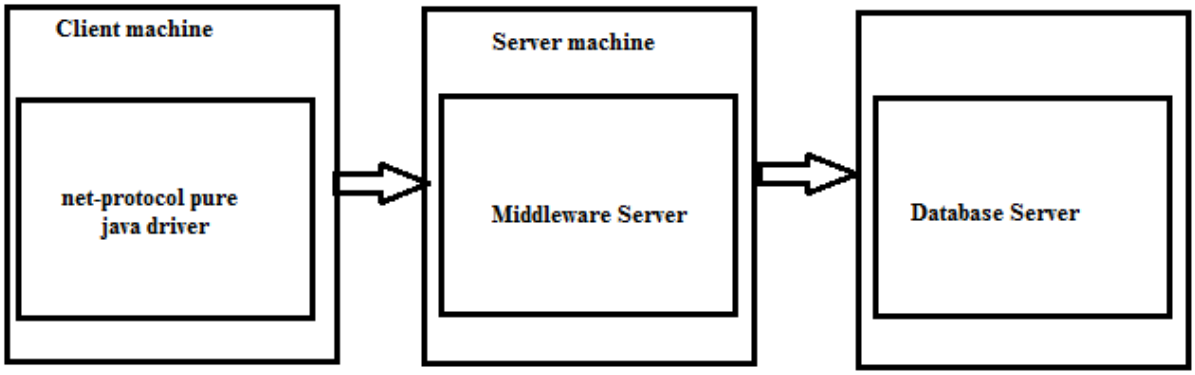
17.3.2 Type - 2: Native - API/partly Java driver

- Converts JDBC calls into database-specific calls such as SQL Server, Informix, Oracle or Sybase.
- Partly-Java drivers communicate with database-specific API (which may be in C/C++) using the Java Native Interface.
- Significantly better Performance than the JDBC-ODBC bridge.



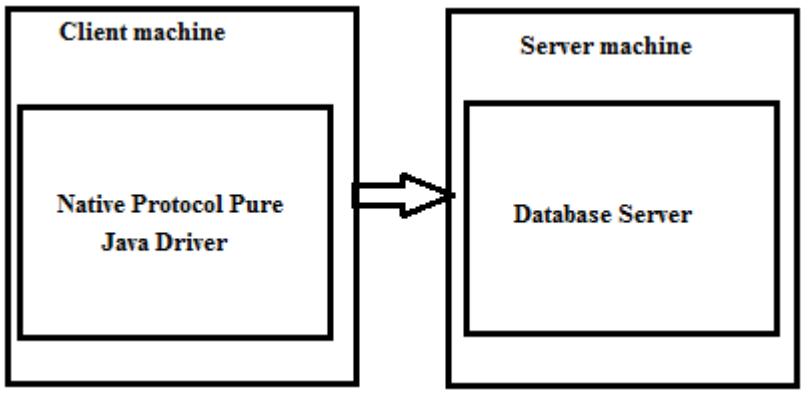
17.3.3Type - 3: Net - protocol/all-Java driver4

- Follows a three-tiered approach whereby the JDBC database requests () are passed through the network to the middle-tier server
- Pure Java client to server drivers which send requests that are not database-specific to a server that translates them into a database-specific protocol.
- If the middle-tier server is written in java, it can use a type 1or type 2JDBC driver to do this



17.3.4Type - 4: Native - protocol / all - java driver

- Converts JDBC calls into the vendor-specific DBMS protocol so that client application can communicate directly with the database server
- Completely implemented in Java to achieve platform independence and eliminate deployment issues.
- Performance is typically very good



17.4 Online Resources

- Sun's JDBC Site
  http://java.sun.com/products/jdbc/
- JDBC Tutorial
  http://java.sun.com/docs/books/tutorial/jdbc/
- List of available JDBC Drivers
  http://industry.java.sun.com/products/jdbc/drivers/
- RowSet Tutorial
  http://java.sun.com/developer/Books/JDBCTutorial/chapter5.html
- JDBC RowSets Implementation Tutorial

  http://java.sun.com/developer/onlineTraining/

  Database/**jdbcrowset**s.pdf

17.5 References:

- Java API documentation 5.0
- Java - A Lab Course by Umair Javed
- JDBC drivers in the wild
- http://www.javaworld.com/javaworld/jw-07-2000/jw-0707-jdbc_p.html

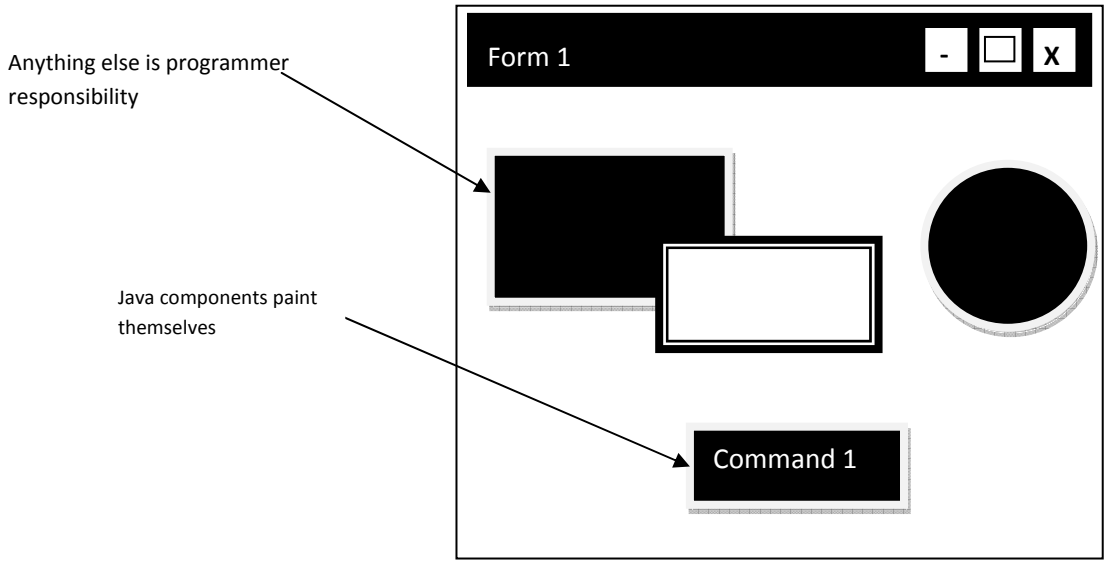
Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 18: Java Graphics

18.1 Painting

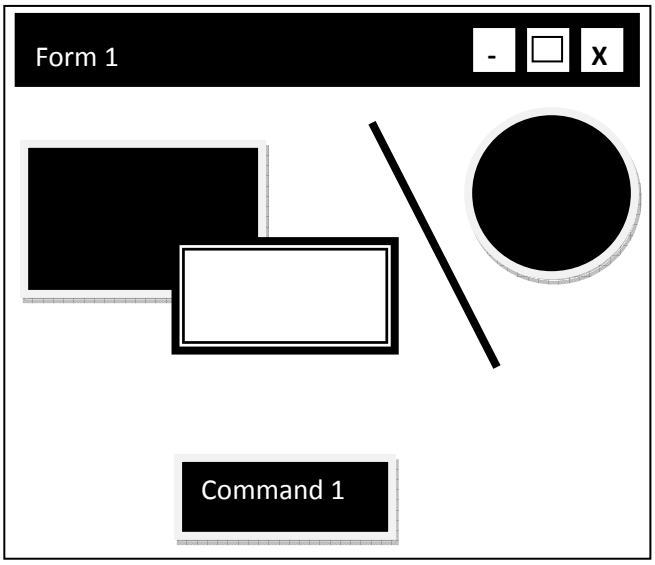
Window is like a painter's canvas. All window paints on the same surface. More importantly, windows don't remember what is under them. There is a need to repaint when portions are newly exposed.

Java components are also able to paint themselves. Most of time, painting is done automatically. However sometimes you need to do drawing by yourself.

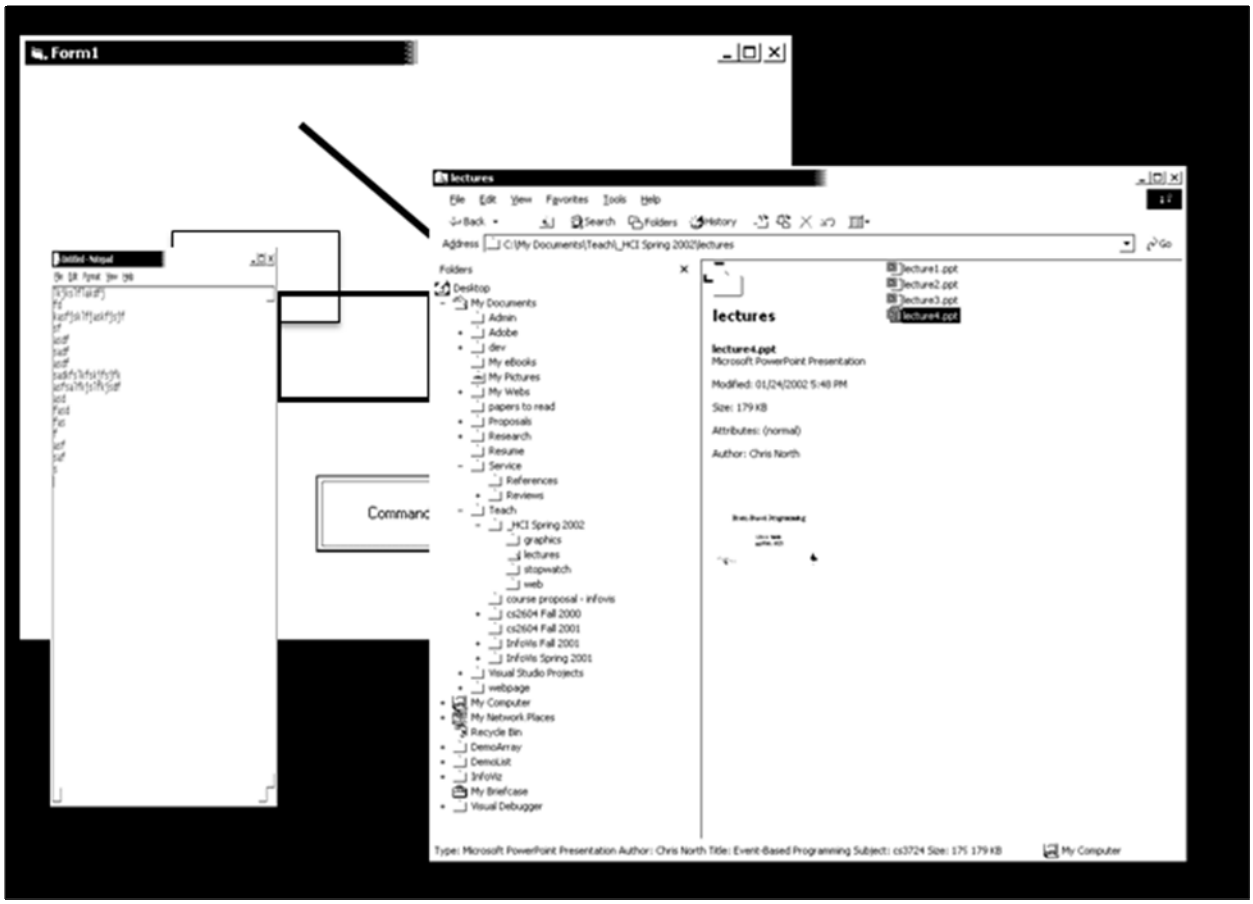


18.1.1 How painting works?

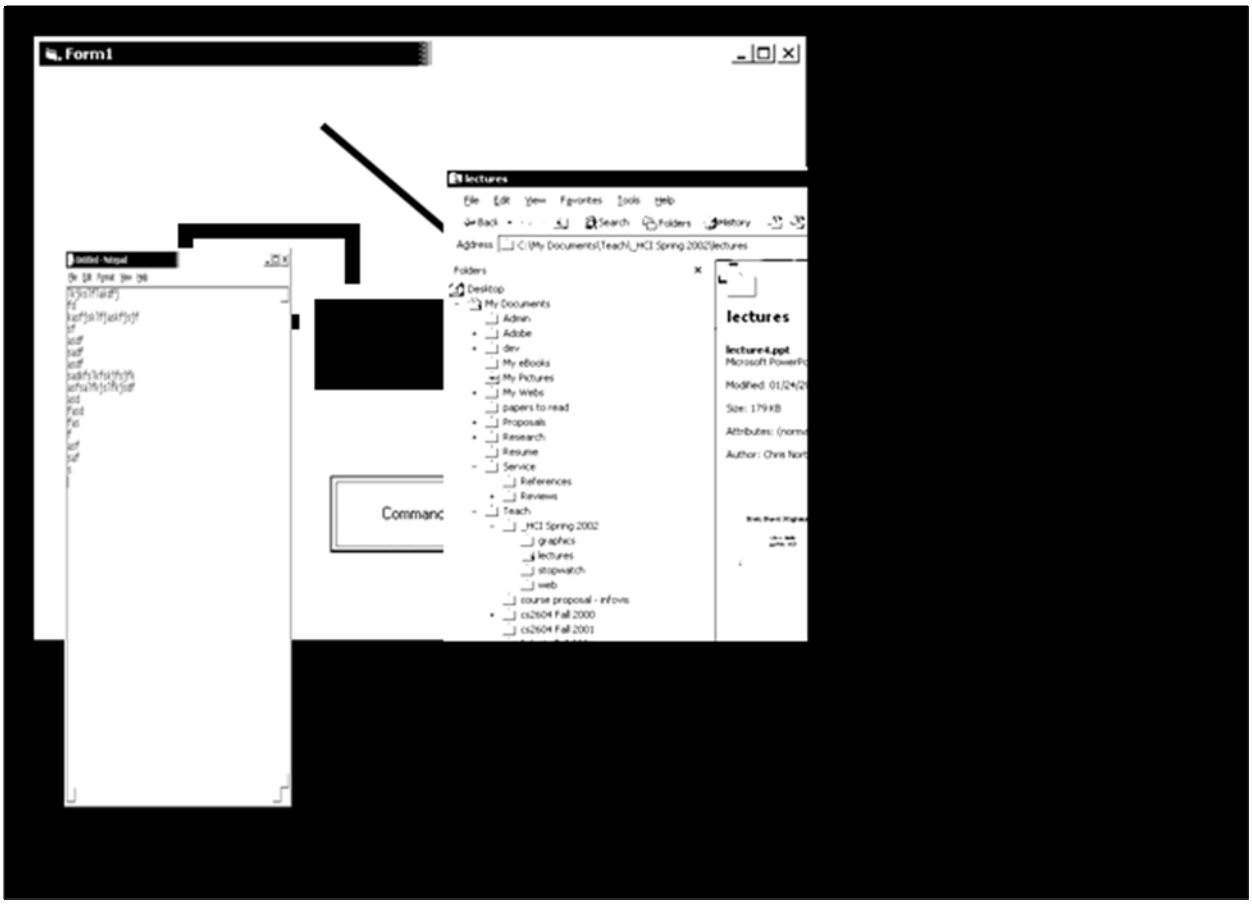
Let's take windows example. Consider the following diagram in which the blue area is representing the desktop. The one frame (myApp) is opened in front of desktop with some custom painting as shown below.



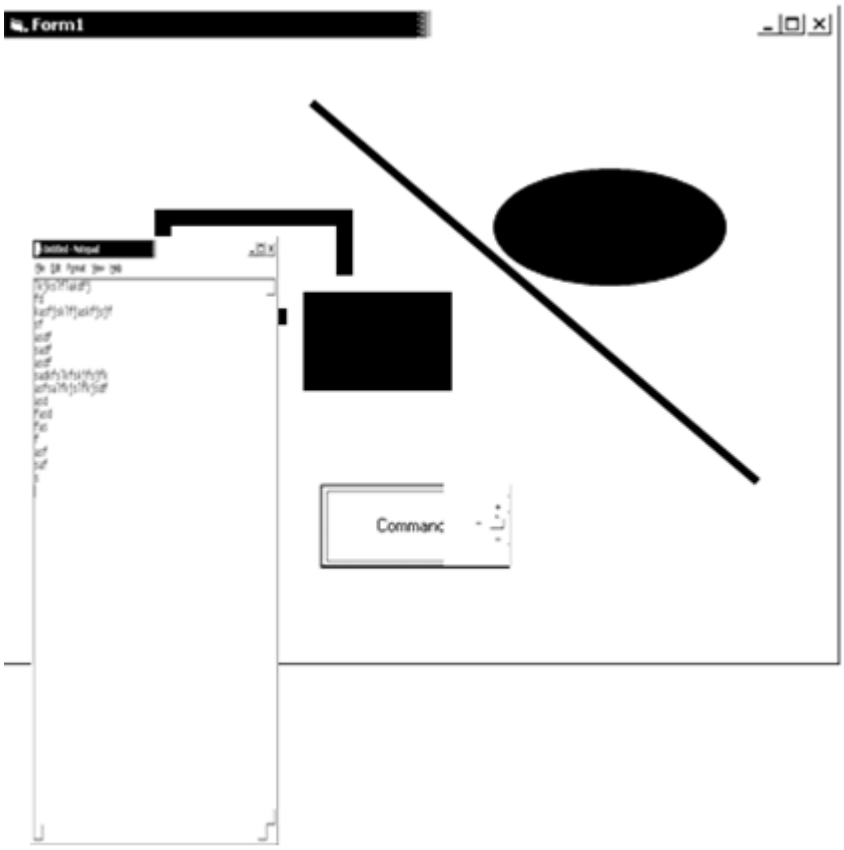
myApp consist of a JPanel. The JPanel contains a JButton. Two rectangles, a circle & a lines are also drawn on the JPanel.fter opening notepad and windows explorer window, diagram will look like this:



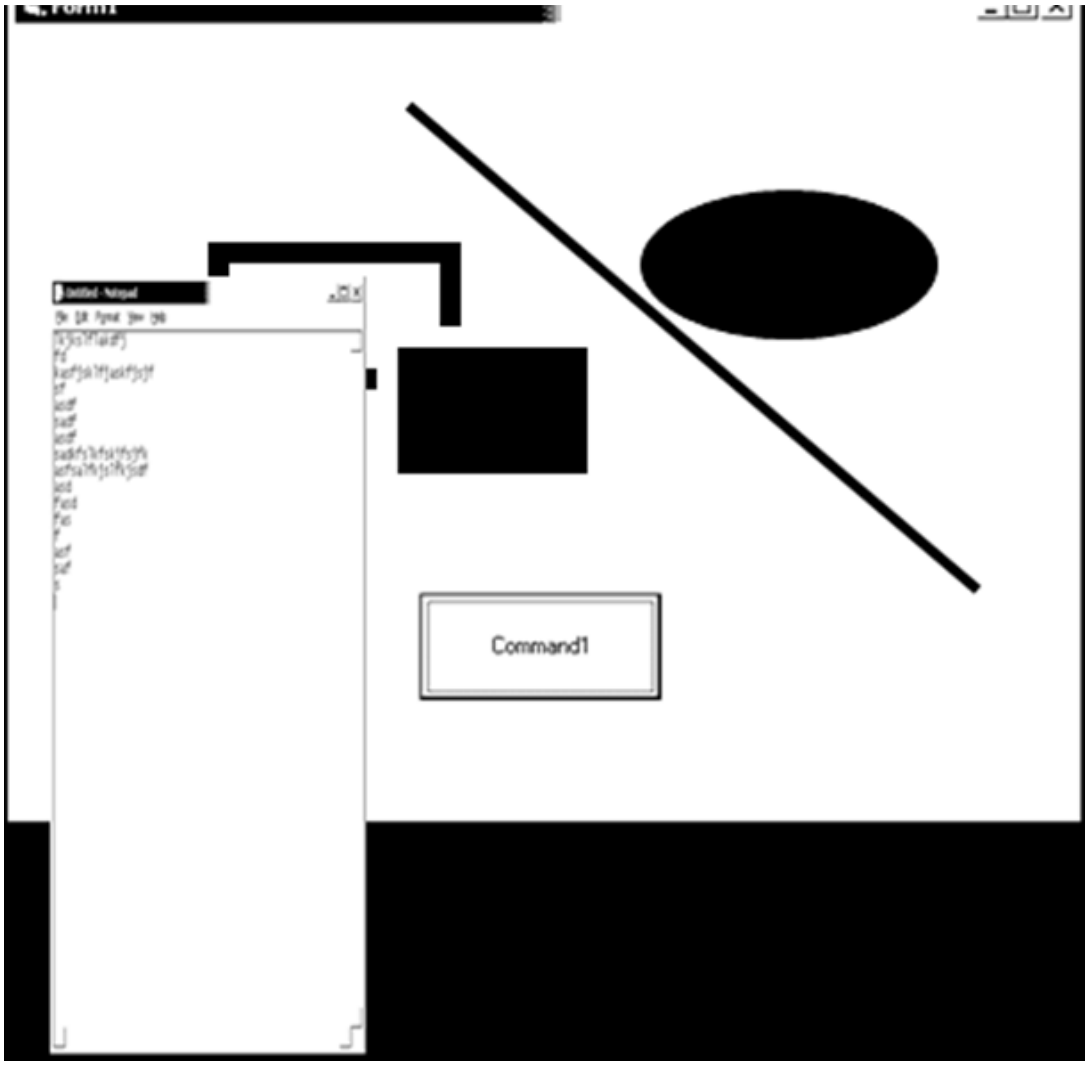
Lets shuts off the windows explorer, the repaint event is sent to desktop first and then to myApp. The figure shown below describes the situation after desktop repaint event get executed. Here you can clearly see that only desktop repaints itself and window explorer remaining part is still opened in front of myApp.



The following figure shows the situation when myApp's JPanel calls its repaint method. Notice that some portion of window explorer is still remains in front of JButton because yet not repaint event is sent to it.



Next, JPanel forwards repaint event to JButton that causes the button to be displayed in its original form. This is all done automatically and we cannot feel this process cause of stunning speed of modern computers that performs all these steps in flash of eye.



18.1.2Painting a Swing Component

Three methods are at the heart of painting a swing component like JPanel etc. For instance, paint()gets called when it's time to render -- then Swing further factors the paint() call into three separate methods, which are invoked in the following order:

protected void paintComponent(Graphics g) protected void
paintBorder(Graphics g) protected void
paintChildren(Graphics g)

Let’s look at these methods in order in which they get executed

18.1.2.1 paintComponent( )

- It is a main method for painting
- By default, it first paints the background
- After that, it performs custom painting (drawing circle, rectangles etc.)

18.1.2.2 paintBorder( )

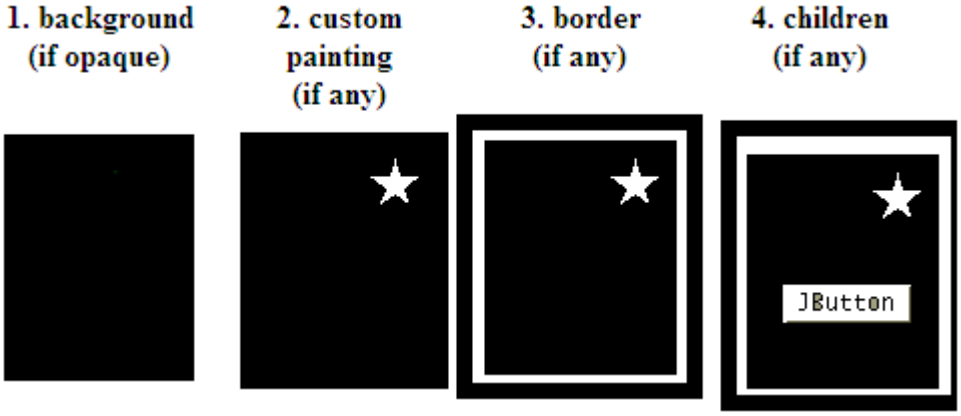
- Tells the components border (if any) to paint.
- It is suggested that you do not override or invoke this method

18.1.2.3 paintChildren( )

- Tells any components contained by this component to paint themselves
- It is suggested that you do not override or invoke this method too.

Example: Understanding methods calls

Consider the following figure



The figure above illustrates the order in which each component that inherits from JComponent paint itself.

Figure 1 to 2 -painting the background and performing custom painting is performed by the *paintComponent* method

In Figure 3 - *paintBorder* is get called
And finally in figure 4 - *paintChildern* is called that causes the JButton to render itself.

Note: The important thing to note here is for JButton (since it is a JComponent), all these methods are also called in the same order.

Your Painting Strategy

- You must follow the three steps in order to perform painting.

Subclass JPanel

- class MyPanel *extends JPanel*
- Doing so MyPanel also becomes a JPanle due to inheritance

Override the paintComponent(Graphics g) method

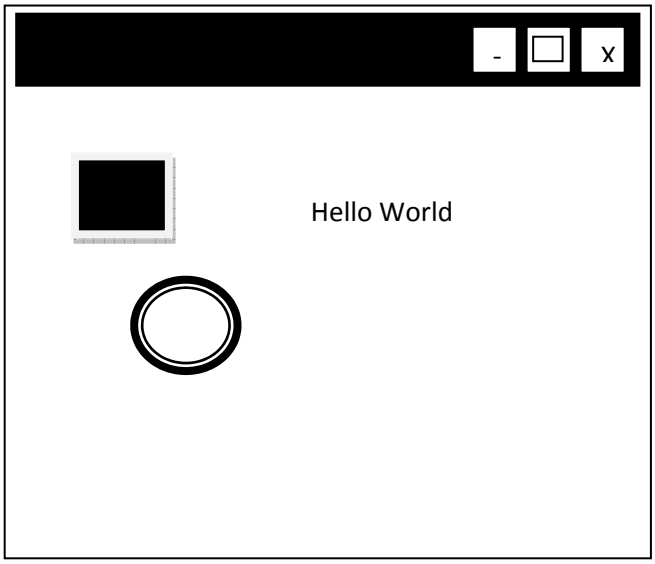
- Inside method using graphics object, do whatever drawing you want to do

Install that JPanel inside a JFrame

- When frame becomes visible through the paintChildren() method your panel become visible
- To become visible your panel will call paintComponent() method which will do your custom drawing

Example Code 18.1:

Suppose we want to draw one circle & rectangle and a string "Hello World".




```
// importing required packages
import javax.swing.*;
import java.awt.*;
// extending class from JPanel
public class MyPanel extends JPanel {
// overriding paintComponent method
public void paintComponent(Graphics g){
// erasing behaviour - this will clear all the
```

```
// previous painting
super.paintComponent(g);

// Down casting Graphics object to Graphics2D
Graphics2D g2 = (Graphics2D)g;
// drawing rectanle
g2.drawRect(20,20,20,20);
// changing the color to blue
g2.setColor(Color.blue);
// drawing filled oval with color i.e. blue
g2.fillOval(50,50,20,20);
// drawing string
g2.drawString("Hello World", 120, 50);
}// end paintComponent
} // end Test class
```

The Test class that contains the main method as well uses MyPanel (previously built) class is given below:

```
// importing required packages
import javax.swing.*;
import java.awt.*;
public class Test {
JFrame f;
// declaring Reference of MyPanel class
MyPanel p;
// parameter less constructor public Test(){
f = new JFrame();
Container c = f.getContentPane();
c.setLayout(new BorderLayout());
// instantiating reference
p = new MyPanel();
// adding MyPanel into container
c.add(p);
f.setSize(400,400);
f.setVisible(true);
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
} // end constructor
// main method
```

```
public static void main(String args[ ]){
Test t = new Test();
}
} // end Test class
```

Note: Here we have used only some methods (drawRect( ) & fillOval( ) etc.) of Graphics class. For a complete list, see the Java API documentation.

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 19: How to Animate?

If we want to animate something like ball, moving from one place to another, we constantly need to call paintComponent( ) method and to draw the shape (ball etc.) at new place means at new coordinates.
Painting is managed by system, so calling paintComponent() directly is not recommended at all. Similarly calling paint( ) method is also not recommended. Why? Because such code may be invoked at times when it is not appropriate to paint -- for instance, before the component is visible or has access to a valid Graphicsobject.
Java gives us a solution in the form of *repaint( )* method. Whenever we need to repaint, we call this method that in fact makes a call to *paint( )* method at appropriate time.

19.1 Problem & Solution

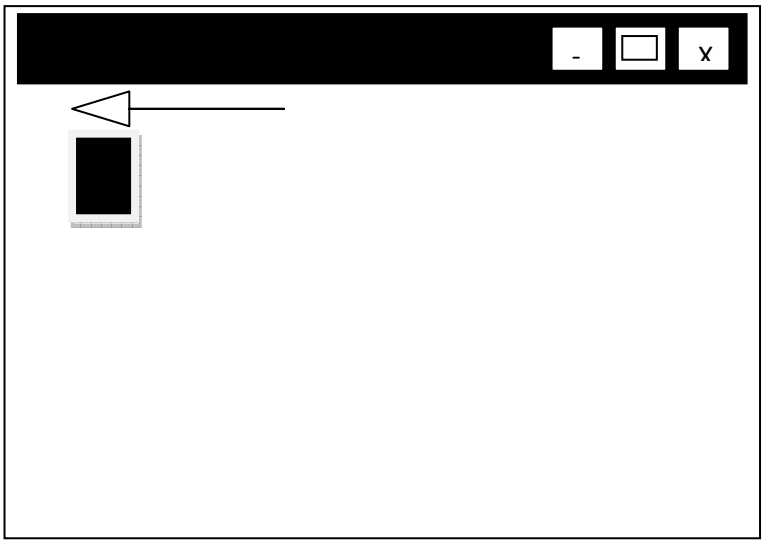
- What to do to move the shapes present in example code 18.1 (last example) when a mouse is dragged
- First time painting is what we already have done
- When a mouse is clicked find the co-ordinates of that place and paint Rectangle at that place by requesting, using repaint() call
- Here instead of Hard-coding the position of co-ordinates uses some variables. For example mx, my
    - In the last example code, we draw a rectangle by passing hard-coded values like 20

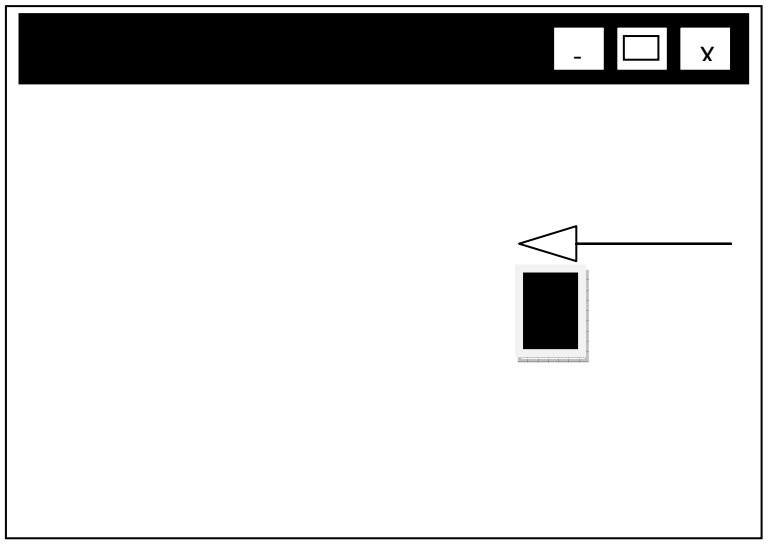
      g.drawRect(20,20,20,20);
    - Now, we'll use variables so that change in a variable value causes to display a rectangle at a new location

      g.drawRect(**mx**,**my**,20,20);
- Similarly, you have seen a tennis game (during lecture). Now, what to do code the paddle movement.
- In the coming up example. We are doing it using mouse, try it using mouse.

Example Code 19.1

The following outputs were produced when mouse is dragged from one location to anther





First we examine the MyPanel.java class that is drawing a filled rectangle.

```
import javax.swing.*;
import java.awt.*;
// extending class from JPanel
public class MyPanel extends JPanel {
// variables used to draw rectangles at different locations
int mX = 20;
int mY = 20;
// overriding paintComponent method
public void paintComponent(Graphics g){
// erasing behaviour - this will clear all the previous painting
super.paintComponent(g);
// Down casting Graphics object to Graphics2D
Graphics2D g2 = (Graphics2D)g;
// changing the color to blue
g2.setColor(Color.blue);
// drawing filled oval with color i.e. blue
// using instance variables
g2.fillRect(mX,mY,20,20);
}// end paintComponent
```

```
} // end MyPanel class
```

The Test class is given below. Additionally this class also contains the code for handling mouse events.

```
// importing required packages
 import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class Test {
JFrame f;
// declaring Reference of MyPanel class
MyPanel p;
// parameter less constructor
public Test(){
f = new JFrame();
Container c = f.getContentPane();
c.setLayout(new BorderLayout());
// instantiating reference
p = new MyPanel();
// adding MyPanel into container
c.add(p);
f.setSize(400,400);
f.setVisible(true);
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// creating inner class object
Handler h = new Handler();
// registering MyPanel to handle events
p.addMouseMotionListener(h);
} // end constructor
// inner class used for handling events
public class Handler extends MouseMotionAdapter{
// capturing mouse dagged events
public void mouseDragged(MouseEvent me){
// getting the X-Position of mouse and assigning
// value to instance variable mX of MyPanel class
p.mX =me.getX();
// getting the Y-Position of mouse and assigning
// value to instance variable mX of MyPanel class
p.mY = me.getY();
// call to repaint causes rectangle to be drawn on
// new location
```

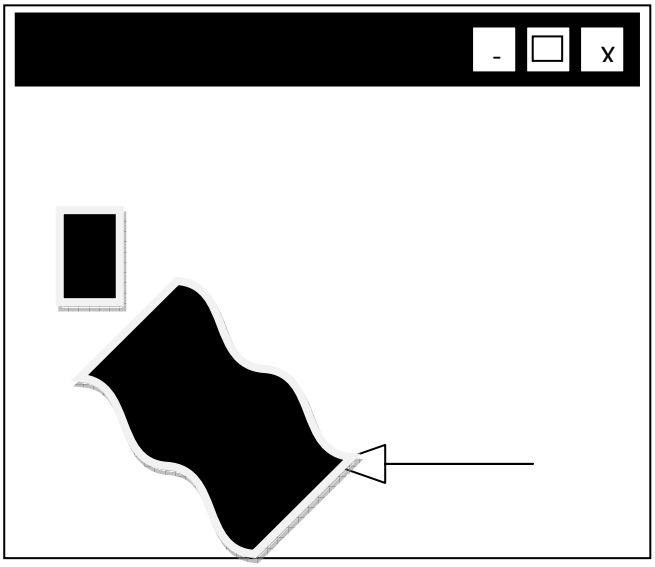
```
p.repaint() ;
} // end mouseDragged
} // end Handler class
// main method
public static void main(String args[ ]){
Test t = new Test();
}
} // end MyPanel class
```

On executing this program, when you drag mouse from one location to another, rectangle is also in sync with the movement of mouse. Notice that previously drawn rectangle is erased first.

If we exclude or comment out the following line from MyPanel class

```
super.paintComponent(g);
```

Dragging a mouse will produce a similar kind of output shown next



Example Code 19.2: Ball Animation

The ball is continuously moving freely inside the corner of the frames. The sample outputs are shown below:

First we examine the MyPanel.java class that is drawing a filled oval.

```
import javax.swing.*;
import java.awt.*;
// extending class from JPanel
public class MyPanel extends JPanel {
// variables used to draw oval at different locations
int mX = 200;
int mY = 0;
// overriding paintComponent method
public void paintComponent(Graphics g){
// erasing behaviour - this will clear all the
// previous painting
super.paintComponent(g);
// Down casting Graphics object to Graphics2D
Graphics2D g2 = (Graphics2D)g;
// changing the color to blue
g2.setColor(Color.blue);
// drawing filled oval with blue color
// using instance variables
g2.fillOval(mX,mY,20,20);
}// end paintComponent
} // end MyPanel class
```

The Test class is given below. Additionally this class also contains the code for handling mouse events.

```
// importing required packages
 import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class AnimTest implements ActionListener {
```

```
JFrame f;
MyPanel p;
// used to control the direction of ball int x, y;
public AnimTest(){
f = new JFrame();
Container c = f.getContentPane();
c.setLayout(new BorderLayout());
x = 5;
y = 3;
p = new MyPanel();
c.add(p);
f.setSize(400,400);
f.setVisible(true);
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
// creating a Timer class object, used for firing
// one or more action events after a specified delay
// Timer class constructor requires time in
// milliseconds and object of class that handles
// action events
Timer t = new Timer (5, this);
// starts the timer, causing it to start sending
// action events to listeners
t.start();
} // end constructor
// event handler method
public void actionPerformed(ActionEvent ae){
// if ball reached to maximum width of frame minus
// 40 since diameter of ball is 40 then change the
// X-direction of ball
if (f.getWidth()-40 == p.mX)
x = -5;
// if ball reached to maximum height of frame
// minus 40 then change the Y-direction of ball
if (f.getHeight()-40 == p.mY)
y = -3;
// if ball reached to min. of width of frame,
// change the X-direction of ball
if (p.mX == 0 )
x = 5;
// if ball reached to min. of height of frame,
// change the Y-direction of ball
if (p.mY == 0 )
y = 3;
// Assign x,y direction to MyPanel's mX & mY
p.mX += x;
```

```
p.mY += y;
// call to repaint() method so that ball is drawn on
// new locations
p.repaint();
} // end actionPerformed() method
// main method
public static void main(String args[ ]){
AnimTest at = new AnimTest();
}
} // end AnimTest class
```

19.2 References

- Java, A Lab Course by Umair Javed
- Painting in AWT & Swing
  http://java.sun.com/products/jfc/tsc/articles/painting/index.html
- Performing Custom Painting
  http://java.sun.com/docs/books/tutorial/uiswing/14painting/index.html

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 20: Applets

20.1 Basic Definition

- A small program written in Java and included in a HTML page.
- It is independent of the operating system on which it runs
- An applet is a Panel that allows interaction with a Java program
- A applet is typically embedded in a Web page and can be run from a browser
- You need special HTML in the Web page to tell the browser about the applet
- For security reasons, applets run in a sandbox: they have no access to the client's file system

20.2 Applets Support

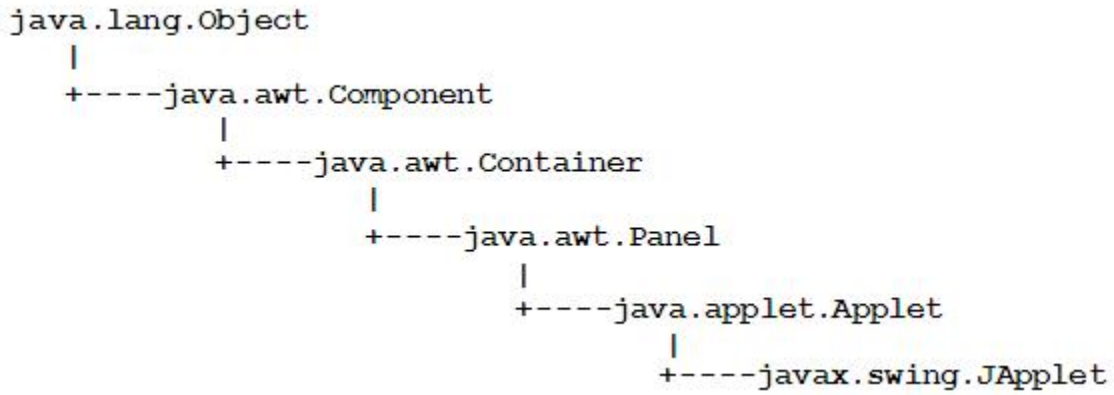
- Most modern browsers support Java 1.4 if they have the appropriate plugin
- Sun provides an application appletviewer to view applets without using browser.
- In general you should try to write applets that can be run with any browser

20.3 What an Applet is?

- You write an applet by extending the class Applet or JApplet
- Applet is just a class like any other; you can even use it in applications if you want
- When you write an applet, you are only writing part of a program
- The browser supplies the main method

20.4 The genealogy of Applet

The following figure shows the inheritance hierarchy of the JApplet class. This hierarchy determines much of what an applet can do and how, as you'll see on the next few pages.



Example Code 20.1: Writing a Simple Applet

Below is the source code for an applet called HelloApplet. This displays a *"Hello World"* string. Note that no main method has been provided.

```
// File HelloApplet.java

//step 1: importing required packages
import java.awt.*;
import javax.swing.*;
// extending class from JApplet so that our class also becomes an
//applet
public class HelloApplet extends JApplet {

// overriding paint method
public void paint(Graphics g) {
// write code here u want to display & draw by using
// Graphics object
g.drawString("Hello World", 30 , 30);
}
} // end class
```

After defining the HelloApplet.java, the next step is to write .html file. Below is the source code of Test.html file. The Test.html contains the ordinary html code except one.

```
<html>
<head>
<title> Simple Applet </title> </head>

<body>

<!-- providing the class name of applet with width &
height
-->
<applet code="HelloApplet.class"
width=150 height=100>
</applet>
</body>
</html>
```

Compile & Execute

By simply double clicking on Test.html file, you can view the applet in your browser. However, you can also use the appletviewer java program for executing or running applets.

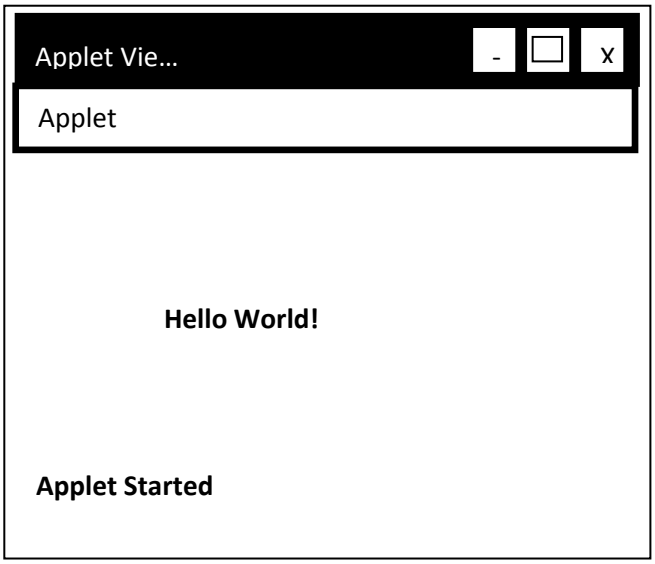
The applet viewer is invoked from the command line by the command

appletviewer *htmlfile*

where *htmlfile* is the name of the file that contains the html document. For our example, the command looks like this:

appletviewer *Test.html*

As a result, you will see the following output



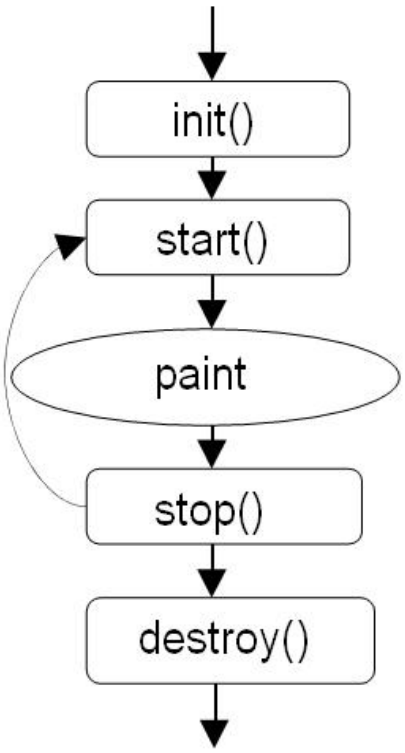
20.5 Applet Life Cycle Methods

When an applet is loaded, an instance of the applet's controlling class (an Applet subclass) is created. After that an applet passes through some stages or methods, each of them are build for specific purpose.

An applet can react to major events in the following ways:

- It can **initialize** itself.
- It can **start** running.
- It can **stop** running.
- It can perform a **final cleanup**, in preparation for being unloaded

The applet's life cycle methods are called in the specific order shown below. Not every applet needs to override every one of these methods.



Let's take a look on each method in detail and find out what they do:

20.5.1init( )

- Is called only once.
- The purpose of init( ) is to *initialize* the applet each time it's loaded (or reloaded).
- You can think of it as a constructor

20.5.2 start( )

- To start the applet's execution
- For example, when the applet's loaded or when the user revisits a page that contains the applet
- start( ) is also called whenever the browser is maximized

20.5.3 paint( )

- paint( ) is called for the first time when the applet becomes visible
- Whenever applet needs to be repainted, paint( ) is called again
- Do all your painting in paint( ), or in a method that is called from paint( )

20.5.4 stop( )

- To stop the applet's execution, such as when the user leaves the applet's page or quits the browser.
- stop( ) is also called whenever the browser is minimized

20.5.5 destroy( )

- Is called only once.
- To perform a final cleanup in preparation for unloading

Example Code 20.2: Understanding Applet Life Cycle Methods

The following code example helps you in understanding the calling sequence of applet's life cycle methods. These methods are only displaying debugging statements on the console.

```
// File AppletDemo.java
//step 1: importing required packages
import java.awt.*;
import javax.swing.*;
// extending class from JApplet so that our class also becomes an
//applet
public class AppletDemo extends JApplet {
// overriding init method
public void init ( ) {
System.out.println("init() called");
}
// overriding start method
public void start ( ){
System.out.println("start() called");
}
// overriding paint method
public void paint(Graphics g){
System.out.println("paint() called");
}
// overriding stop method
public void stop(){
System.out.println("stop() called");
}
// overriding destroy method
public void destroy(){
System.out.println("destroy() called");
}
} // end class
```

The *DemoTest.html* file is using this applet. The code snippet of it given below:

```
<html>
<head>
<title> Applet Life Cycle Methods </title> </head>
<body>
<!-- providing the class name of applet with width &
height
-->
<applet code="AppletDemo.class”
width=150 height=100>
</applet>
</body>
</html>
```

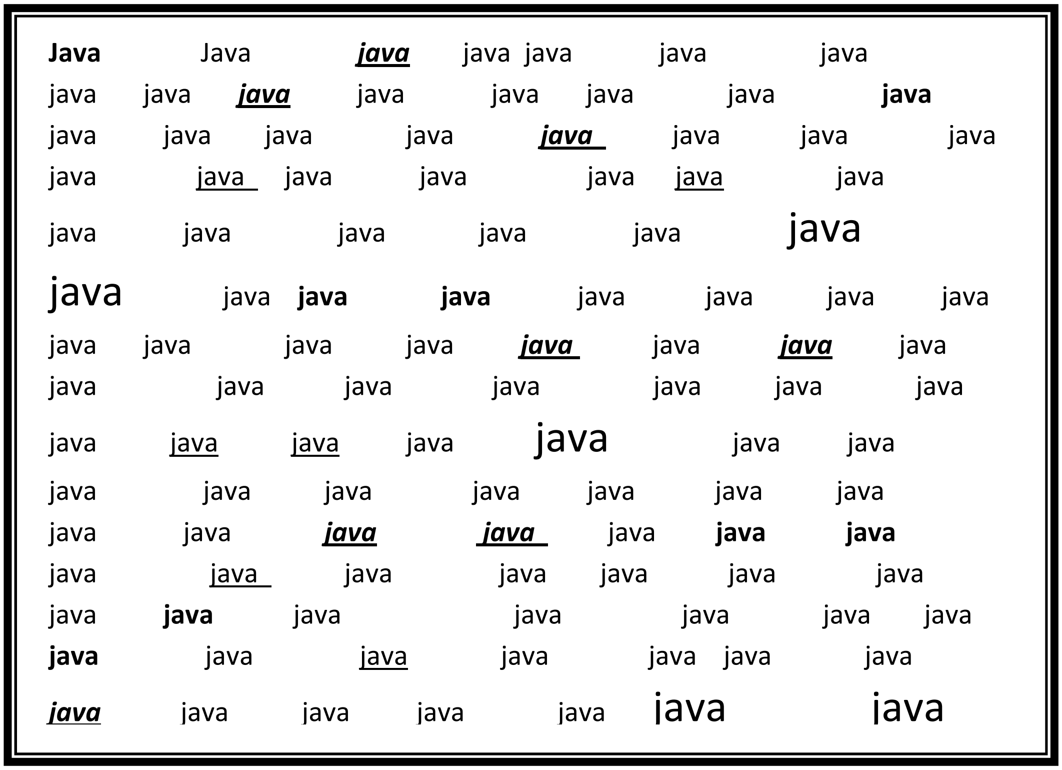
Compile & Execute

To understand the calling sequence of applet life cycle methods, you have to execute it by using appletviewer command. Do experiments like maximizing, minimizing the applet, bringing another window in front of applet and keep an eye on console output.

Example Code 20.3: Animated Java Word

Sample Output

The browser output of the program is given below:



Design Process

- The Program in a single call of paint method
    - Draws string “java” on 40 random locations
    - For every drawing, it selects random font out of 4 different fonts
    - For every drawing, it selects random color out of 256 * 256 * 256 RGB colors
- Repaint is called after every 1000 ms.
- After 10 calls to repaint, screen is cleared

Generating Random Numbers

- Use static method random of Math class
    - Math.random() ;
- Returns positive double value greater than or equal to 0.0 or less than 1.0.
-
- Multiply the number with appropriate scaling factor to increase the range and type cast it, if needed.
    - int i = (int)(Math.random() * 5); // will generate random numbers between 0 & 4.

Program’s Modules

The program is build using many custom methods. Let’s discuss each of them one by one that will help in understanding the overall logic of the program.

- **drawJava( )**

  As name indicates, this method will be used to write String "java" on random locations. The code is given below:

```
// method drawJava
public void drawJava(Graphics2D g2) {

// generate first number randomly. The panel width is 1000
int x = (int) (Math.random() * 1000);

// generate second number randomly. The panel height is 700
int y = (int) (Math.random() * 700);

// draw String on these randomly selected numbers
g2.drawString("java", x, y);
}
```

- **chooseColor( )**

  This method will choose color randomly out of 256 * 256 * 256 possible colors. The code snippet is given below:

```
// method chooseColor
public Color chooseColor() {
// choosing red color value randomly
 int r = (int) (Math.random() * 255);
// choosing green color value randomly
 int g = (int) (Math.random() * 255);
// choosing blue color value randomly
 int b = (int) (Math.random() * 255);
// constructing a color by providing R-G-B values
Color c = new Color(r, g, b);
// returning color
return c;
}
```

- **chooseFont( )**

  This method will choose a Font for text (java) to be displayed out of 4 available fonts. The code snippet is given below:

```
// method chooseFont
public Font chooseFont() {

// generating a random value that helps in choosing a font
```

```
int fontChoice = (int) (Math.random() * 4) + 1;
// declaring font reference
Font f = null;

// using switch based logic for selecting font
switch (fontChoice) {
case 1:
f = new Font("Serif", Font.BOLD + Font.ITALIC, 20);
break;
case 2:
f = new Font("SansSerif", Font.PLAIN, 17);
break;
case 3:
f = new Font("Monospaced", Font.ITALIC, 23);
break;
case 4:
f = new Font("Dialog", Font.ITALIC, 30);
break;
} // end switch
// returns Font object
return f;

} //end chooseFont
```

- **paint( )**

  The last method to be discussed here is paint(). By overriding this method, we will print string "java" on 40 random locations. For every drawing, it selects random font out of 4 different fonts & random color out of 256 * 256 * 256 RGB colors.

  Let's see, how it happens:

```
// overriding method paint
public void paint(Graphics g) {
// incrementing clear counter variable.
clearCounter++;
// printing 40 "java" strings on different locations by
// selcting random font & color
for (int i = 1; i <= 40; i++) {
// choosing random color by calling chooseColor() method
Color c = chooseColor();
// setting color
g2.setColor(c);
// choosing random Font by calling chooseColor() method
```

```
Font f = chooseFont();
g2.setFont(f);
// drawing string “java” by calling drawJava() method
drawJava(g2);
}
// end for loop
Graphics2D g2 = (Graphics2D) g;
// checking if paint is called 10 times then clears the
// screen and set counter again to zero
if (clearCounter == 10) {
g2.clearRect(0, 0, 1000, 700);
clearCounter = 0;
}
} // end paint method
```

Merging Pieces

By inserting all method inside *JavaAnim.java* class, the program will look like one given below. Notice that it contains methods discussed above with some extra code with which you are already familiar.

```
// File JavaAnim.java
//step 1: importing required packages import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class JavaAnim extends JApplet implements ActionListener {
// used to count how many times paint is called int clearCounter;
// declaring Timer reference
Timer t;
// overriding init method, used to initialize variables
public void init() {
setBackground(Color.black);
clearCounter = 0;
Timer t = new Timer(1000, this);
t.start();
}
// overriding paint method - discussed above
public void paint(Graphics g) {
clearCounter++;
Graphics2D g2 = (Graphics2D) g;
if (clearCounter == 10) {
g2.clearRect(0, 0, 1000, 700);
clearCounter = 0;
}
```

```
for (int i = 1; i <= 40; i++) {
Color c = chooseColor();
g2.setColor(c);
Font f = chooseFont();
g2.setFont(f);
drawJava(g2);
}
}
// overriding actionPerformed()of ActionListener interface
// called by Timer object
public void actionPerformed(ActionEvent ae) {
repaint();
}
// chooseColor method - discussed above
public Color chooseColor() {
int r = (int) (Math.random() * 255);
int g = (int) (Math.random() * 255);
int b = (int) (Math.random() * 255);
Color c = new Color(r, g, b);
return c;
} // chooseFont method - discussed above
public Font chooseFont() {
int fontChoice = (int) (Math.random() * 4) + 1;
Font f = null;
switch (fontChoice) {
case 1:
f = new Font("Serif", Font.BOLD + Font.ITALIC, 20);
break;
case 2:
f = new Font("SansSerif", Font.PLAIN, 17);
break;
case 3:
f = new Font("Monospaced", Font.ITALIC, 23);
break;
case 4:
f = new Font("Dialog", Font.ITALIC, 30);
break;
}
return f;
}
// drawJava() method - discussed above
public void drawJava(Graphics2D g2) {
int x = (int) (Math.random() * 1000);
int y = (int) (Math.random() * 700);
g2.drawString("java", x, y);
```

```
}
} // end class
```

The *AnimTest.html* file is using this applet. The code snippet of it given below:

```
<html>
<head>
<title> Animated Java Word </title> </head>
<body>
<applet code="JavaAnim.class" width=1000 height=700> </applet>
</body>
</html>
```

Compile & Execute

You can execute it directly using browser or by using appletviewerapplication. For having fun, you can use "your name" instead of "java" and watch it in different colors.

20.6 References:

- Java, A Lab Course by Umair Javed
- Writing Applets
  - o http://java.sun.com/docs/books/tutorial/**applet**/

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 21: Socket Programming

21.1 Basic Definition

- A socket is one endpoint of a two-way communication link between two programs running generally on a network.
- A socket is a bi-directional communication channel between hosts. A computer on a network often termed as host.

21.2 Socket Dynamics

- As you have already worked with files, you know that file is an abstraction of your hard drive. Similarly you can think of a socket as an abstraction of the network.
- Each end has input stream (to send data) and output stream (to receive data) wired up to the other host.
- You store and retrieve data through files from hard drive, without knowing the actual dynamics of the hard drive. Similarly you send and receive data to and from network through socket, without actually going into underlying mechanics.
- You read and write data from/to a file using streams. To read and write data to socket, you will also use streams.

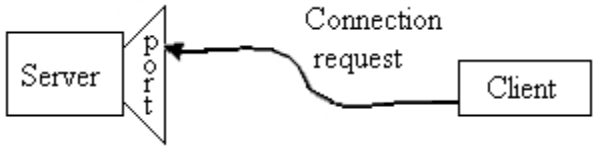
21.3 What is Port?

- It is a transport address to which processes can listen for connections request.
- There are different protocols available to communicate such as TCP and UDP. We will use TCP for programming in this handout
- There are 64k ports available for TCP sockets and 64k ports available for UDP, so at least theoretically we can open 128k simultaneous connections.
- There are well-known ports which are
    - o below 1024
    - o provides standard services
    - o Some well-known ports are:
        - FTP works on port 21
        - HTTP works on port 80
        - TELNET works on port 23 etc.

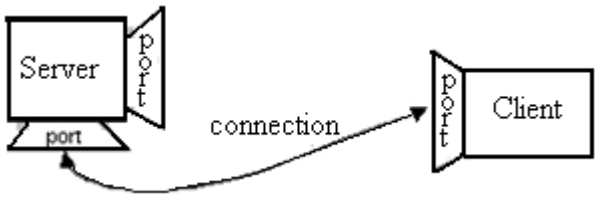
21.4 How Client - Server Communicate

- Normally, a server runs on a specific computer and has a socket that is bound to a specific port number.
- The server just waits, listening to the socket for a client to make a connection request.

- On the client side: The client knows the hostname of the machine on which the server is running and the port number to which the server is connected.



- As soon as client creates a socket that socket attempts to connect to the specified server.
- The server listens through a special kind of socket, which is named as server socket.
- The sole purpose of the server socket is to listen for incoming request; it is not used for communication.
- If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket, a communication socket, bound to a different port number.
- The server needs a new socket (and consequently a different port number) so that it can continue to listen through the original server socket for connection requests while tending to the needs of the connected client. This scheme is helpful when two or more clients try to connect to a server simultaneously (a very common scenario).



- On the server side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.
- Note that the socket on the client side is not bound to the port number used to make contact with the server. Rather, the client is assigned a port number local to the machine on which the client is running.
- The client and server can now communicate by writing to or reading from their sockets.

21.5 Steps - To Make a Simple Client

To make a client, process can be split into 5 steps. These are:

21.5.1 Import required package

You have to import two packages

- java.net.*;

- java.io.*;

21.5.2 Connect / Open a Socket with Server

Create a client socket (communication socket)

```
Socket s          = new Socket("serverName", serverPort) ;
```

- **serverName:** Name or address of the server you wanted to connect such as http://www.google.com or 172.2.4.98 etc. For testing if you are running client and server on the same machine then you can specify "localhost" as the name of server
- **serverPort :** Port number you want to connect to

The scheme is very similar to our home address and then phone number.

21.5.3 Get I/O Streams of Socket

Get input & output streams connected to your socket

- **For reading data from socket**

As stated above, a socket has input stream attached to it.

```
InputStream is = s.getInputStream();
// now to convert byte oriented stream into character oriented buffered reader
// we use intermediary stream that helps in achieving above stated purpose
InputStreamReader isr= new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);
```

- **For writing data to socket**

A socket has also output stream attached to it. Therefore,

```
OutputStream os = s.getOutputStream();
// now to convert byte oriented stream into character oriented print writer
// here we will not use any intermediary stream because PrintWriter constructor
// directly accepts an object of OutputStream
PrintWriter pw = new PrintWriter(os, true);
```

Here notice that true is also passed to so that output buffer will flush.

21.5.4 Send / Receive Message

Once you have the streams, sending or receiving messages isn't a big task. It's very much similar to

the way you did with files

- **To send messages**

```
pw.println("hello world");
```

- **To read messages**

```
String recMsg = br.readLine();
```

21.5.5 Close Socket

Don't forget to close the socket, when you finished your work

```
s.close();
```

21.6 Steps - To Make a Simple Server

To make a server, process can be split into 7 steps. Most of these are similar to steps used in making a client. These are:

21.6.1 Import required package

You need the similar set of packages you have used in making of client

- java.net.*;
- java.io.*;

21.6.2 Create a Server Socket

In order to create a server socket, you will need to specify port no eventually on which server will listen for client requests.

```
ServerSocket ss = new ServerSocket(serverPort) ;
```

- **serverPort:** port local to the server i.e. a free port on the server machine. This is the same port number that is given in the client socket constructor

21.6.3 Wait for Incoming Connections

The job of the server socket is to listen for the incoming connections. This listening part is done

through the accept method.

```
Socket s = ss.accept();
```

The server program blocks (stops) at the accept method and waits for the incoming client connection when a request for connection comes it opens a new communication socket (s) and use this socket to communicate with the client.

21.6.4 Get I/O Streams of Socket

Once you have the communication socket, getting I/O streams from communication socket is similar to the way did in making a client

- **For reading data from socket**

```
InputStream is = s.getInputStream();
InputStreamReader isr=new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);
```

- **For writing data to socket**

```
OutputStream os = s.getOutputStream();
PrintWriter pw = new PrintWriter(os, true);
```

21.6.5 Send / Receive Message

Sending and receiving messages is very similar as discussed in making of client

- **To send messages:**

```
pw.println("hello world");
```

- **To read messages**

```
String recMsg = br.readLine();
```

21.6.6 Close Socket

```
s.close();
```

Example Code 21.1: Echo Server & Echo Client

The client will send its name to the server and server will append "hello" with the name send by the client. After that, server will send back the name with appended "hello".

EchoServer.java

Let's first see the code for the server

```
// step 1: importing required package
import java.net.*;
import java.io.*;
```

```
import javax.swing.*;
public class EchoServer{
public static void main(String args[]){
try {
//step 2: create a server socket
ServerSocket ss = new ServerSocket(2222);
System.out.println("Server started...");
/* Loop back to the accept method of the server socket and wait for a new connection request. So
server will continuously listen for requests
*/
while(true) {
// step 3: wait for incoming connection
Socket s = ss.accept();
System.out.println("connection request recieved");

// step 4: Get I/O streams
InputStream is = s.getInputStream();
InputStreamReader isr= new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);

OutputStream os = s.getOutputStream();
PrintWriter pw = new PrintWriter(os,true);
// step 5: Send / Receive message
// reading name sent by client
String name = br.readLine();
// appending “hello” with the received name
String msg = "Hello " + name + " from Server";
// sending back to client
pw.println(msg);
// closing communication sockey
s.close();
} // end while
}catch(Exception ex){
System.out.println(ex);
}
}
} // end class
```

EchoClient.java

The code of the client is given below

```
// step 1: importing required package
import java.net.*;
import java.io.*;
import javax.swing.*;
public class EchoClient{
```

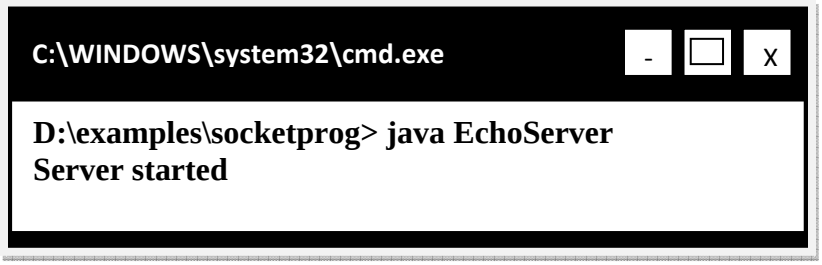
```
public static void main(String args[]){
try {
//step 2: create a communication socket

/* if your server will run on the same machine then you can pass “localhost” as server address.Notice
that port no is similar to one passed while creating server socket */
Socket s = new Socket(“localhost”, 2222);
// step 3: Get I/O streams
InputStream is = s.getInputStream();
InputStreamReader isr= new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);
OutputStream os = s.getOutputStream();
PrintWriter pw = new PrintWriter(os,true);

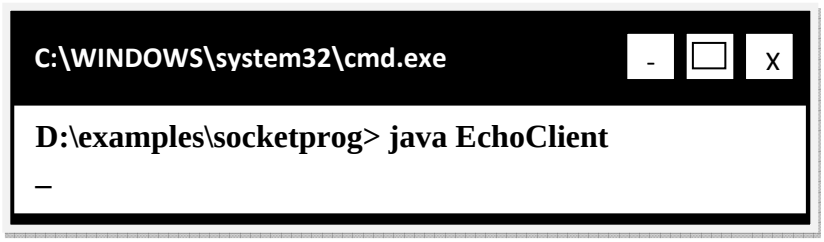
// step 4: Send / Receive message
// asking user to enter his/her name
String msg = JOptionPane.showInputDialog( "Enter your name");
// sending name to server
pw.println(msg);
// reading message (name appended with hello) from
// server
msg = br.readLine();
// displaying received message
JOptionPane.showMessageDialog(null , msg);
// closing communication socket
s.close();
}catch(Exception ex){
System.out.println(ex);
}
}
} // end class
```

Compile & Execute

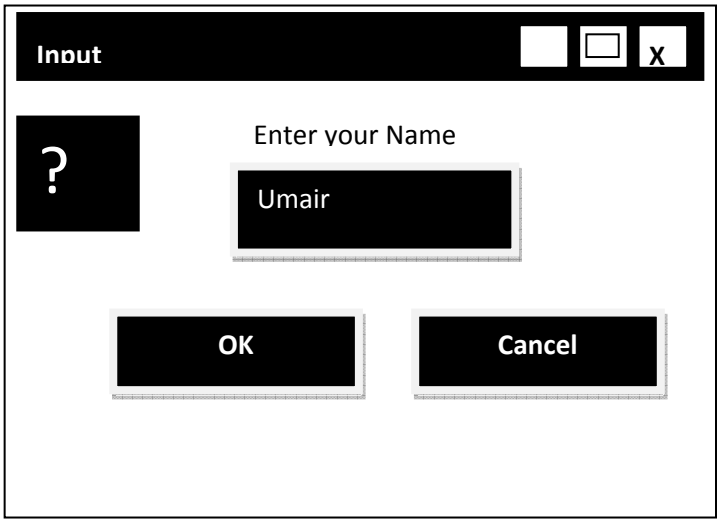
After compiling both files, run EchoServer.java first, from the command prompt window. You’ll see a message of “server started” as shown in the figure below. Also notice that cursor is continuously blinking since server is waiting for client request



Now, open another command prompt window and run EchoClient.java from it. Look at EchoServer window; you'll see the message of "request received". Sooner, the EchoClient program will ask you to enter name in input dialog box. After entering name press ok button, with in no time, a message dialog box will pop up containing your name with appended "hello" from server. This whole process is illustrated below in pictorial form:







Sending name to server



Response from server

Notice that server is still running, you can run again EchoClient.java as many times untill server is running.

To have more fun, run the server on a different computer and client on a different. But before doing that find the IP of the computer machine on which your EchoServer will eventually run. Replace "localhost" with the new IP and start conversion over network

21.7 References

- Entire material for this handout is taken from the book **JAVA A Lab Course** by **Umair Javed**. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose without the consent of author.

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 22:

22.1 Problem

22.1.1 What?

- You want to send an object to a stream.

22.1.2 Motivation

- A lot of code involves boring conversion from a file to memory
  - As you might recall that AddressBook program reads data from file and then parses it
- This is a common problem

22.1.3 Revisiting AddressBook

We read record from a text file named persons.txt. The person record was present in the file in the following format:

```
Ali,defence,9201211
Usman,gulberg,5173940
Salman,LUMS,5272670
```

persons.txt

The code that was used to construct Person objects after reading information from the file is given below. Here only the part of code is shown, for complete listing, see AddressBook code in your earlier handout.

```
……….
FileReader fr = new FileReader("persons.txt");
BufferedReader br = new BufferedReader(fr);
String line = br.readLine();

while ( line != null ) {
tokens = line.split(",");
name = tokens[0];
add  = tokens[1];
ph    = tokens[2];
PersonInfo p =new PersonInfo(name, add, ph);
// you can add p into arraylist, if needed
```

```
line = br.readLine();
}
```

As you have seen a lot of parsing code is required for converting a line into PersonInfo objects. Serialization mechanism eases developer's life by achieving all above in a very simple way.

22.2 Serialization in Java

- Java provides an extensive support for serialization
- Object knows how to read or write themselves to streams
- **Problem:**
    - As you know, objects get created on heap and have some values therefore Objects have some state in memory
    - You need to save and restore that state.
    - The good news is that java serialization takes care of it automatically

22.2.1 Serializable Interface

- By implementing this interface a class declares that it is willing to be read/written by automatic serialization machinery
- Found in java.io package
- Tagging interface - has no methods and serves only to identify the semantics of being serializable

22.2.2 Automatic Writing

- System knows how to recursively write out the state of an object to stream
- If an object has the reference of another object, the java serialization mechanism takes care of it and writes it too.

22.2.3 Automatic Reading

- System knows how to read the data from Stream and re-create object in memory
- The recreated object is of type "Object" therefore Down-casting is required to convert it into actual type.

22.2.4 Serialization: How it works?

- To write an object of PersonInfo, ObejctOutputStream and its method writeObject( ) will be used

```
PersonInfo p = new PersonInfo( );
ObejctOutputStream out;
// writing PersonInfo's object p
out.writeObject(p);
```

- To read that object back, ObejctInputStream and its method readObject() will be used

```
ObejctInputStream in;
// reading PersonInfo's object. Remember type
casting // is required
PersonInfo obj = (PersonInfo)in.readObject( );
```

Example Code 22.1: Reading / Writing PersonInfo objects

We want to send PersonInfo object to stream. You have already seen this class number of times before. Here it will also implement serializable interface.

PersonInfo.java

```
import javax.swing.*;
import java.io.* ;
class PersonInfo implements Serializable{
String name;
String address;
 String phoneNum;
//parameterized constructor
public PersonInfo(String n, String a, String p) {
name = n;
address = a;
 phoneNum = p;
}
//method for displaying person record on GUI
public void printPersonInfo( ) {
 JOptionPane.showMessageDialog(null ,"name: " + name + "address:" +address + "phone no:" +
phoneNum);
}
} // end class
```

WriteEx.java

The following class will serialize PersonInfo object to a file

```
import java.io.*;
public class WriteEx{
public static void main(String args[ ]){
PersonInfo pWrite = new PersonInfo("ali", "defence", "9201211");
try {
// attaching FileOutput stream with “ali.dat”
FileOutputStream fos = new FileOutputStream("ali.dat");
// attaching ObjectOutput stream over FileOutput stream
ObjectOutputStream out = new ObjectOutputStream(fos);
//serialization
// writing object to ‘ali.dat’
out.writeObject(pWrite);
// closing streams out.close();
fos.close();
} catch (Exception ex){ System.out.println(ex);
}
}
}      // end class
```

ReadEx.java

The following class will read serialized object of PersonInfo from file i.e “ali.dat”

```
import java.io.*;
public class ReadEx{
public static void main(String args[ ]){
try {
// attaching FileInput stream with “ali.dat”
FileInputStream fis = new FileInputStream("ali.dat");
// attaching FileInput stream over ObjectInput stream
ObjectInputStream in = new ObjectInputStream(fis);
//de-serialization
// reading object from ‘ali.dat’
PersonInfo pRead = (PersonInfo)in.readObject( );
// calling printPersonInfo method to confirm that
// object contains same set of values before
// serializatoion
pRead.printPersonInfo();
// closing streams
 in.close();
fis.close();
} catch (Exception ex){
System.out.println(ex);
}
}//end main function
} // end class
```

Compile & Execute

After compilation, first run the WriteEx.java file and visit the "ali.dat" file. Then run ReadEx.javafrom different command or same command prompt.

22.3 Object Serialization & Network

- You can read / write to a network using sockets.
- All you need to do is attach your stream with socket rather than file.
- The class version should be same on both sides (client & network) of the network .

Example Code 22.2: Sending/Reading Objects to/from Network

We are going to use same PersonInfo class listed in example code 22.1. An object of PersonInfo class will be sent by client on network using sockets and then be read by server from network.

Sending Objects over Network

The following class ClientWriteNetEx.javawill send an object on network

```
import java.net.*;
import java.io.*;
 import javax.swing.*;
public class ClientWriteNetEx{
public static void main(String args[]){
try {
PersonInfo p = new PersonInfo("ali", "defence", "9201211");
// create a communication socket
Socket s = new Socket("localhost", 2222);
// Get I/O streams
OutputStream is = s.getOutputStream();
// attaching ObjectOutput stream over Input stream
ObjectOutputStream oos= new ObjectOutputStream(is);
// writing object to network
oos.writeObject(p);
// closing communication socket
s.close();
}catch(Exception ex){
System.out.println(ex);
 }
```

```
}
} // end class
```

Reading Objects over Network

The following class ServerReadNetEx.javawill read an object of PersonInfo sent by client.

```
import java.net.*;
 import java.io.*;
import javax.swing.*;
public class ServerReadNetEx{
public static void main(String args[]){
try {
// create a server socket
ServerSocket ss = new ServerSocket(2222);
System.out.println("Server started...");
/* Loop back to the accept method of the server socket and wait for a new connection request. So
server will continuously listen for requests
*/
while(true) {
// wait for incoming connection
Socket s = ss.accept();
System.out.println("connection request recieved");
// Get I/O streams
InputStream is = s.getInputStream();
// attaching ObjectOutput stream over Input stream
ObjectInputStream ois = new ObjectInputStream(is);
// read PersonInfo object from network
PersonInfo p = (PersonInfo)ois.readObject( );
p.printPersonInfo();
// closing communication socket
s.close();
} // end while
}catch(Exception ex){
System.out.println(ex); }
}
} // end class
```

Compile & Execute

After compiling both files, run ServerReadNetEx.javafirst, from the command prompt window. Open another command prompt window and run ClientWriteNetEx.java from it. The ClientWriteNetEx.java will send an Object of PersonInfo to ServerReadNetEx.java that displays that object values in dialog box after reading it from network.

22.4 Preventing Serialization

- Often there is no need to serialize sockets, streams & DB connections etc because they do not represent the state of object, rather connections to external resources
- To do so, **transient** keyword is used to mark a field that should not be serialized
- So we can mark them as,
  - **transient** Socket s;
  - **transient** OutputStream os;
  - **transient** Connection con;
- Transient fields are returned as null on reading

Example Code 22. 3: transient

Assume that we do not want to serialize phoneNumattribute of PersonInfo class, this can be done as shown below

PersonInfo.java

```
import javax.swing.*;
import java.io.*
class PersonInfo implements Serializable{
String name;
String address;
transient String phoneNum;
public PresonInfo(String n, String a, String p) {
name = n;
address = a;
phoneNm = p;
}
public void printPersonInfo( ) {
JOptionPane.showMessageDialog(null , “name: ” + name + “address:” +address + “phone no:” +
phoneNum);
}
} // end class
```

22.5 References

Entire material for this handout is taken from the book **JAVA A Lab Course** by **Umair Javed**. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose without the consent of author.

Note: Coding exercises in working condition for this lecture are also available on **“Downloads”** section of LMS.

Lecture 23: Multithreading

23.1 Introduction

Multithreading is the ability to do multiple things at once within the same application. It provides finer granularity of concurrency. A *thread* — sometimes called an *execution context* or a *lightweight process* — is a single sequential flow of control within a program.

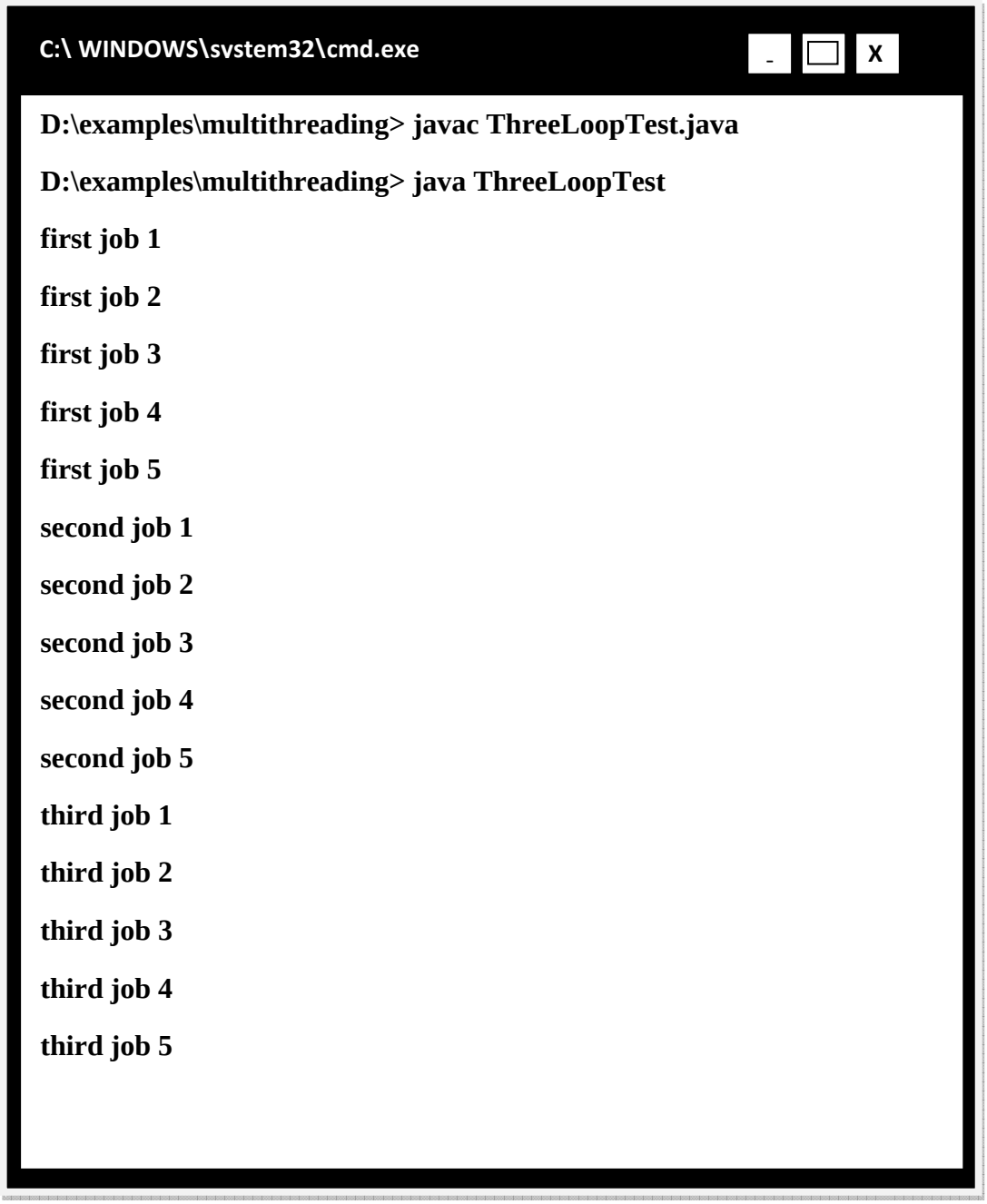
Threads are light weight as compared to processes because they take fewer resources then a process. A *thread* is easy to create and destroy. Threads share the same address space i.e. multiple threads can share the memory variables directly, and therefore may require more complex synchronization logic to avoid *deadlocks* and *starvation*.

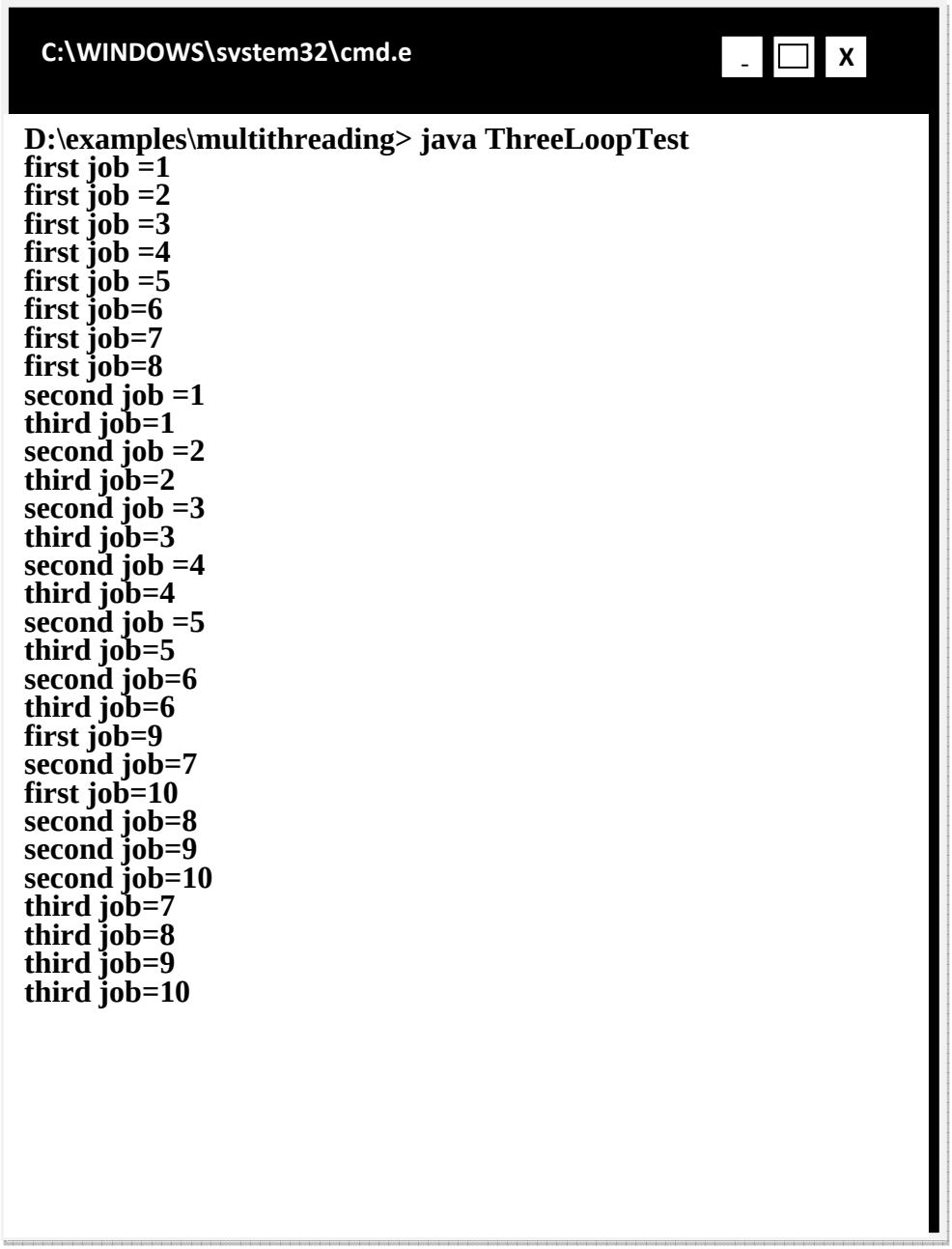
23.2 Sequential Execution vs. Multithreading

Every program has atleast one thread. Programs without multithreading executes sequentially. That is, after executing one instruction the next instruction in sequence is executed. If a function is called then until the completion of the function the next instruction is not executed. Similarly if there is a loop then instructions after loop only gets executed when the loop gets completed. Consider the following java program having three loops in it.

```
// File ThreeLoopTest.java
public class ThreeLoopTest {
public static void main (String args[ ]) {
//first loop
for (int i=1; i<= 5; i++)
System.out.println("first " +i);
// second loop
for (int j=1; j<= 5; j++)
System.out.println("second " + j);
// third loop
for (int k=1; k<= 5; k++)
System.out.println("third " + k);
} // end main
} // end class
```

When the program executes, the loops are executed sequentially, one after the other. It generates the following output.


C:\ WINDOWS\svstem32\cmd.exe
_
X
D:\examples\multithreading> javac ThreeLoopTest.java
D:\examples\multithreading> java ThreeLoopTest
first job 1
first job 2
first job 3
first job 4
first job 5
second job 1
second job 2
second job 3
second job 4
second job 5
third job 1
third job 2
third job 3
third job 4
third job 5



Note: Each loop has 5 iterations in the *ThreeLoopTest* program.

However, if we use multithreading — with one *thread* per loop — the program may generate the following output.

Note: Each loop has 10 iterations in the *ThreadTest* program. Your output can be different from the one given above.

Notice the difference between the outputs of the two programs. In *ThreeLoopTest* each loop generated a sequential output while in *ThreadTest* the output of the loops got intermingled i.e. concurrency took place and loops executed simultaneously.

Let us code our first multithreaded program and try to learn how Java supports multithreading.

23.3 Java Threads

Java includes built-in support for threading. While other languages have threads bolted on to an existing structure i.e. threads were not the part of the original language but latter came into existence as the need arose.

All well known operating systems these days support multithreading. JVM transparently maps *Java Threads* to their counter-parts in the operating system i.e. *OS Threads*. JVM allows threads in Java to take advantage of hardware and operating system level advancements. It keeps track of threads and schedules them to get CPU time. Scheduling may be pre-emptive or cooperative. So it is the job of JVM to manage different tasks of thread. Let's see how we can create threads?

23.3.1 Creating Threads in Java

There are two approaches to create threads in Java.

- Using Interface
- Using Inheritance

Following are the steps to create threads by using *Interface*:

- Create a class where you want to put some code that can run in parallel with some other code and let that class implement the *Runnable* interface.
- Runnable interface has the run() method therefore provide the implementation for the *run()* method and put your code that you want to run in parallel here.
- Instantiate *Thread* class object by passing *Runnable* object in constructor
- Start thread by calling *start()* method

Following are the steps to create threads by using **Inheritance**:

- Inherit a class from java.lang.Thread class
- Override the run() method in the subclass
- Instantiate the object of the subclass
- Start thread by calling start() method

23.3.1.1 Threads Creation Steps Using Interface

To write a multithreaded program using *Runnable* interface, follow these steps:

- **Step 1** - Implement the Runnable Interface
  class Worker implements Runnable

- **Step 2** - Provide an Implementation of run() method
  public void run( ){
  // write thread behavior
  // code that will be executed by the thread

- **Step 3** - Instantiate Thread class object by passing Runnable object in the constructor
  Worker w = new Worker ("first"); Thread t =
  new Thread (w);

- **Step 4** - Start thread by calling start() method
  t.start();

23.3.1.2 Threads Creation Steps Using Inheritance

To write a multithreaded program using inheritance from Thread class, follow these steps:

- **Step 1** - Inherit from Thread Class
  class Worker extends Thread

- **Step 2** - Override run() method
  public void run( ){
  // write thread behavior
  // code that will execute by thread

- **Step 3** - Instantiate subclass object
  Worker w = new Worker("first");

- **Step 4** - Start thread by calling start() method
  w.start();

23.4 Three Loops: Multi-Threaded Execution

So far we have explored:

- What is multithreading?
- What are Java Threads?
- Two ways to write multithreaded Java programs

Now we will re-write the *ThreeLoopTest* program by using *Java Threads*. At first we will use the *Interface* approach and then we will use *Inheritance*.

Code Example using Interface

```
// File Worker.java
public class Worker implements Runnable {
private String job ;
//Constructor of Worker class
 public Worker (String j ){
job = j;
}
//Implement run() method of Runnable interface
public void run ( ) {
for(int i=1; i<= 10; i++)
System.out.println(job + " = " + i);
}
} // end class
```

```
// File ThreadTest.java
public class ThreadTest{
public static void main (String args[ ]){
//instantiate three objects
Worker first = new Worker (“first job”);
Worker second = new Worker (“second job”);
Worker third = new Worker (“third job”);
//create three objects of Thread class & passing worker
//(runnable) to them
Thread t1 = new Thread (first );
Thread t2 = new Thread (second);
Thread t3 = new Thread (third);
//start threads to execute
t1.start();
t2.start();
 t3.start();
}//end main
```

```
} // end class
```

Code Example using Inheritance

Following code is similar to the code given above, but uses *Inheritance* instead of interface:

```
// File Worker.java
public class Worker extends Thread{
private String job ;
//Constructor of Worker class
 public Worker (String j ){
job = j;
}
//Override run() method of Thread class
public void run ( ) {
for(int i=1; i<= 10; i++)
System.out.println(job + " = " + i);
}
} // end class
```

// File ThreadTest.java

```
public class ThreadTest{
public static void main (String args[ ]) {
//instantiate three objects of Worker (Worker class is now
//becomes a Thread because it is inheriting from it)class

Worker first = new Worker ("first job");
Worker second = new Worker ("second job");
Worker third = new Worker ("third job");
//start threads to execute
first.start();
second.start();
third.start();
}//end main
} // end class
```

23.5 Thread Priorities

Threads provide a way to write concurrent programs. But on a single CPU, all the threads do not run simultaneously. JVM assigns threads to the CPU based on thread priorities. Threads with higher priority are executed in preference to threads with lower priority. A thread's default priority is same

as that of the creating thread i.e. parent thread.

A Thread's priority can be any integer between 1 and 10. We can also use the following predefined constants to assign priorities.

Thread.MAX_PRIORITY (typically 10)
Thread.NORM_PRIORITY (typically 5)
Thread.MIN_PRIORITY (typically 1)

To change the priority of a thread, we can use the following method

setPriority(int priority)

It changes the priority of this thread to integer value that is passed. It throws an IllegalArgumentExceptionif the priority is not in the range MIN_PRIORITY to MAX_PRIORITY i.e. (1-10).

For example, we can write the following code to change a thread's priority.

```
Thread t = new Thread (RunnableObject);
// by using predefined constant
t.setPriority (Thread.MAX_PRIORITY);
// by using integer constant
t.setPriority (7);
```

23.5.1 Thread Priority Scheduling

The Java runtime environment supports a very simple, deterministic scheduling algorithm called *fixed-priority scheduling*. This algorithm schedules threads on the basis of their priority relative to other *Runnable* threads.

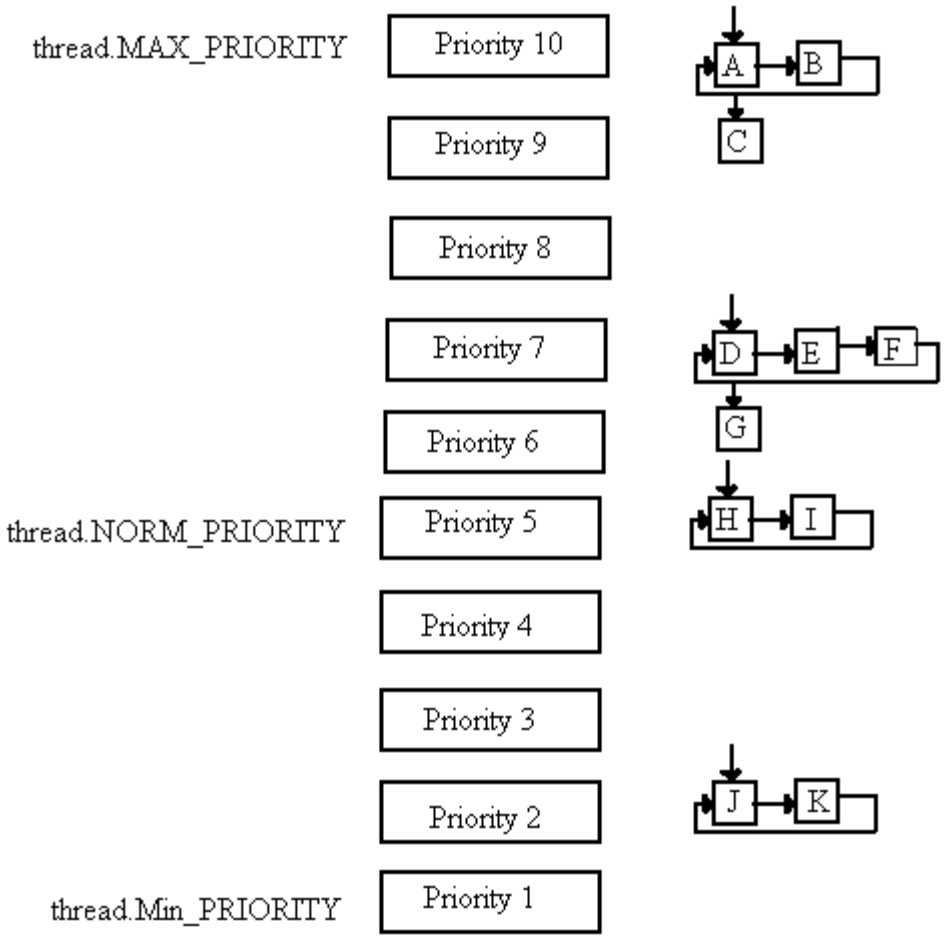
At any given time, when multiple threads are ready to be executed, the runtime system chooses for execution the Runnable thread that has the highest priority. Only when that thread *stops*, *yields* (will be explained later), or becomes *Not Runnable* will a lowerpriority thread start executing. If two threads of the same priority are waiting for the CPU, the scheduler arbitrarily chooses one of them to run. The chosen thread runs until one of the following conditions becomes true:

- A higher priority thread becomes *Runnable*.
- It yields, or its run()method exits.
- On systems that support time-slicing, its time allotment has expired.

Then the second thread is given a chance to run, and so on, until the interpreter exits.

Consider the following figure in which threads of various priorities are represented by capital

alphabets A, B, …, K. A and B have same priority (highest in this case). J and K have same priority (lowest in this case). JVM start executing with A and B, and divides CPU time between these two threads arbitrarily. When both A and B comes to an end, it chooses the next thread C to execute.



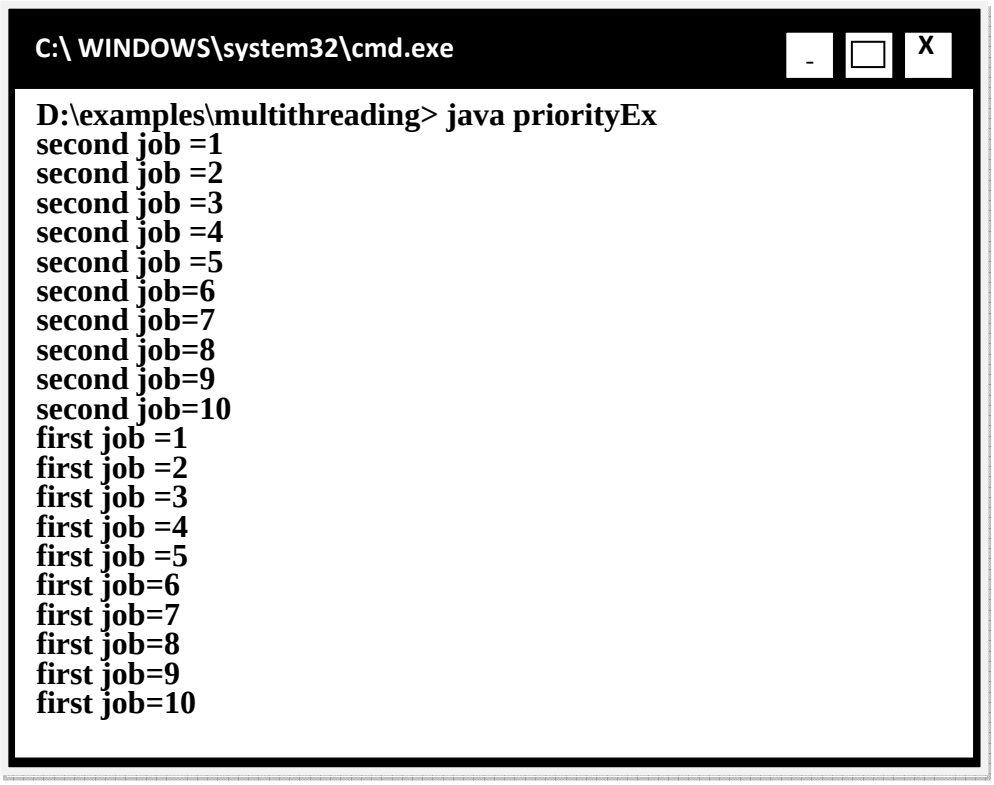
Code Example: Thread Priorities

Try following example to understand how JVM executes threads based on their priorities.

```
// File PriorityEx.java
public class PriorityEx{
public static void main (String args[ ]){
//instantiate two objects
Worker first = new Worker (“first job”);
Worker second = new Worker (“second job”);
//create two objects
Thread t1 = new Thread (first );
Thread t2 = new Thread (second);
//set thread priorities
```

```
t1.setPriority (Thread.MIN_PRIORITY);
t2.setPriority (Thread.MAX_PRIORITY);
//start threads to execute
t1.start();
t2.start();
}//end main
} // end class
```

Output



23.5.2 Problems with Thread Priorities

However, when using priorities with *Java Threads*, remember the following two issues:

First a Java thread priority may map differently to the thread priorities of the underlying OS. It is because of difference in priority levels of JVM and underlying OS.
For example

- Solaris has 2^{32}-1 priority levels
- Windows NT has only 7 user priority levels

Second, starvation can occur for lower-priority threads if the higher-priority threads never terminate, sleep, or wait for I/O indefinitely.

23.6 References:

- Java, A Practical Guide by Umair Javed.
- Java How to Program by Deitel and Deitel.
- CS193j handouts on Stanford.

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 24: More on Multithreading

In this handout, we'll cover different aspects of multithreading. Some examples are given to make you understand the topic of multithreading. First we will start with an example that reads data from two text files simultaneously.

Example Code: Reading Two Files Simultaneously

The task is to read data from file "first.txt" & "second.txt" simultaneously. Suppose that files contain the following data as shown below:

```
first 1
first 2
first 3
first 4
first 5
first 6
first 7
first 8
first 9
first 10
```

first.txt

```
second 1
second 2
second 3
second 4
second 5
second 6
second 7
second 8
second 9
second 10
```

second.txt

Following is the code of ReadFile.java that implements Runnable interface. The file reading code will be written inside run ( )method

// File ReadFile.java

```
import java.io.*;
public class ReadFile implements Runnable{
//attribute used for name of file
String fileName;
// param constructor
public ReadFile(String fn){
fileName = fn;
}
// overriding run method
// this method contains the code for file reading
public void run ( ){
try
{
// connecting FileReader with attribute fileName
FileReader fr = new FileReader(fileName);
BufferedReader br = new BufferedReader(fr);

String line = "";
// reading line by line data from file
// and displaying it on console
line = br.readLine();
while(line != null) {
System.out.println(line);
line = br.readLine();
}
fr.close();
br.close();
}
catch (Exception e){
System.out.println(e);
}
} // end run() method
}
```

Next, look at the Test.javaclass that contains the main() method.

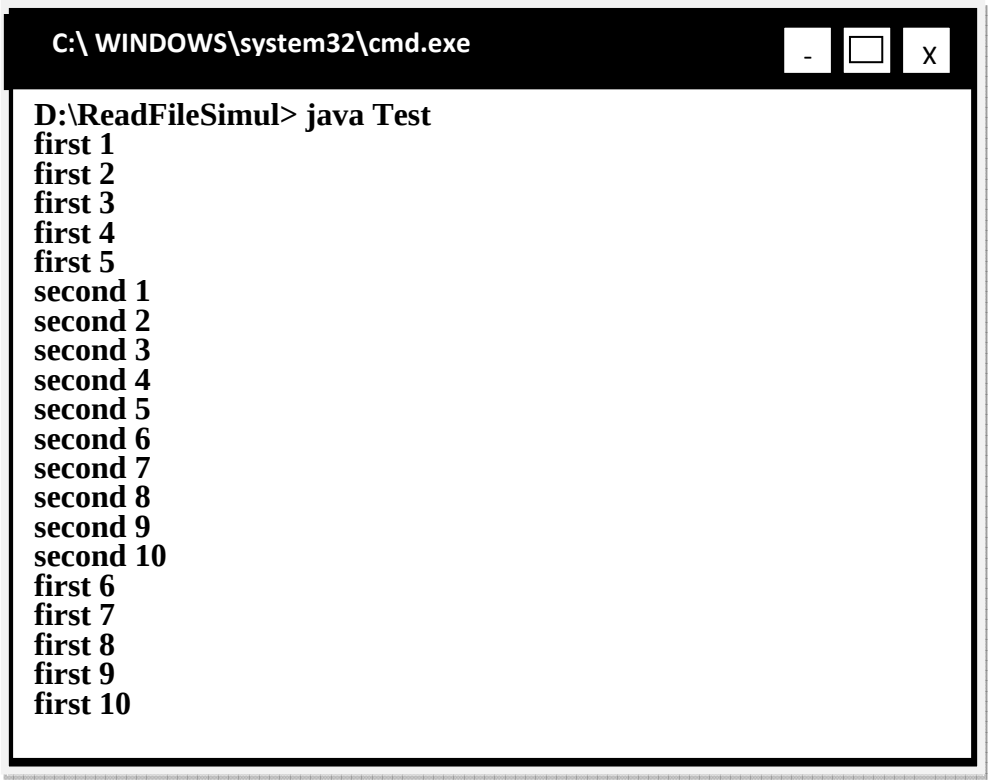
```
// File Test.java
 public class Test {
public static void main (String args[]){

// creating ReadFile objects by passing file names to them
ReadFile first = new ReadFile("first.txt");
```

```
ReadFile second = new ReadFile("second.txt");
// Instantiating thread objects and passing
// runnable (ReadFile) objects to them
Thread t1 = new Thread(first);
Thread t2 = new Thread(second);
// starting threads that cause threads to read data from
// two different files simultaneously
 t1.start();
t2.start();
}
}
```

Output

On executing Test class, following kind output would be generated:



24.1 Useful Thread Methods

Now let's discuss some useful thread class methods.

24.1.1sleep(int time) method

- Causes the currently executing thread to wait for the time (milliseconds) specified
- Waiting is efficient equivalent to non-busy. The waiting thread will not occupy the processor
- Threads come out of the sleep when the specified time interval expires or when interrupted by some other thread
- Thread coming out of sleep may go to the running or ready state depending upon the availability of the processor. The different states of threads will be discussed later
- High priority threads should execute sleep method after some time to give low priority threads a chance to run otherwise starvation may occur
- sleep() method can be used for delay purpose i.e. anyone can call Thread.sleep()method
- Note that sleep() method can throw InterruptedException. So, you need try-catch block

Example Code: Demonstrating sleep ( ) usage

Below the modified code of Worker.java is given that we used in the previous handout.

```
// File Worker.java
public class Worker implements Runnable {
private String job ;
//Constructor of Worker class
public Worker (String j ){
job = j;
}
//Implement run() method of Runnable interface
public void run ( ) {
for(int i=1; i<= 10; i++) {
try {
Thread.sleep(100);
// go to sleep for 100 ms
}catch (Exception ex){
System.out.println(ex);
}
System.out.println(job + " = " + i);
} // end for
} // end run
} // end class
```

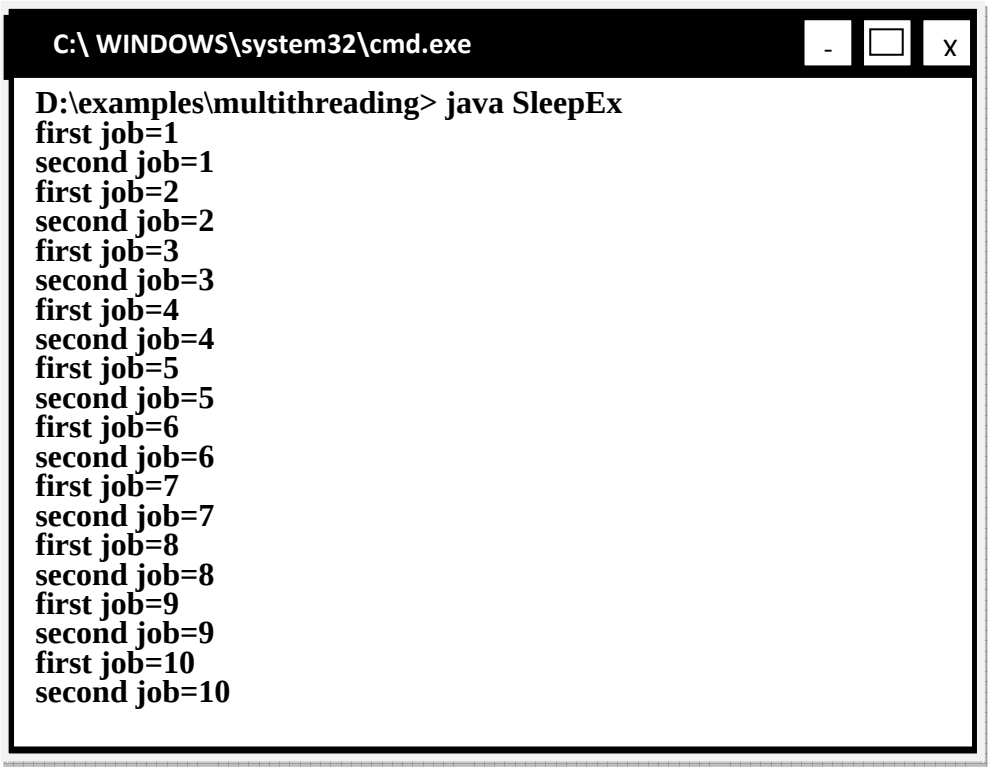
Below is the code of SleepEx.java that contains the main() method. It will use the Worker class created above.

```
// File SleepEx.java
```

```
public class SleepEx {
public static void main (String args[ ]){
// Creating Worker objects
Worker first= new Worker ("first job");
Worker second = new Worker ("second job");
// Instantiating thread class objects
Thread t1 = new Thread (first );
Thread t2 = new Thread (second);
// starting thread
t1.start();
t2.start();
}
} // end class
```

Output

On executing SleepEx.java, the output will be produced with exact alternations between first thread & second thread. On starting threads, first thread will go to sleep for 100 ms. It gives a chance to second thread to execute. Later this thread will also go to sleep for 100 ms. In the mean time the first thread will come out of sleep and got a chance on processor. It will print job on console and again enters into sleep state and this cycle goes on until both threads finished the run()method.



Example Code: Using sleep ( ) for delay purpose

Before jumping on to example code, lets reveal another aspect about main()method. When you run a Java program, the VM creates a new thread and then sends the main(String[] args)message to the class to be run! Therefore, there is always at least one running

thread in existence. However, we can create more threads which can run concurrently with the existing default thread.

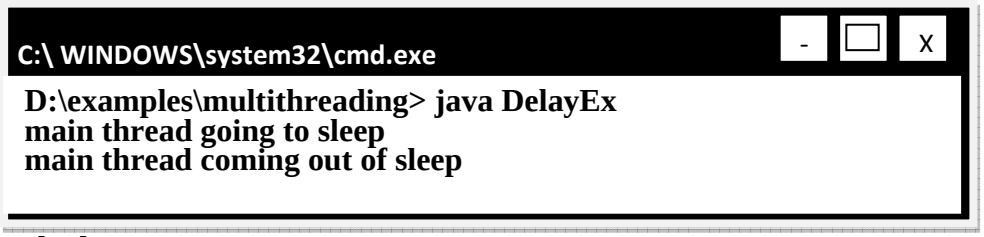
sleep()method can be used for delay purpose. This is demonstrated in the DelayEx.java given below:

```
// File DelayEx.java
public class DelayEx {
public static void main (String args[ ]){
System.out.println("main thread going to sleep");
try {
// the main thread will go to sleep causing delay
Thread.sleep(100);
}catch (Exception ex){
System.out.println(ex);
}
System.out.println("main thread coming out of sleep");
}
// end main()
} // end class
```

Output

On executing DelayEx class, you will experience a delay after the first statement displayed. The second statement will print when the time interval expired. This has been show below in the following two diagrams:





24.1.2yield( ) method

- Allows any other threads of the same priority to execute (moves itself to the end of the priority queue)
- If all waiting threads have a lower priority, then the yielding thread resumes execution on the CPU
- Generally used in cooperative scheduling schemes

Example Code: Demonstrating yield ( ) usage

Below the modified code of Worker.javais given

```
// File Worker.java
public class Worker implements Runnable {
private String job ;
//Constructor of Worker class
public Worker (String j ){
job = j;
}
//Implement run() method of Runnable interface
public void run ( )
{
for(int i=1; i<= 10; i++) {
// giving chance to a thread to execute of same priority


Thread.yield( );
System.out.println(job + " = " + i);
} // end for
} // end run
} // end class
```

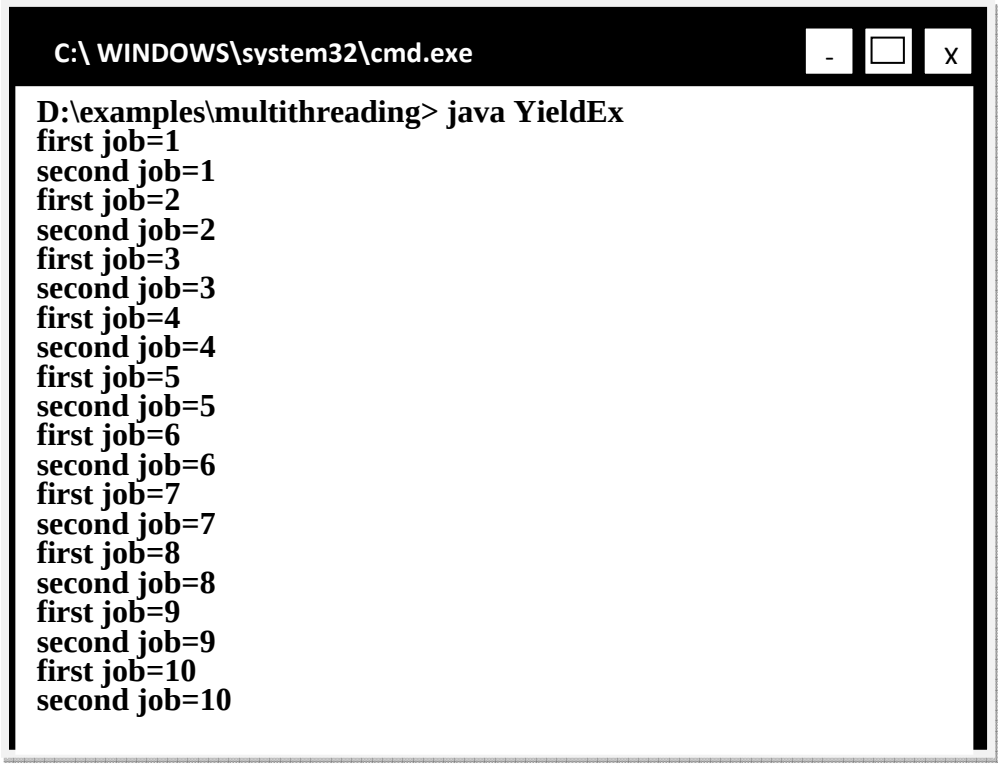
Below is the code of YieldEx.java that contains the main ()method.It will use the Worker class created above.

```
// File YieldEx.java public class YieldEx {
public static void main (String args[ ]){
// Creating Worker objects
Worker first = new Worker (“first job”);
Worker second = new Worker (“second job”);
// Instantiating thread class objects
Thread t1 = new Thread (first );
Thread t2 = new Thread (second);
// starting thread
t1.start();
t2.start();
}
} // end class
```

Output

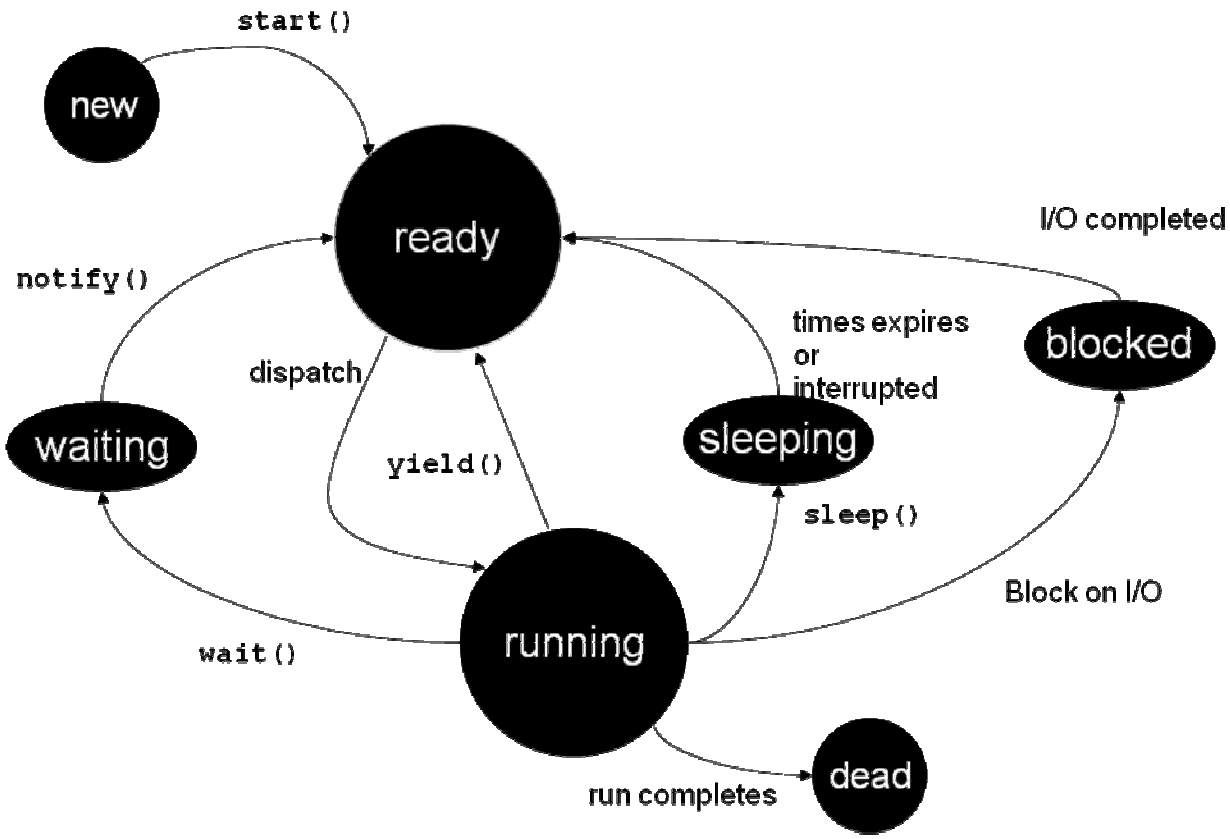
Since both threads have the same priority (until we change the priority of some thread explicitly).

Therefore both threads will execute on alternate basis. This can be confirmed from the output given below:



24.2 Thread States: Life Cycle of a Thread

A thread can be in different states during its lifecycle as shown in the figure.



Some Important states are:

24.2.1New state

- When a thread is just created

24.2.2 Ready state

- Thread's start() method invoked
- Thread can now execute
- Put it into the Ready Queue of the scheduler

24.2.3 Running state

- Thread is assigned a processor and now is running

24.2.4 Dead state

- Thread has completed or exited
- Eventually disposed of by system

24.3 Thread's Joining

- Used when a thread wants to wait for another thread to complete its run() method
- For example, if thread2 sent the thread2.join() message, it causes the currently executing thread to block efficiently until thread2 finishes its run() method
- Calling join method can throw InterruptedException, so you must use try-catch block to handle it

Example Code: Demonstrating join( ) usage

Below the modified code of Worker.java is given. It only prints the job of the worker

```
// File Worker.java
public class Worker implements Runnable {
private String job ;
public Worker (String j ){
job = j;
}
public void run ( ) {
for(int i=1; i<= 10; i++) {
System.out.println(job + " = " + i); } // end for
} // end run
} // end class
```

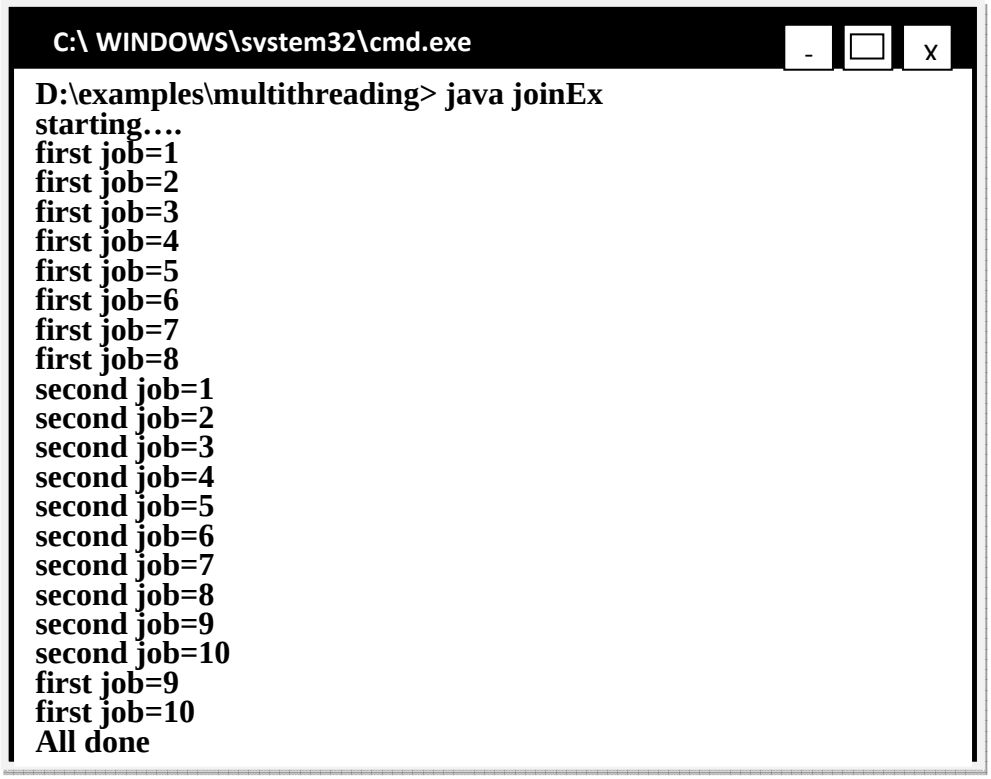
The class JoinEx will demonstrate how current running (main) blocks until the remaining threads finished their run ()

```
// File JoinEx.java
public class JoinEx {
public static void main (String args[ ]){
 Worker first = new Worker ("first job");
Worker second = new Worker ("second job");
Thread t1 = new Thread (first );
Thread t2 = new Thread (second);
System.out.println("Starting...");
// starting threads
 t1.start();
 t2.start();
// The current running thread (main) blocks until both
//workers have finished
```

```
try {
 t1.join();
 t2.join();
}
catch (Exception ex) {
System.out.println(ex);
}
System.out.println("All done ");
} // end main
}
```

Output

On executing JoinEx, notice that *"Starting"* is printed first followed by printing workers jobs. Since main thread does not finish until both threads have finished their run (). Therefore *"All done"* will be print on last.



24.4 References:

- Java, A Practical Guide by Umair Javed
- Java tutorial by Sun: http://java.sun.com/docs/books/tutorial/
- CS193j handouts on Stanford

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

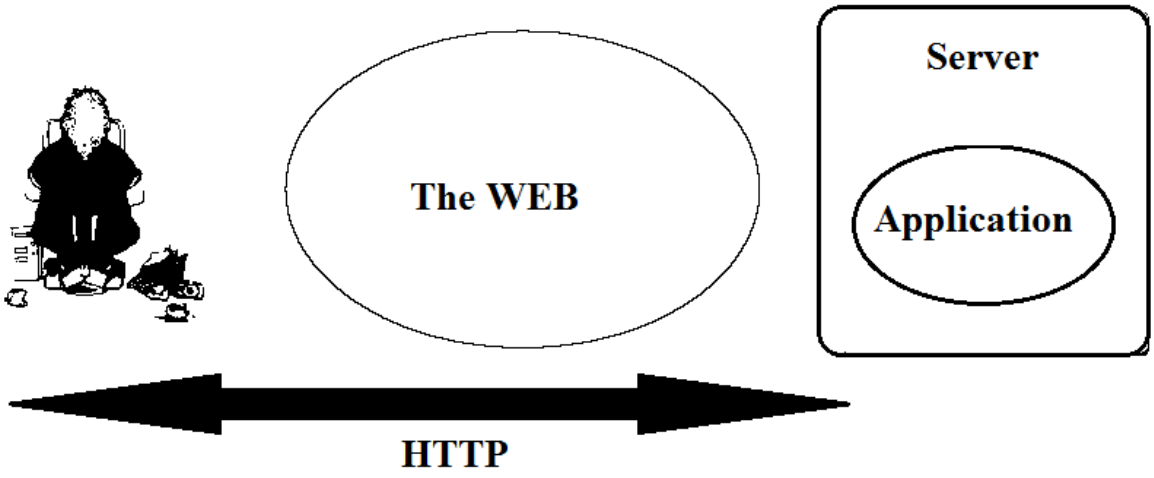
Lecture 25: Web Application Development

25.1 Introduction

Because of the wide spread use of internet, web based applications are becoming vital part of IT infrastructure of large organizations. For example web based employee performance management systems are used by organizations for weekly or monthly reviews of employees. On the other hand online course registration and examination systems can allow students to study while staying at their homes.

25.2 Web Applications

In general a web application is a piece of code running at the server which facilitates a remote user connected to web server through HTTP protocol. HTTP protocol follows *stateless Request-Response communication model*. Client (usually a web-browser) sends a request to Server, which sends back appropriate response or error message.



A Typical Web Application

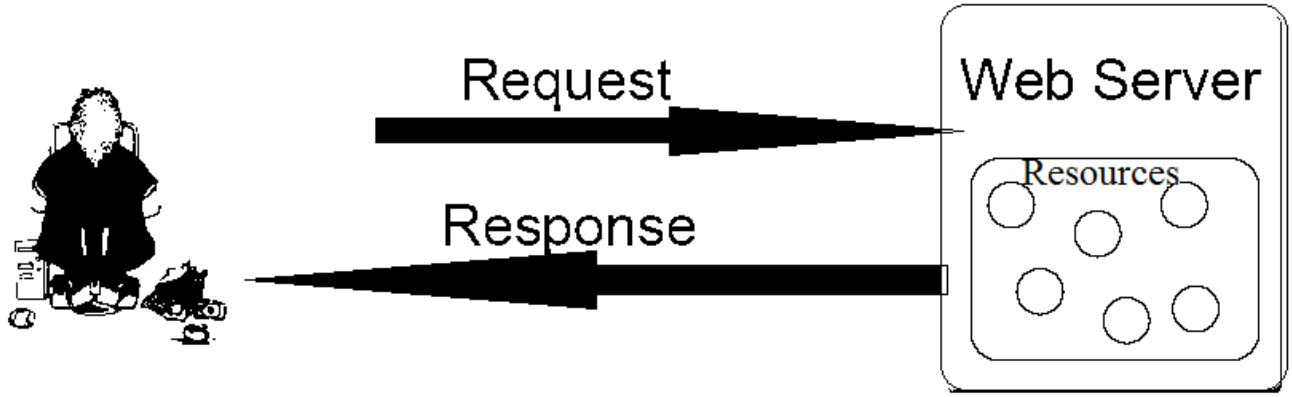
A web server is software which provides users, access to the services that are present on the internet. These servers can provide support for many protocols used over internet or intranet like HTTP, FTP, telnet etc

25.3 HTTP Basics

A protocol defines the method and way of communication between two parties. For example when we talk to our teacher we use a certain way which is different from the way that we adopt with our friends or parents. Similarly there are many different protocols used by computers to communicate with each other depending on applications.

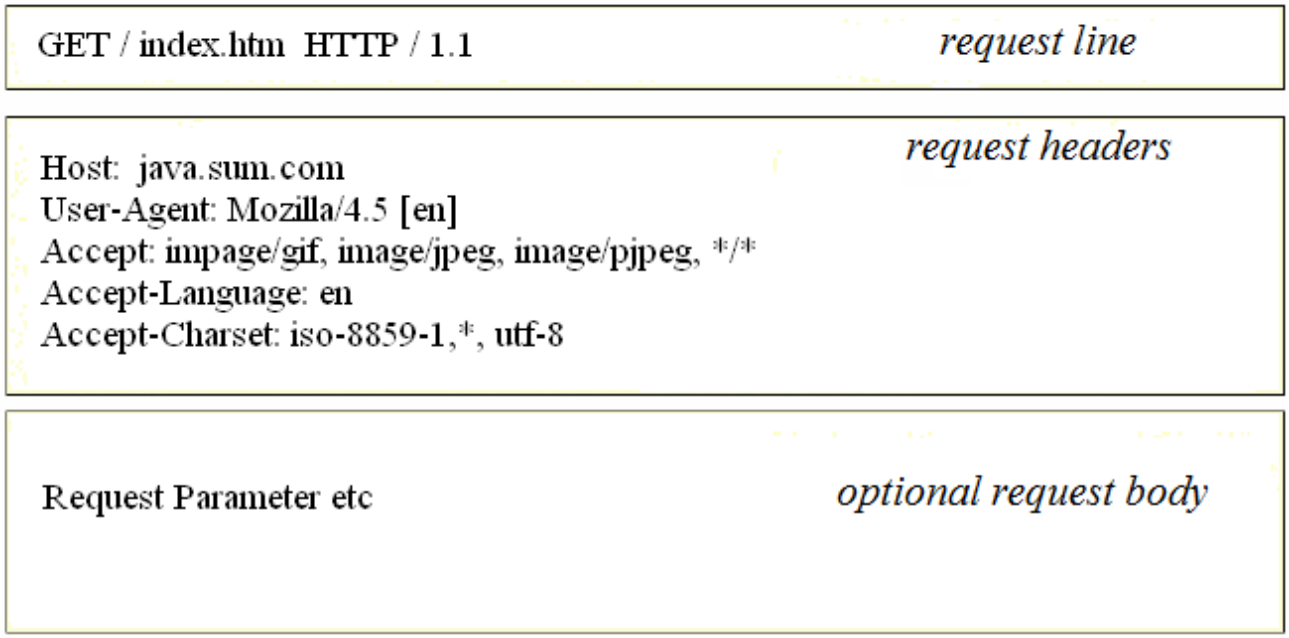
For example an Echo Server only listens to incoming name messages and sends back hello message, while HTTP protocol uses various types of request-response messages.

HTTP Communication Model



25.3.1 Parts of an HTTP request

- **Request Method:** It tells the server the type of action that a client wants to perform
- **URI:** Uniform Resource Indicator specifies the address of required document or resource
- **Header Fields:** Optional headers can be used by client to tell server extra information about request e.g. client software and content type that it understands.



HTTP Request Example

- **Body:** Contains data sent by client to the server
- Other request headers like FROM (email of the person responsible for request) and VIA (used by gateways and proxies to show intermediate sites the request passes) can also be used.
- **Request Parameters**
    - Request can also contain addition information in form of request parameters
        - In URL as query string e.g.
        - http://www.gmail.com/register**?name=ali&state=punjab**
        - As part of request body (see Figure 3)

25.3.2 Parts of HTTP response

- **Result Code:** A numeric status code and its description.
- **Header Fields:** Servers use these fields to tell client about server information like configurations and software etc.
- **Body:** Data sent by server as part of response that is finally seen by the user.

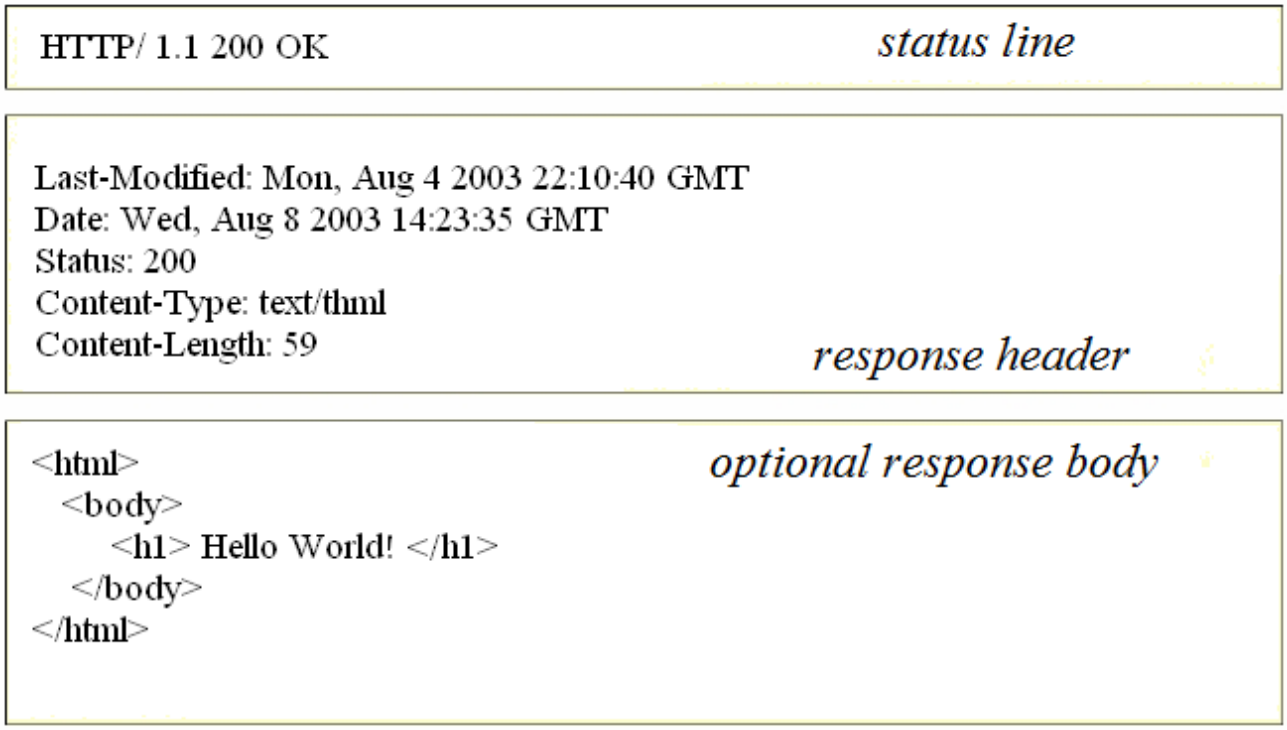


Figure 4: HTTP Response Example

25.3.3 HTTP Response Codes

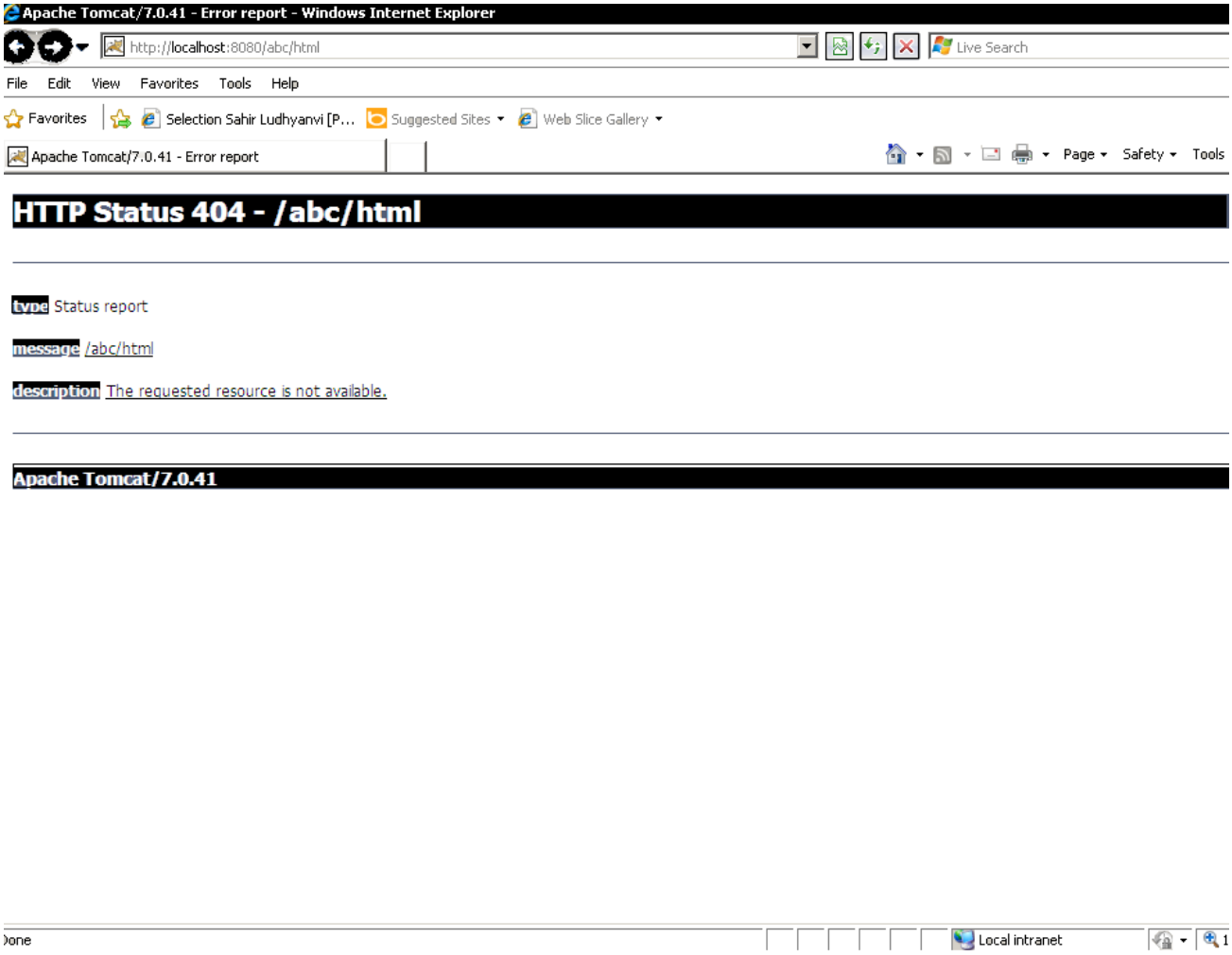
- An HTTP Response code tell the client about the state of the response i.e. whether it's a valid response or some error has occurred etc. HTTP Response codes fall into five general categories

  - **100-199**
    - Codes in the 100s are informational, indicating that the client should respond with some other action.
    - 100: Continue with partial request.

  - **200-299**
    - Values in the 200s signify that the request was successful.
    - 200: Means everything is fine.

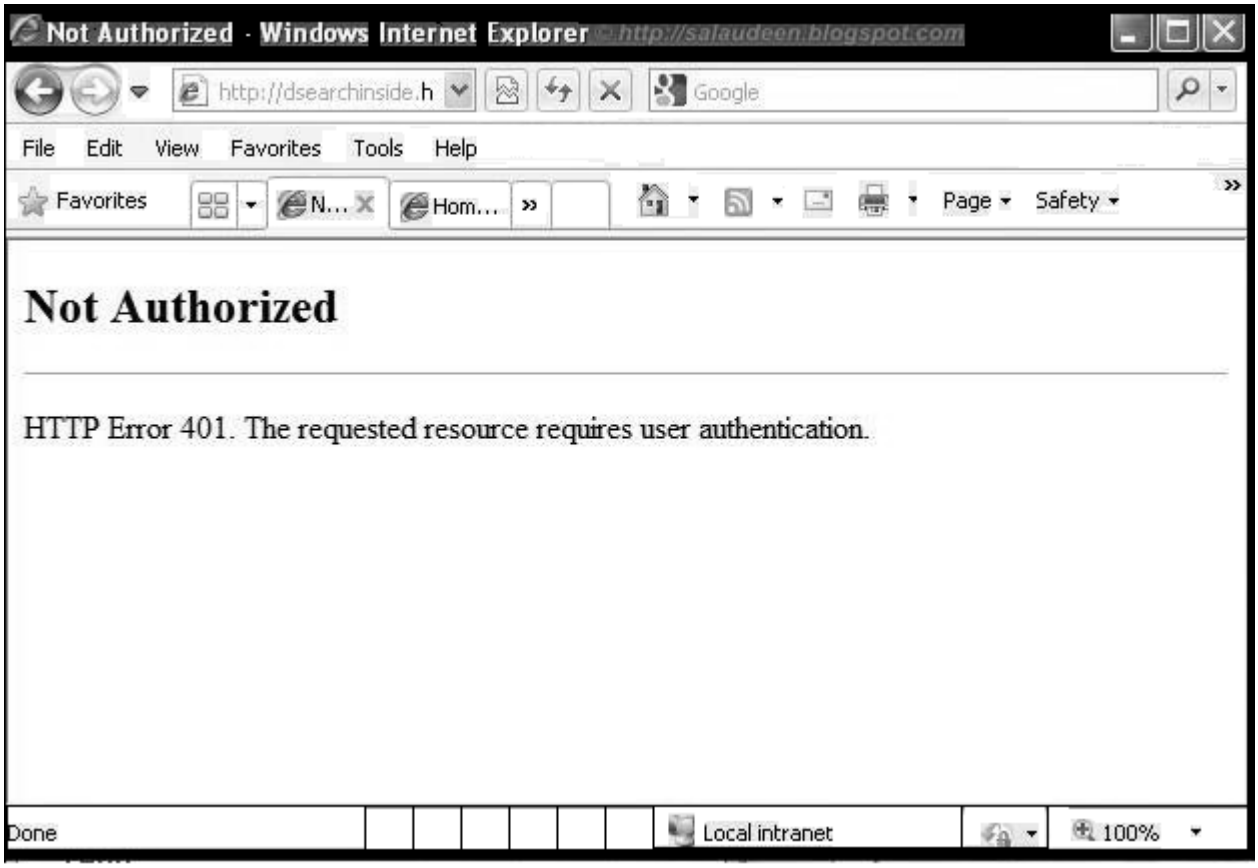
  - **300-399**
    - Values in the 300s are used for files that have moved and usually include a Location header indicating the new address.
    - 300: Document requested can be found several places; they'll be listed in the returned document.

- **400-499**
  - Values in the 400s indicate an error by the client.
  - 404: Indicates that the requested resource is not available.
  - 401: Indicates that the request requires HTTP authentication.
  - 403: Indicates that access to the requested resource has been denied.



- **500-599**
  - Codes in the 500s signify an error by the server.
  - 503: Indicates that the HTTP server is temporarily overloaded and unable to handle the request.

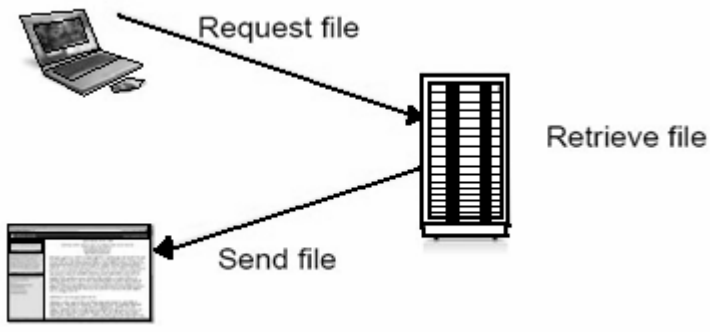
404: Indicates That The Requested Resource Is Not Available



401: Indicates That Request Requires HTTP Authentication

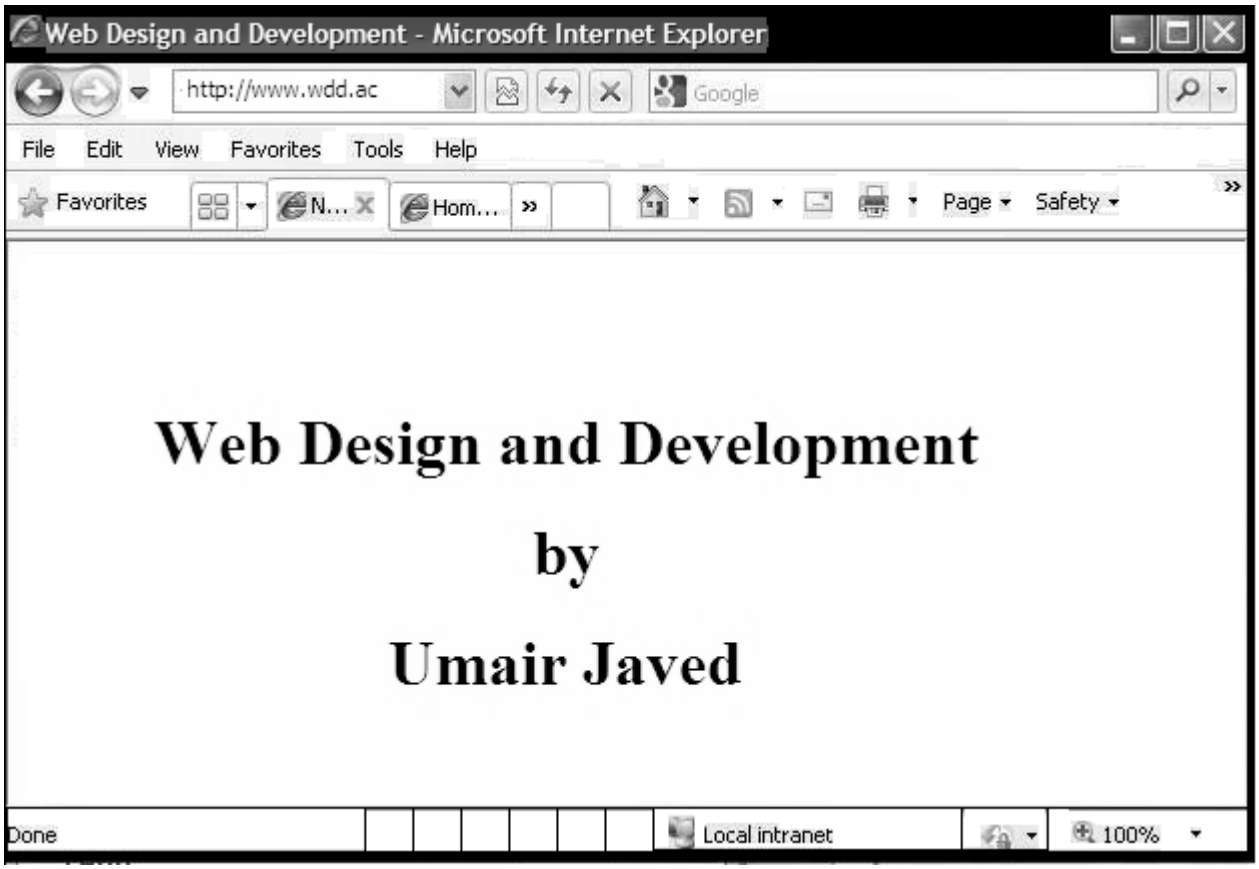
25.4 Server Side Programming

Web server pages can be either static pages or dynamic pages. A static web page is a simple HTML (Hyper Text Transfer Language) file. When a client requests an HTML page the server simple sends back response with the required page.

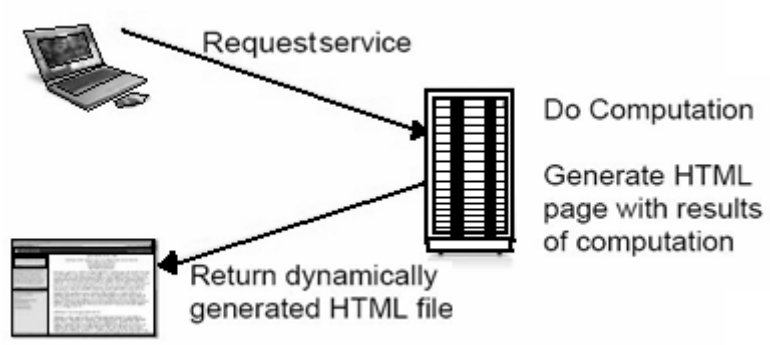


Static Web Page Request And Response

An example of static web page is given below



While in case of dynamic web page s server executes an application which generates HTML web pages according to specific requests coming from client. These dynamically generated web pages are sent back to client with the response.



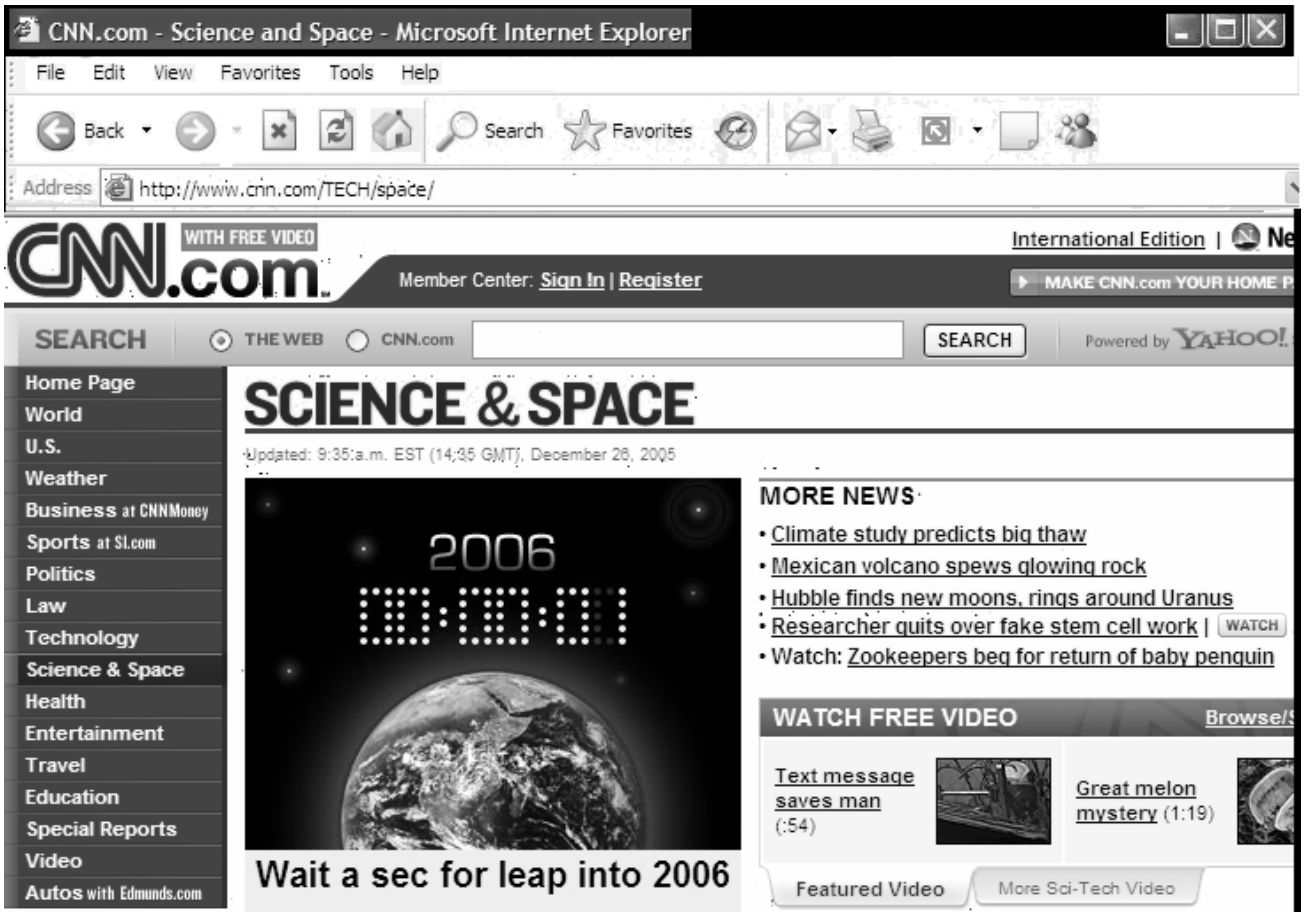
25.4.1 Why build Pages Dynamically?

We need to create dynamic web pages when the content of site changes frequently and client specific response is required. Some of the scenarios are listed below

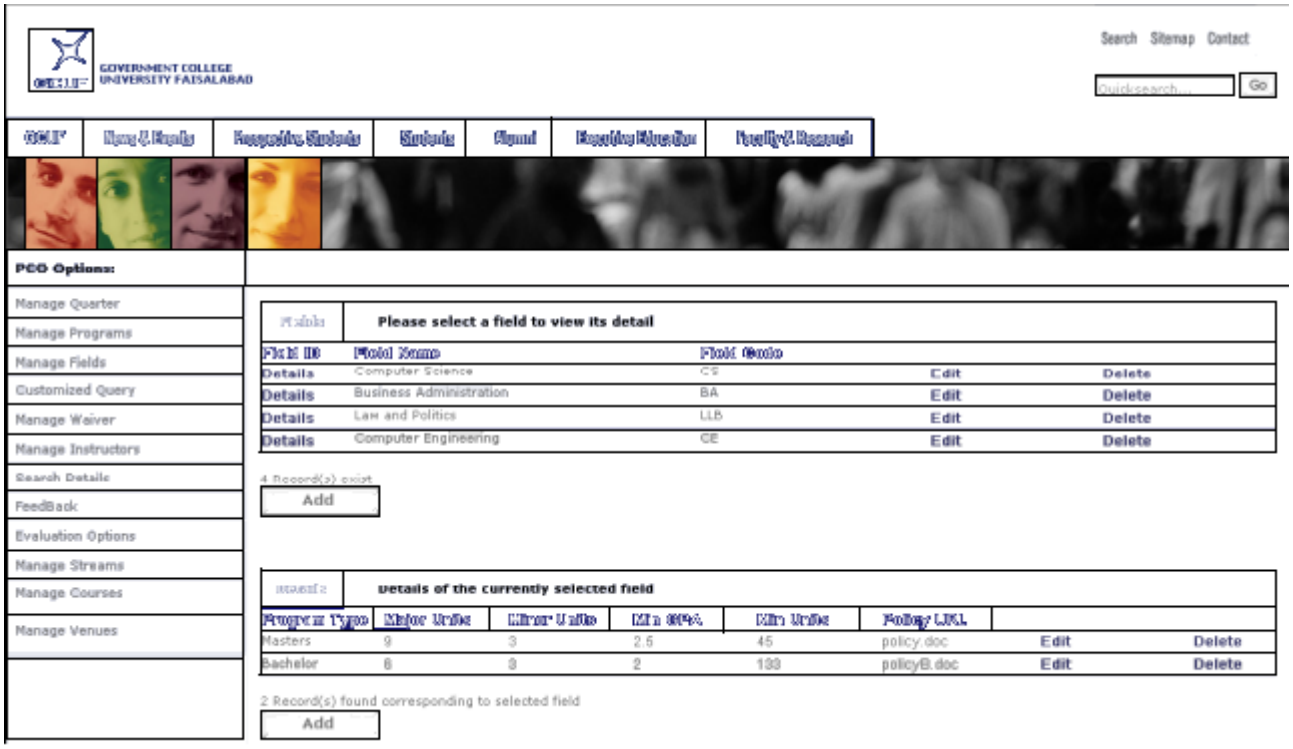
- The web page is based on data submitted by the user e.g. results page from search engines and order confirmation pages at on line stores.



- The Web page is derived from data that changes frequently e.g. a weather report or news headlines page.



- The Web page uses information from databases or other server-side resources e.g. an e-commerce site could use a servlet to build a Web page that lists the current price and availability of each item that is for sale.

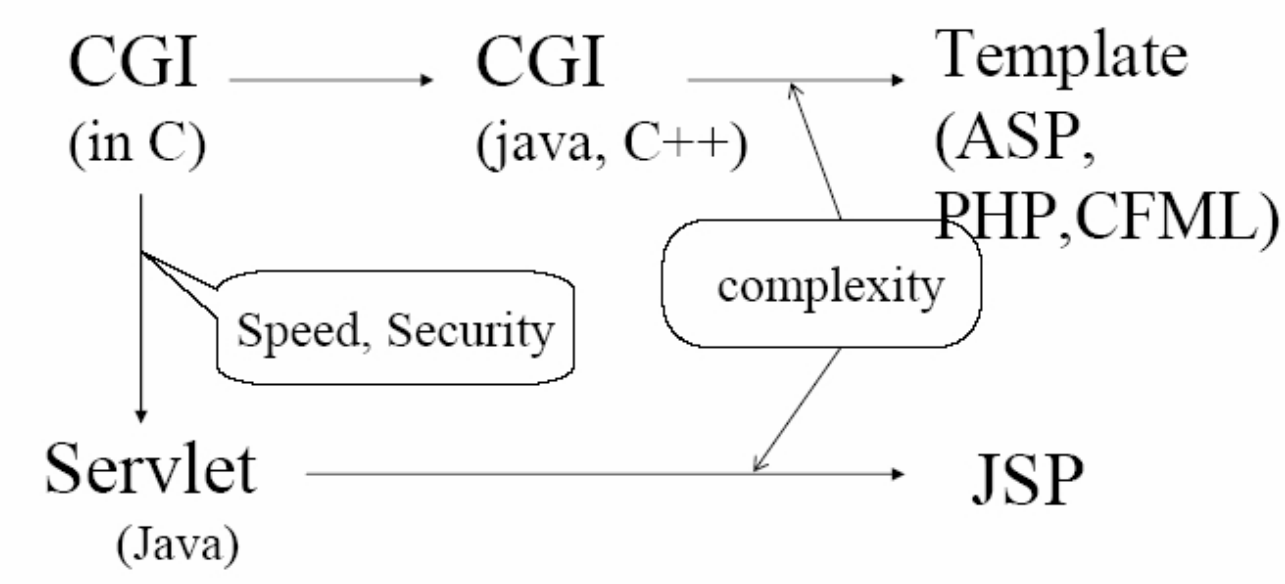


Server side programming involves

- Using technologies for developing web pages that include dynamic content.
- Developing web based applications which can produce web pages that contain information that is connection-dependent or time-dependent.

25.4.2 Dynamic Web Content Technologies Evolution

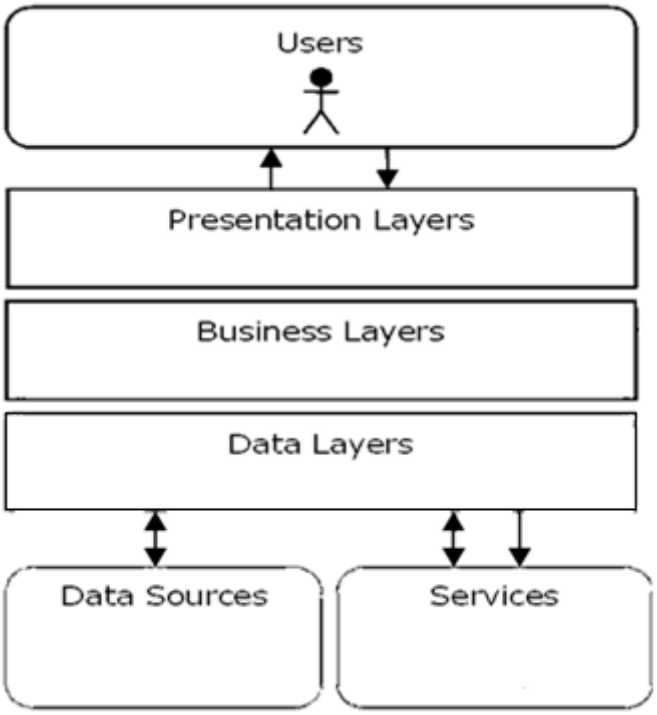
Dynamic web content development technologies have evolved through time in speed, security, ease of use and complexity. Initially C based CGI programs were on the server. Then template based technologies like ASP and PHP were then introduced which allowed ease of use for designing complex web pages. Sun Java introduced Servlets and JSP that provided more speed and security as well as better tools for web page creation.



Dynamic Web Content Technologies Evolution

25.5 Layers & Web Application

Normally web applications are partitioned into logical layers. Each layer performs a specific functionality which should not be mixed with other layers. Layers are isolated from each other to reduce coupling between them but they provide interfaces to communicate with each other.



Simplified View Of A Web Application And Its Layers

25.5.1 Presentation Layer:

- Provides a user interface for client to interact with application. This is the only part of application visible to client.

25.5.2 Business Layer

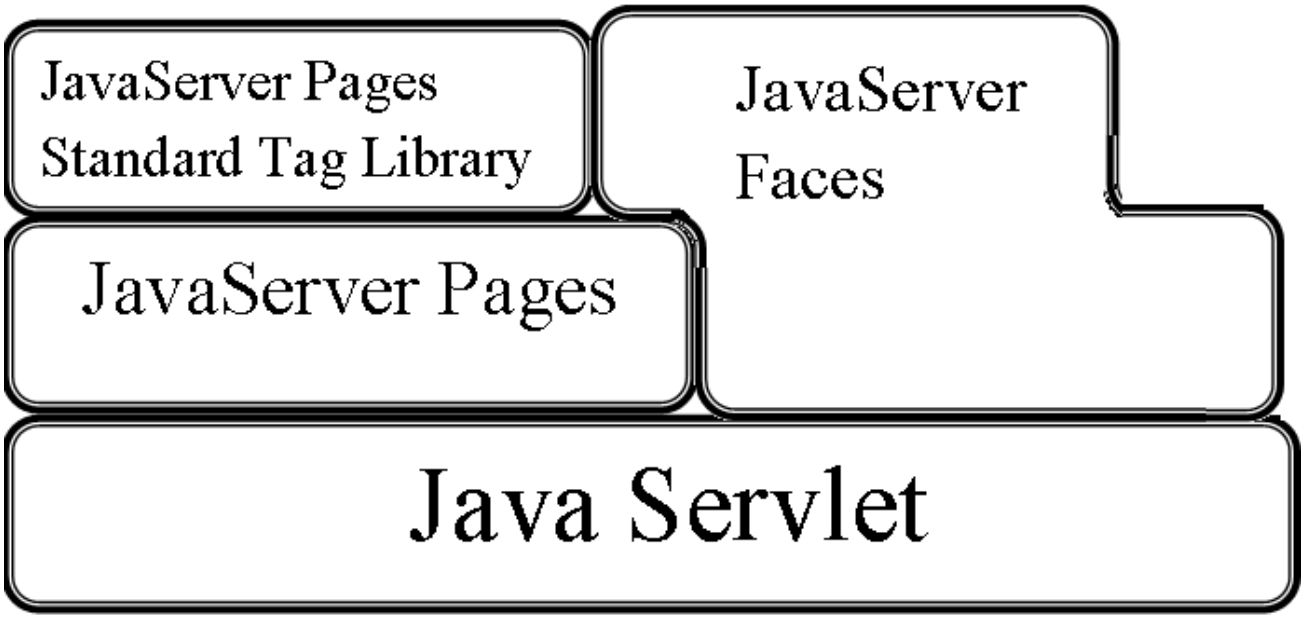
- The business or service layer implements the actual business logic or functionality of the application. For example in case of online shopping systems this layer handles transaction management.

25.5.3 Data Layer

- This layer consists of objects that represent real-world business objects such as an Order, OrderLineItem, Product, and so on.

25.6 Java - Web Application Technologies

There are several Java technologies available for web application development which includes Java Servlets, JavaServer Pages, and JavaServer Faces etc.



Java Web Application Technologies
(Presentation/Web Tier)

25.7 References:

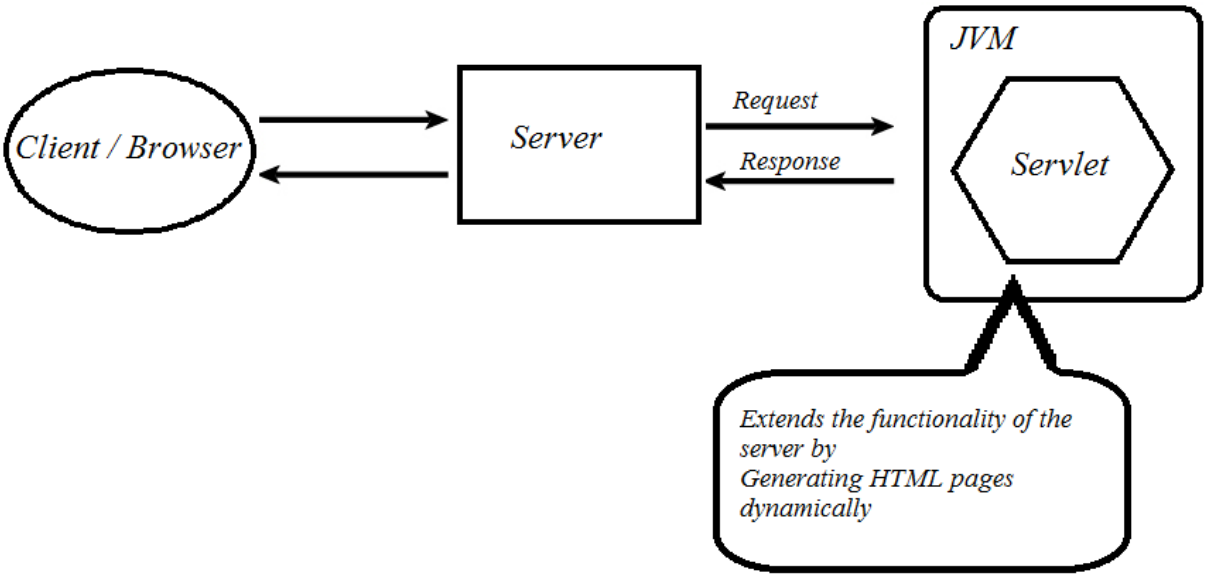
- Java, A Practical Guide by Umair Javed.
- Java tutorial by Sun: http://java.sun.com/docs/books/tutorial/ .

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 26: Java Servlets

Servlets are java technology's answer to CGI programming. CGI was widely used for generating dynamic content before Servlets arrived. They were programs written mostly in C, C++ that run on a web server and used to build web pages.

As you can see in the figure below, a client sends a request to web server, server forwards that request to a servlet, servlet generates dynamic content, mostly in the form of HTML pages, and returns it back to the server, which sends it back to the client. Hence we can say that servlet is extending the functionality of the webserver (The job of the earlier servers was to respond only to request, by may be sending the required html file back to the client, and generally no processing was performed on the server)



26.1 What Servlets can do?

- Servlets can do anything that a java class can do. For example, connecting with database, reading/writing data to/from file etc.
- Handles requests sent by the user (clients) and generates response dynamically (normally HTML pages).
- The dynamically generated content is send back to the user through a webserver (client)

26.2 Servlets vs. other SSP technologies

The java's servlet technology has following advantage over their counter parts:

26.2.1 Convenient

Servlets can use the whole java API e.g. JDBC. So if you already know java, why learn Perl or C. Servlets have an extensive infrastructure for automatically parsing and decoding HTML form data, reading and sending HTTP headers, handling cookies and tracking session etc and many more utilities.

26.2.2 Efficient

With traditional CGI, a new process is started for each request while with servlets each request is handled by a lightweight java thread, not a heavy weight operating system process. (more on this later)

26.2.3 Powerful

Java servlets let you easily do several things that are difficult or impossible with regular CGI. For example, servlets can also share data among each other

26.2.4 Portable

Since java is portable and servlets is a java based technology therefore they are generally portable across web servers

26.2.5 Inexpensive

There are numbers of free or inexpensive web servers available that are good for personal use or low volume web sites. For example Apache is a commercial grade webserver that is absolutely free. However some very high end web and application servers are quite expensive e.g. BEA weblogic. We'll also use Apache in this course

26.3 Software Requirements

To use java servlets will be needed

- J2SE
- Additional J2EE based libraries for servlets such as servlet-api.jar and jsp- api.jar. Since these libraries are not part of J2SE, you can download these APIs separately. However these APIs are also available with the web server you'll be using.

- A capable servlet web engine (webserver)

26.4 Jakarta Servlet Engine (Tomcat)



Jakarta is an Apache project and tomcat is one of its subprojects. Apache Tomcat is an open source web server, which is used as an official reference implementation of Java Servlets and Java Server Pages technologies.

Tomcat is developed in an open and participatory environment and released under the Apache software license

26.4.1 Environment Setup

To work with servlets and JSP technologies, you first need to set up the environment. Tomcat installation can be performed in two different ways **(a)** using .zip file **(b)** using .exe file. This setup process is broken down into the following steps:

1. Download the Apache Tomcat Server
2. Install Tomcat
3. Set the JAVA_HOME variable
4. Set the CATALINA_HOME variable
5. Set the CLASSPATH variable
6. Test the Server

26.4.2Environment Setup Using .zip File

Let's take a detail look on each step and get some hands on experience of environment setup.

26.4.2.1 Download the Apache Tomcat Server

From the http://tomcat.apache.org, download the zip file for the current release (e.g. jakarta-tomcat-5.5.9.zip or any latest version) on your C:\ drive. There are different releases available on site. Select to download .zip file from the Binary Distributions core section.

Note: J2SE 5.0 must be installed prior to use the 5.5.9 version of tomcat.

26.4.2.2 Installing Tomcat using .zip file

- Unzip the file into a location (e.g. C:\). (Rightclick on the zip file and select *unziphere* option)

- When the zip file will unzipped a directory structure will be created on your computer such as:

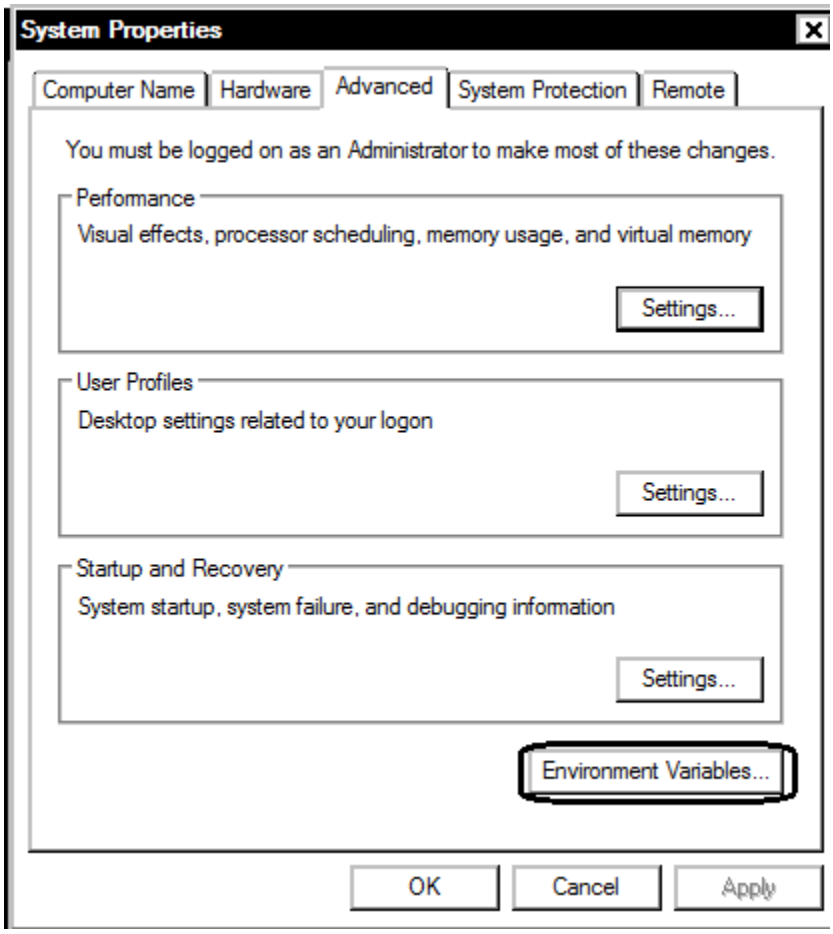


- The C:\jakarta-tomcat-5.5.9 folder is generally referred as **root** directory or CATALINA_HOME

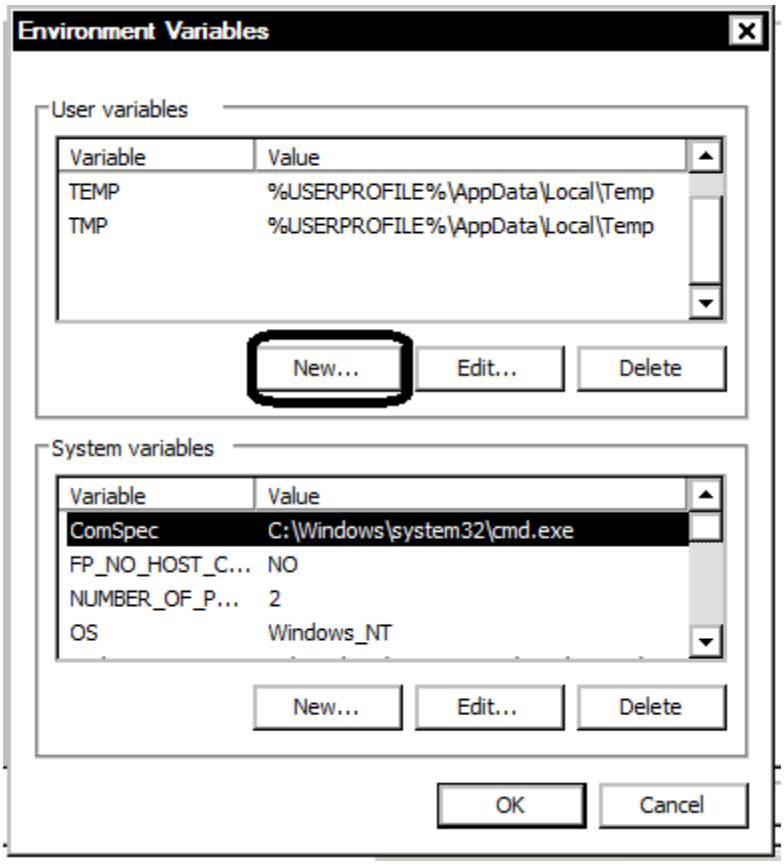
Note: After extraction**,** make sure C:\jakarta-tomcat-5.5.9 contains a bin subdirectory. Sometimes students create their own directory and unzip the file there such as C:\jakarta-tomcat-5.5.9\jakarta-tomcat-5.5.9.This causes problems while giving path information

26.4.2.3 Set the JAVA_HOME variable

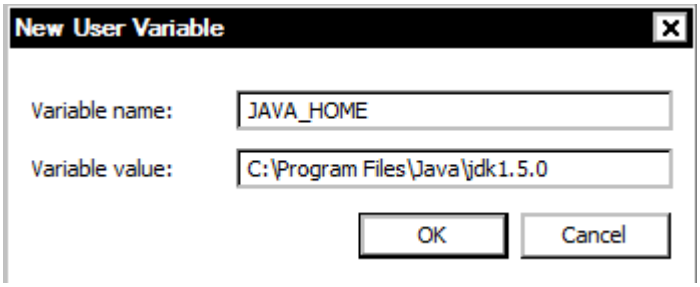
- JAVA_HOME indicates the root directory of your jdk. Set the JAVA_HOME environment variable to tell Tomcat, where to find java
- This variable should list the base JDK installation directory, not the bin subdirectory
- To set it, right click on My Computer icon. Select the advanced tab, a System Properties window will appear in front of you like shown below. Select the Environment Variables button to proceed.



- On clicking Environment Variable button, the Environment Variables window will open as shown next



- Create a new User variable by clicking New button as shown above, the New User Variable window will appear
- Set name of variable JAVA_HOME
- The value is the installation directory of JDK (for example C:\Program Files\j2sdk_nb\j2sdk1.4.2). This is shown below in the picture. **Please note that bin folder is not included in the path.**



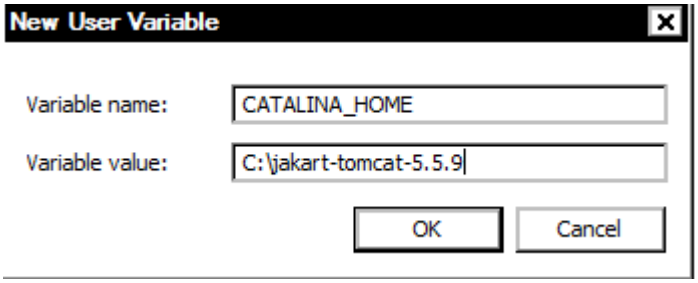
- Press Ok button to finish

26.4.2.4 Set the CATALINA_HOME variable

CATALINA_HOME is used to tell the system about the root directory of the TOMCAT. There are various files (classes, exe etc) needed by the system to run.

CATALINA_HOME is used to tell your system (in this case your web server Tomcat) where the required files are.

- To Set the CATALINA_HOME environment variable, create another User Variable.
- Type CATALINA_HOME as the name of the environment variable.
- Its value should be the path till your top-level Tomcat directory. If you have unzipped the Tomcat in C drive. It should be C:\jakarta-tomcat-5.5.9. This is shown below:



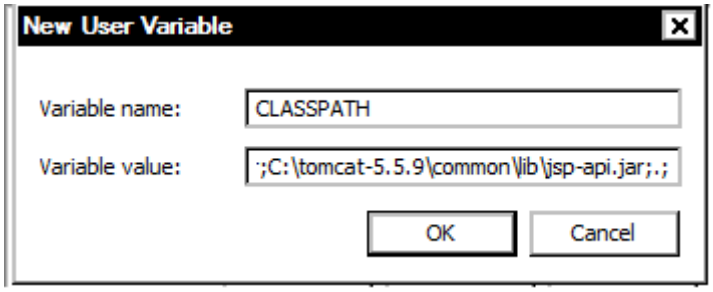
- Press Ok button to finish

Note: To run Tomcat (web server) you need to set only the two environment variables and these are JAVA_HOME & CATALINA_HOME

26.4.2.5 Set the CLASSPATH variable

Since servlets and JSP are not part of the Java 2 platform, standard edition, you have to identify the servlet classes to the compiler. The *server* already knows about the servlet classes, but the *compiler* (i.e., javac) you use for compiling source files of servlet does not. So if you don't set your CLASSPATH, any attempt to compile servlets, tag libraries, or other classes that use the servlet API will fail with error messages about unknown classes.

- To Set the CLASSPATH environment variable, create another User Variable.
- Type CLASSPATHas the name of the environment variable.
- Its value should be the path for servlet-api.jar and jsp-api.jar. These file can be found on following path:



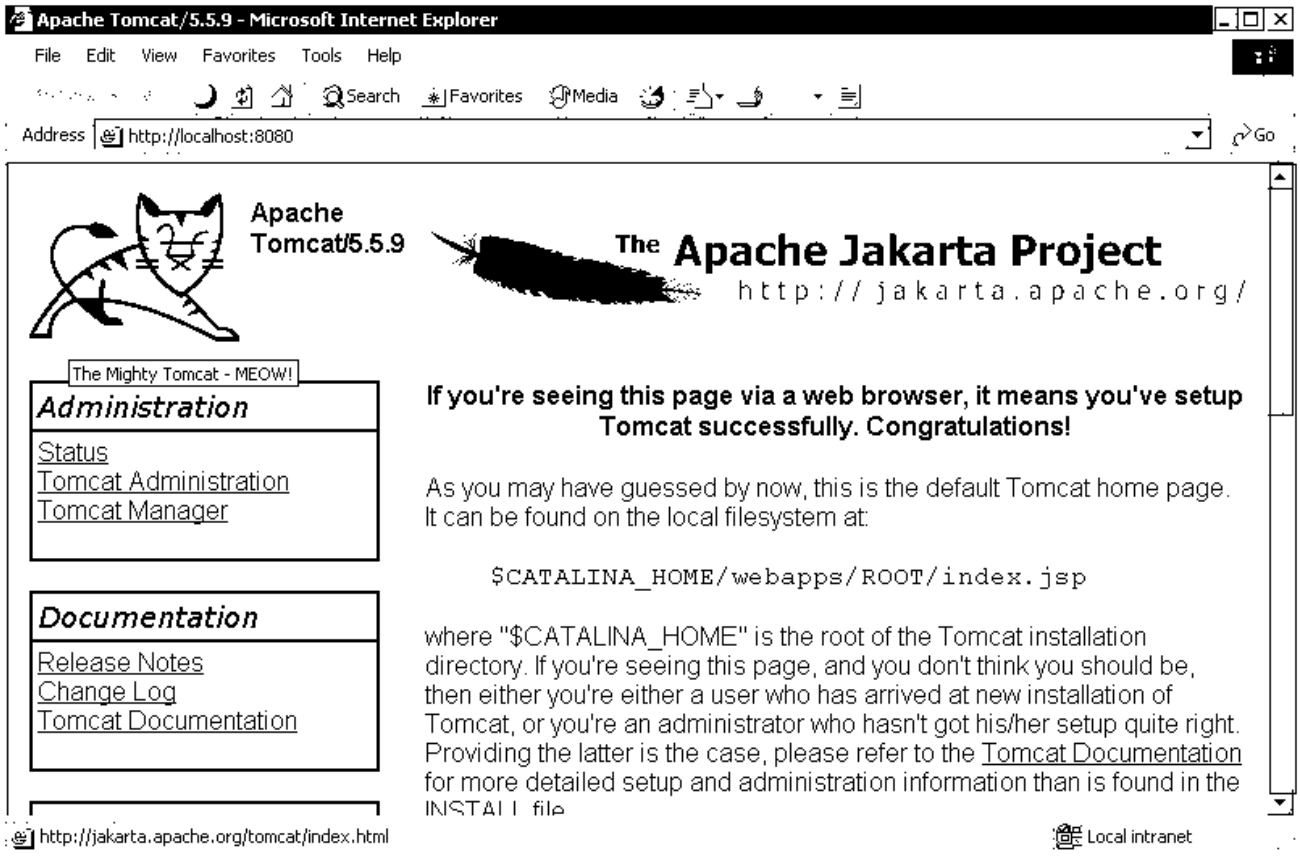
- Press OK button to finish the setting of CLASSPATH variable

26.4.2.6 Test the server

Before making your own servlets and JSP, verify that the server is working properly. Follow these steps in order to do that:

- Open the C:\jakarta-tomcat-5.5.9*bin* folder and locate the startup.batfile.
- Double clicking on this file will open up a DOS window, which will disappear, and another DOS window will appear, the second window will stay there. If it does not your paths are not correctly set.
- Now to check whether your server is workig or not, open up a browser window and type http://localhost:8080. This should open the default page of omcat as shown in next diagram:

Note: If default page doesn't displayed, open up an internet explorer window, move on to Tools Internet Options Connections LAN Settings. Make sure that option of *"Bypass proxy server for local addresses"* is unchecked.



There is another easier way to carry out the environment setup using .exefile. However, it is strongly recommended that you must complete the environment setup using .zip file to know the essential fundamentals.

26.4.3 Environment Setup Using .exe File

Let's look at the steps involved to accomplish the environment setup using .exefile.

26.4.3.1 Download the Apache Tomcat Server

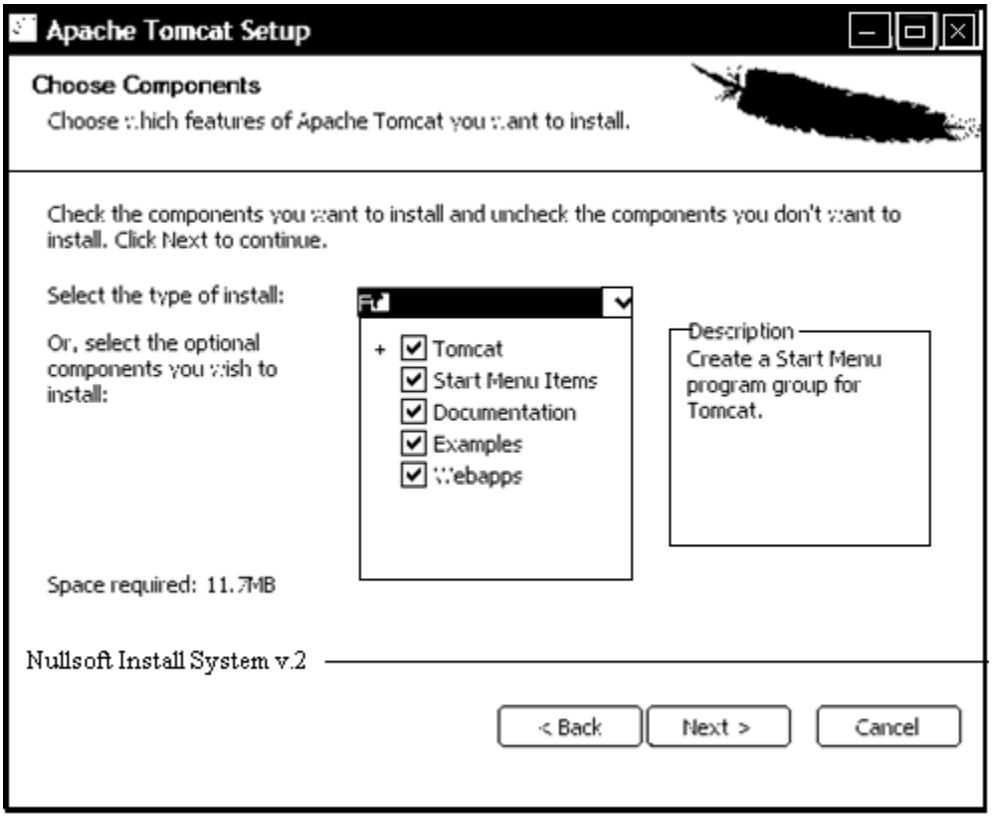
From the http://tomcat.apache.org, download the .exe file for the current release (e.g. jakarta-tomcat-5.5.9.zip) on your C:\ drive. There are different releases available on site. Select to download Windows executable (.exe) file from Binary Distributions Core section.

Note: J2SE 5.0 must be installed to use the 5.5.9 version of tomcat.

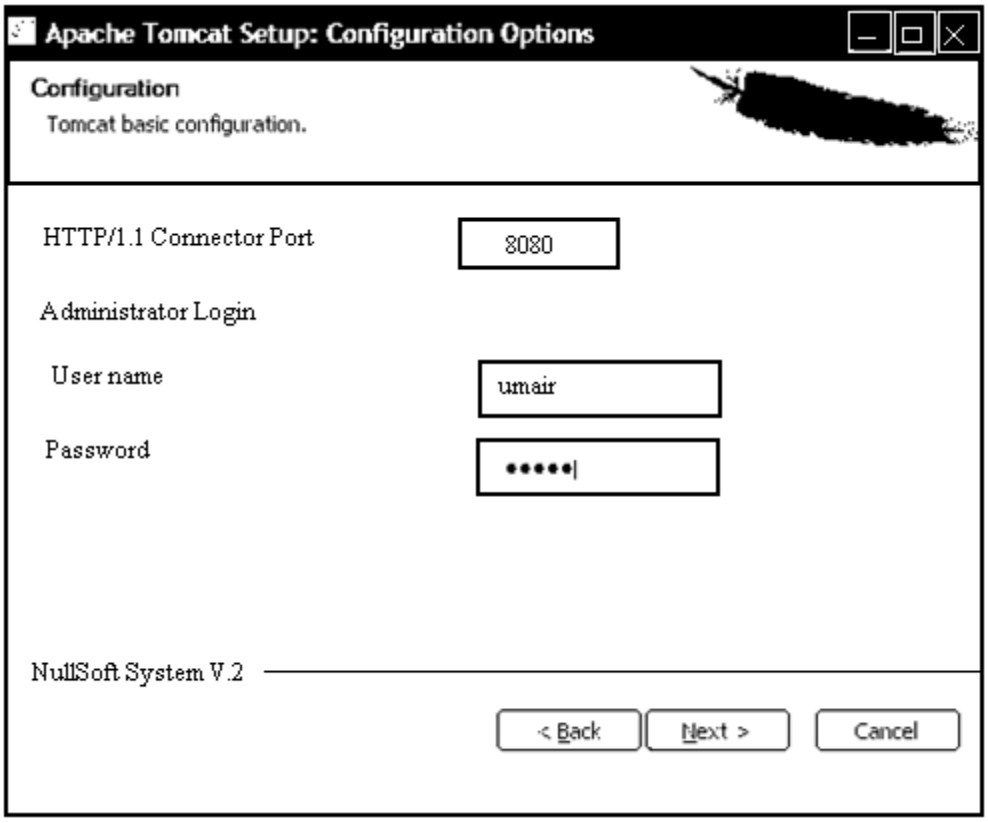
26.4.3.2 Installing Tomcat using .exe file

- Run the .exefile by double clicking on it.

- Moving forward in setup, you will reach to the following window



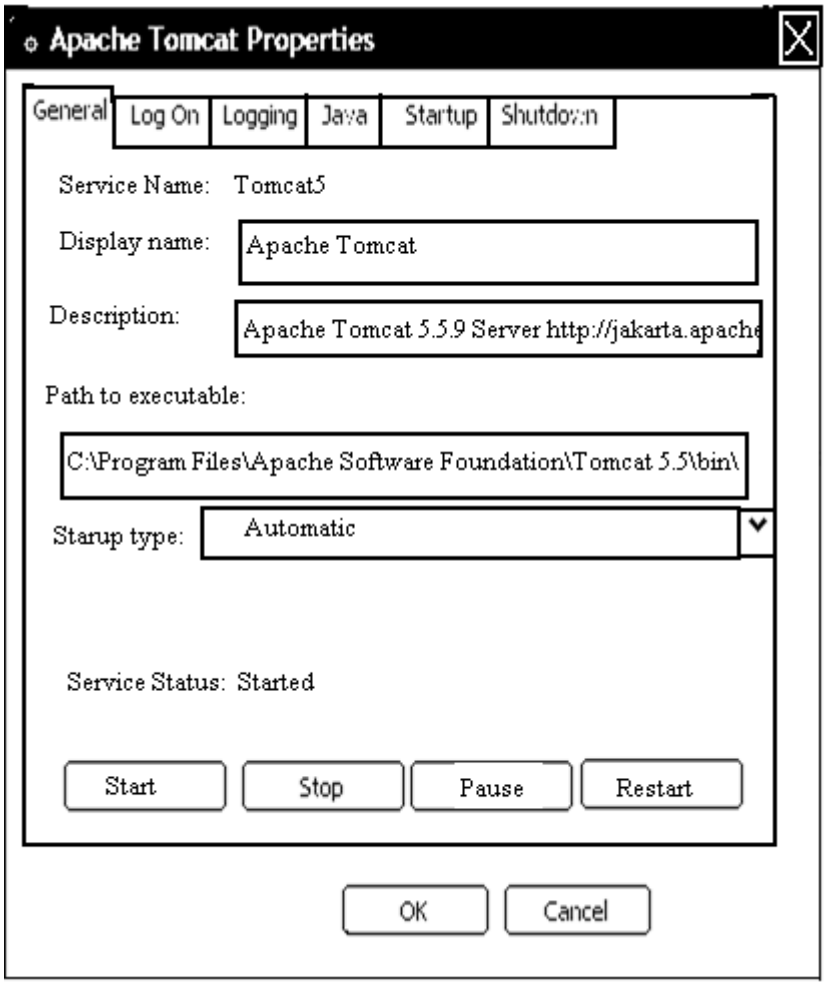
- Select install type "Full" and press Next button to proceed.
- Choose the folder in which you want to install Apache Tomcat and press *Next* to proceed.
- The configuration window will be opened. Leave the port unchanged (since by default web servers run on port 8080, you can change it if you really want to). Specify the user name & password in the specified fields and press Next button to move forward. This is also shown in the diagram coming next:



- The setup will automatically select the Java Virtual Machine path. Click
- Install button to move ahead.
- Finish the setup with the *Run Apache Tomcat* option selected. It will cause the tomcat server to run in quick launch bar as shown in diagram below. The Apache Tomcat shortcuts will also added to Programs menu.



- Double clicking on this button will open up Apache Tomcat Properties window. From here you can start or stop your web server. You can also configure many options if you want to. This properties window is shown below:



26.4.3.3 Set the JAVA_HOME variable

Choosing .exe mode does not require completing this step.

26.4.3.4 Set the CATALINA_HOME variable

Choosing .exe mode does not require completing this step.

26.4.3.5 Set the CLASSPATH variable

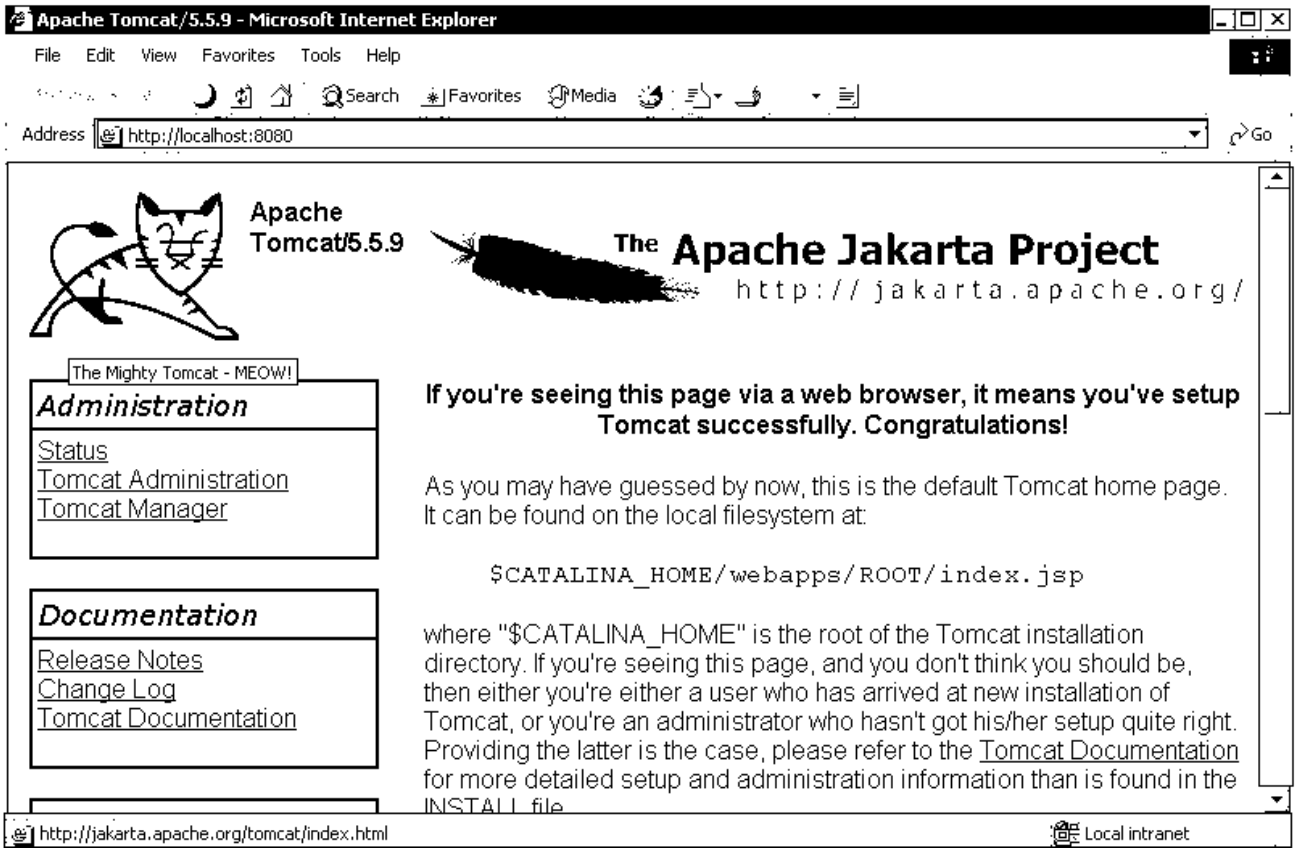
Same as step 5 of .zip installation mode

26.4.3.6 Test the server

If tomcat installation is made using .exe file, follow these steps

- Open the Apache Tomcat properties window by clicking on the Apache Tomcat button from Quick Launch.
- Start the tomcat server if it is not running by clicking on *Start* button.

- Open up a browser window and type http://localhost:8080. This should open the default page of Tomcat as shown in the next diagram:



Note: If default page doesn't displayed, open up an internet explorer window, move on to Tools Internet Options Connections LAN Settings. Make sure that option of "*Bypass proxy server for local addresses"* is unchecked.

26.5 References:

- Java, A Lab Course by Umair Javed.
- Java Servlet & JSP tutotrial http://www.apl.jhu.edu/~hall/**java**/Servlet-Tutorial/ .

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 27: Creating a Simple Web Application in Tomcat

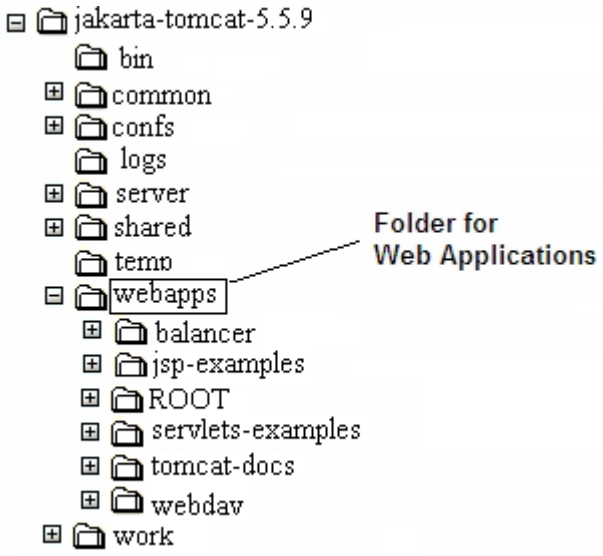
In this handout, we'll discuss the standard tomcat directory structure, a pre-requisite for building any web application. Different nuts and bolts of Servlets will also be discussed. In the later part of this handout, we'll also learn how to make a simple web application using servlet.

27.1 Standard Directory Structure of a J2EE Web Application

A web application is defined as a hierarchy of directories and files in a standard layout. Such hierarchies can be used in two forms

- Unpack
    - Where each directory & file exists in the file system separately
    - Used mostly during development
- Pack
    - Known as Web Archive (WAR) file
    - Mostly used to deploy web applications

The *webapps* folder is the top-level Tomcat directory that contains all the web applications deployed on the server. Each application is deployed in a separate folder often referred as *"context"*.

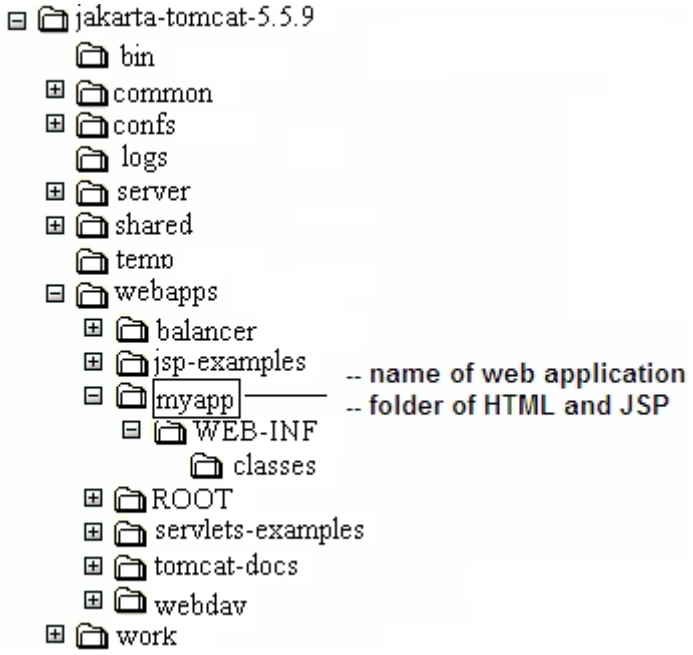


To make a new application e.g *myapp* in tomcat you need a specific folder hierarchy.

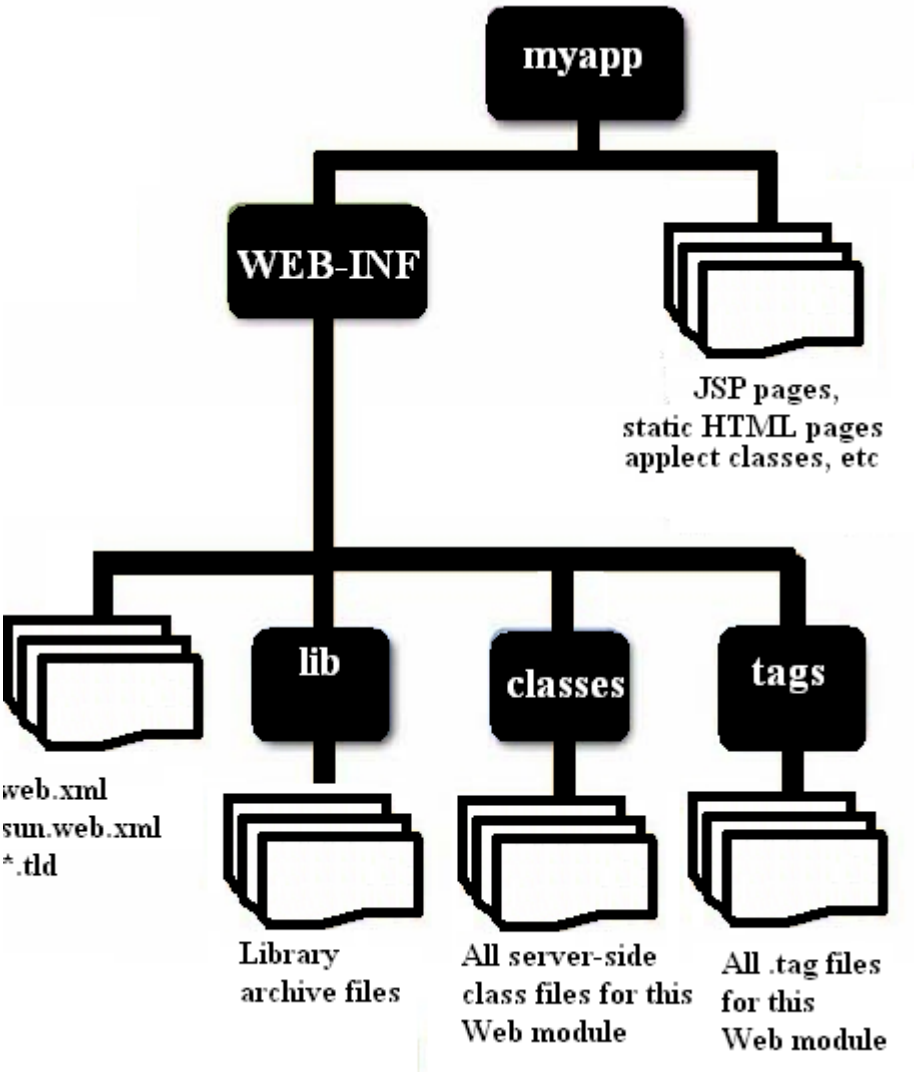
- Create a folder named *myapp* in *C:\jakarta-tomcat-5.5.9\webapps* folder. This name will also appear in the URL for your application. For example *http://localhost:8080/myapp/index.html*

- All JSP and html files will be kept in main application folder (C:\jakarta-tomcat-5.5.9\webapps\myapp)
- Create another folder inside myapp folder and change its name to WEB-INF. Remember WEB-INFis case sensitive and it is **not** WEB_INF
- Configuration files such as web.xml will go in WEB-INF folder (C:\jakarta-tomcat-5.5.9\webapps\myapp\WEB-INF)
- Create another folder inside WEB-INF folder and change its name to classes. Remember classesname is also case sensitive.
- Servlets and Java Beans will go in classes folder (C:\jakarta-tomcat-5.5.9\webapps\myapp\WEB-INF\classes)

That's the minimum directory structure required in order to get started. This is also shown in the figure below:



- To test application hierarchy, make a simple html file e.g. index.html file. Write some basic HTML code into it and save it in main application directory i.e. C:\jakarta-tomcat-5.5.9\webapps\myapp\
- Restart the server and access it by using the URL http://localhost:8080/myapp/index.html
- A more detailed view of the Tomcat standard directory structure is given below.



- Here you can see some other folders like lib& tags under the WEB-INF.
- The lib folder is required if you want to use some achieve files (.jar). For example an API in jar format that can help generating .pdffiles.
- Similarly tags folder is helpful for building custom tags or for using .tag files.

Note: Restart Tomcat every time you create a new directory structure, a servlet or a java bean so that it can recognize it. For JSP and html files you don't have to restart the server.

27.2 Writing Servlets

27.2.1 Servlet Types

- Servlet related classes are included in two main packages javax.servlet and javax.servlet.http.
- Every servlet must implement the javax.servlet.Servlet interface, it

contains the servlet's life cycle methods etc. (Life cycle methods will be discussed in next handout)

- In order to write your own servlet, you can subclass from GernericServlet or HttpServlet

27.2.1.1 GenericServlet class

- Available in javax.servlet package
- Implements *javax.servlet.Servlet*
  Extend your class from this class if you are interested in writing protocol independent servlets

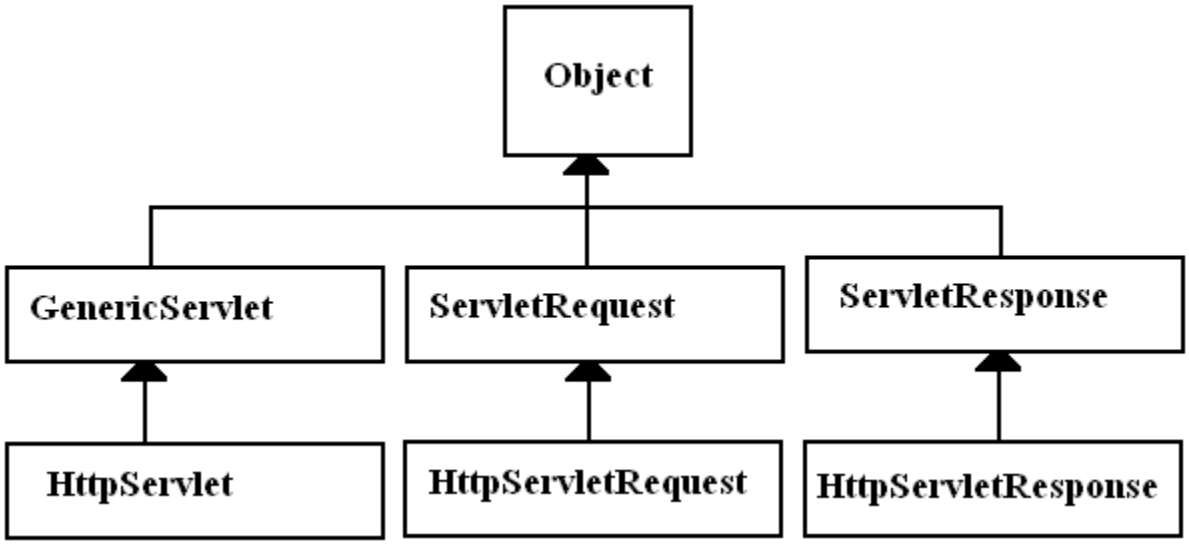
27.2.1.2 HttpServlet class

- Available in javax.servlet.http package
- Extends from GenericServletclass
- Adds functionality for writing HTTP specific servlets as compared to GernericServlet
- Extend your class from HttpServlet, if you want to write HTTP based servlets

27.3 Servlet Class Hierarchy

The Servlet class hierarchy is given below. Like all java classes GenericServlet also inherits from Object class. Apart from GenericServlet and HttpServlet classes, ServletRequest, HttpServletRequest, ServeltResponse and HttpServletResponse are also helpful in writing a servlet.

As you can guess ServletRequest & ServletResponse are used in conjunction with GenericServlet. These classes are used for processing protocol independent requests and generating protocol independent responses respectively.



HttpServletRequest & HttpServletRespose are used for processing HTTP protocol specific requests and generating HTTP specific response. Obviously these classes will be used in conjunction with HttpServet class, which means you are making a HTTP protocol specific servlet.

27.4 Types of HTTP requests

HTTP supports different types of request to be sent over to server. Each request has some specific purpose. The most important ones are **get** & **post**. Given below a brief overview of each request type is given. You can refer to RFC of HTTP for further details.

- **GET:** Requests a page from the server. This is the normal request used when browsing web pages.
- **POST:** This request is used to pass information to the server. Its most common use is with HTML forms.
- **PUT:** Used to put a new web page on a server.
- **DELETE:** Used to delete a web page from the server.
- **OPTIONS:** Intended for use with the web server, listing the supported options.
- **TRACE:** Used to trace servers

27.5 GET & POST, HTTP request types

Some details on GET and POST HTTP request types are given below.

- **GET**
    - Attribute-Value pair is attached with requested URL after '?'.

- For example if attribute is 'name' and value is 'ali' then the request will be http://www.gmail.com/register**?name=ali**
- For HTTP based servlet, override doGet () methods of HttpServlet class to handle these type of requests.

- **POST**

  - Attribute-Value pair attached within the *request body*. For your reference HTTP request diagram is given below again:



  - Override doPost()method of HttpServlet class to handle POST type requests.

27.6 Steps for making a Hello World Servlet

To get started we will make a customary *"HelloWorldServlet"*. Let's see what are the steps involved in writing a servlet that will produce *"Hello World"*

1. Create a directory structure for your application (i.e. helloapp). This is a one time process for any application
2. Create a HelloWorldServlet source file by extending this class from HttpServlet and overriding your desired method. For example doGet() or doPost().
3. Compile it (If get error of not having required packages, check your class path)
4. Place the class file of HelloWorldServlet in the classes folder of your web application (i.e. myapp).
   a. **Note:** If you are using packages then create a complete structure under classes folder

5. Create a deployment descriptor (web.xml) and put it inside WEB-INF folder
6. Restart your server if already running
7. Access it using Web browser

Example Code: HelloWorldServlet.java

```
//File HelloWorldServlet.java
// importing required packages
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
// extending class from HttpServelt
public class HelloWorldServlet extends HttpServlet {
/* overriding doGet() method because writing a URL in the browser by default generate request of GET type As you can see, HttpServletRequest and HttpServletResponse are passed to this method. These objects will help in processing of HTTP request and generating response for HTTP This method can throw ServletException or IOException, so we mention these exception types after method signature
*/
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
/* getting output stream i.e PrintWriter from response object by calling getWriter method on it As mentioned, for generating response, we will use HttpServletResponse object
*/

PrintWriter out = response.getWriter();

/* printing Hello World in the browser using PrintWriter object. You can also write HTML like out.println("<h1> Hello World </h1>")
*/
out.println("Hello World! ");
} // end doGet()
} // end HelloWorldServlet
```

Example Code: web.xml

eXtensible Markup Language (xml) contains custom defined tags which convey information about the content. To learn more about XML visit http://ww.w3schools.com.

Inside web.xml, the **<web-app>** is the root tag representing the web application. All other tags come inside of it.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app>
```

```
<servlet>
<servlet-name>          HelloWorldServlet </servlet-name>
<servlet-class> HelloWorldServlet </servlet-class>   </servlet>
<servlet-mapping>
<servlet-name> HelloWorldServlet </servlet-name> <url-pattern>
/myfirstservlet </url-pattern>
</servlet-mapping>
</web-app>
```

The **<servlet>** tag represents one's servlet name and its class. To specify the name of servlet, **<servlet-name>** tag is used. Similarly to specify the class name of servlet (it is the same name you used for making a servlet), **<servlet-class>**tag is used.

Note: It is important to note here that you can specify any name for a servlet inside **<servlet-name>** tag. This name is used for referring to servlet in later part of web.xml. You can think of it as your id assigned to you by your university while you have actually different name (like **<servlet-class>**).

Next we will define the servlet mapping. By defining servlet mapping we are specifying URL to access a servlet. **<servlet-mapping>** tag is used for this purpose.

Inside **<servlet-mapping>** tag, first you will write the name of the servlet for which you want to specify the URL mapping using **<servlet-name>** tag and then you will define the URL pattern using **<url-pattern>**tag. Notice the forward slash (/) is used before specifying the url. You can specify any name of URL. The forward slash indicates the root of your application.

<url-pattern> **/myfirstservlet** </url-pattern>

Now you can access HelloWorldServelt(if it is placed in myapp application) by giving the following url in the browser

http://localhost:8080**/myapp/myfirstservlet**

Note: Save this web.xml file by placing double quotes("web.xml") around it as you did to save .java files.

27.7 Compiling and Invoking Servlets

- Compile HelloWorldServlet.java using javac command.
- Put HelloWorldServlet.class in C:\jakarta-tomcat-5.5.9/webapps/myapp/WEB-INF/classesfolder
- Put web.xml file in C:\jakarta-tomcat-5.5.9/webapps/myapp/WEB-INF folder
- Invoke your servlet by writing following URL in web browser. Don't forget to

restart your tomcat server if already running

http://localhost:8080/myapp/myfirstservlet

Note: By using IDEs like netBeans® 4.1, you don't have to write web.xml by yourself or even to worry about creating directory structure and to copy files inappropriate locations. However manually undergoing this process will strengthen your concepts and will help you to understand the underlying mechanics.

27.8 References:

- Entire material for this handout is taken from the book **JAVA A Lab Course** by **Umair Javed**. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose without the consent of author.

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 28: Servlets Lifecycle

In the last handout, we have seen how to write a simple servlet. In this handout we will look more specifically on how servlets get created and destroyed. What different set of method are invoked during the lifecycle of a typical servlet.

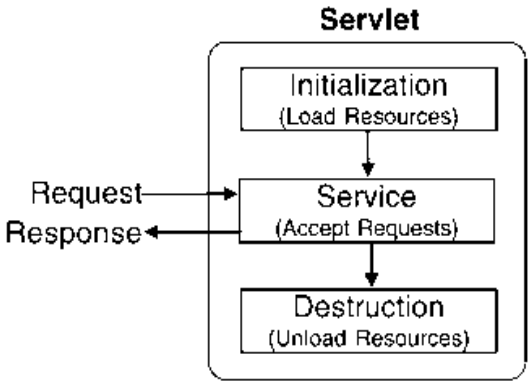
The second part consists on reading HTML form data through servlet technology. This will be explored in detail using code example

28.1 Stages of Servlet Lifecycle

A servlet passes through the following stages in its life.

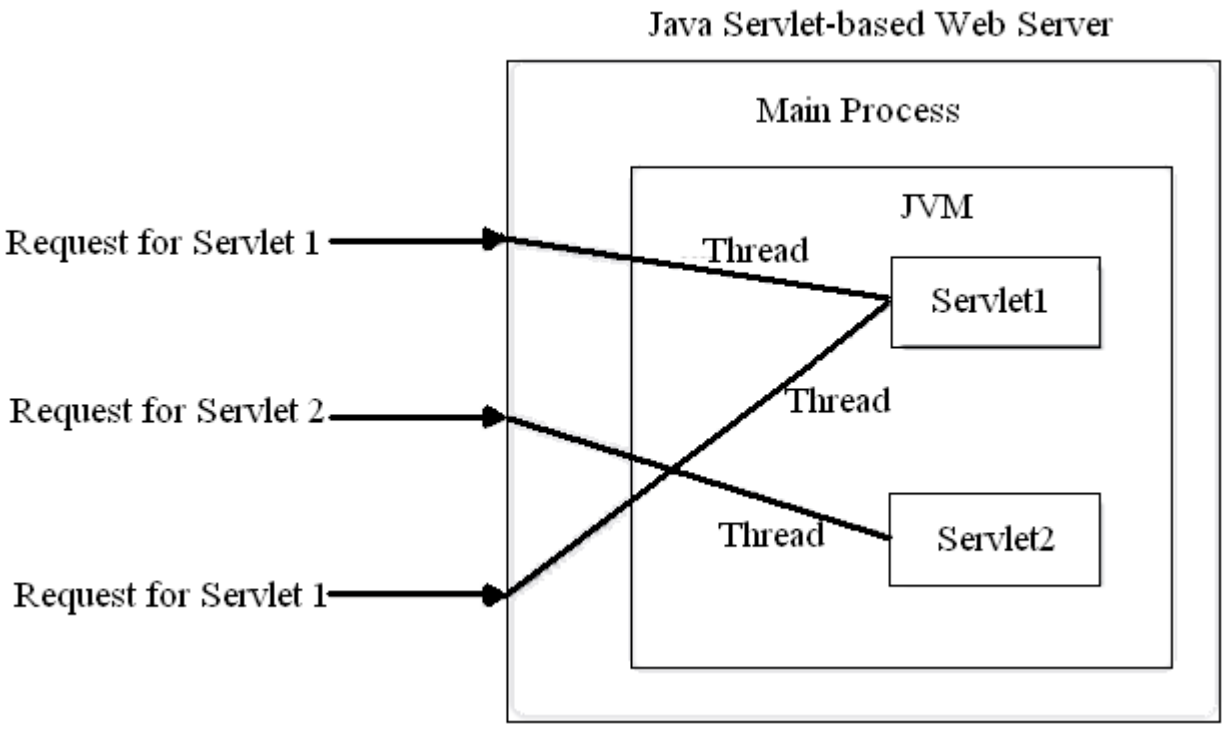
- Initialize
- Service
- Destroy

As you can conclude from the diagram below, that with the passage of time a servlet passes through these stages one after another.



28.1.1 Initialize

When the servlet is first created, it is in the initialization stage. The webserver invokes he init()method of the servlet in this stage. It should be noted here that init() is only called once and is not called for each request. Since there is no constructor available in Servlet so this urges its use for one time initialization (loading of resources, setting of parameters etc) just as the



init()method of applet.

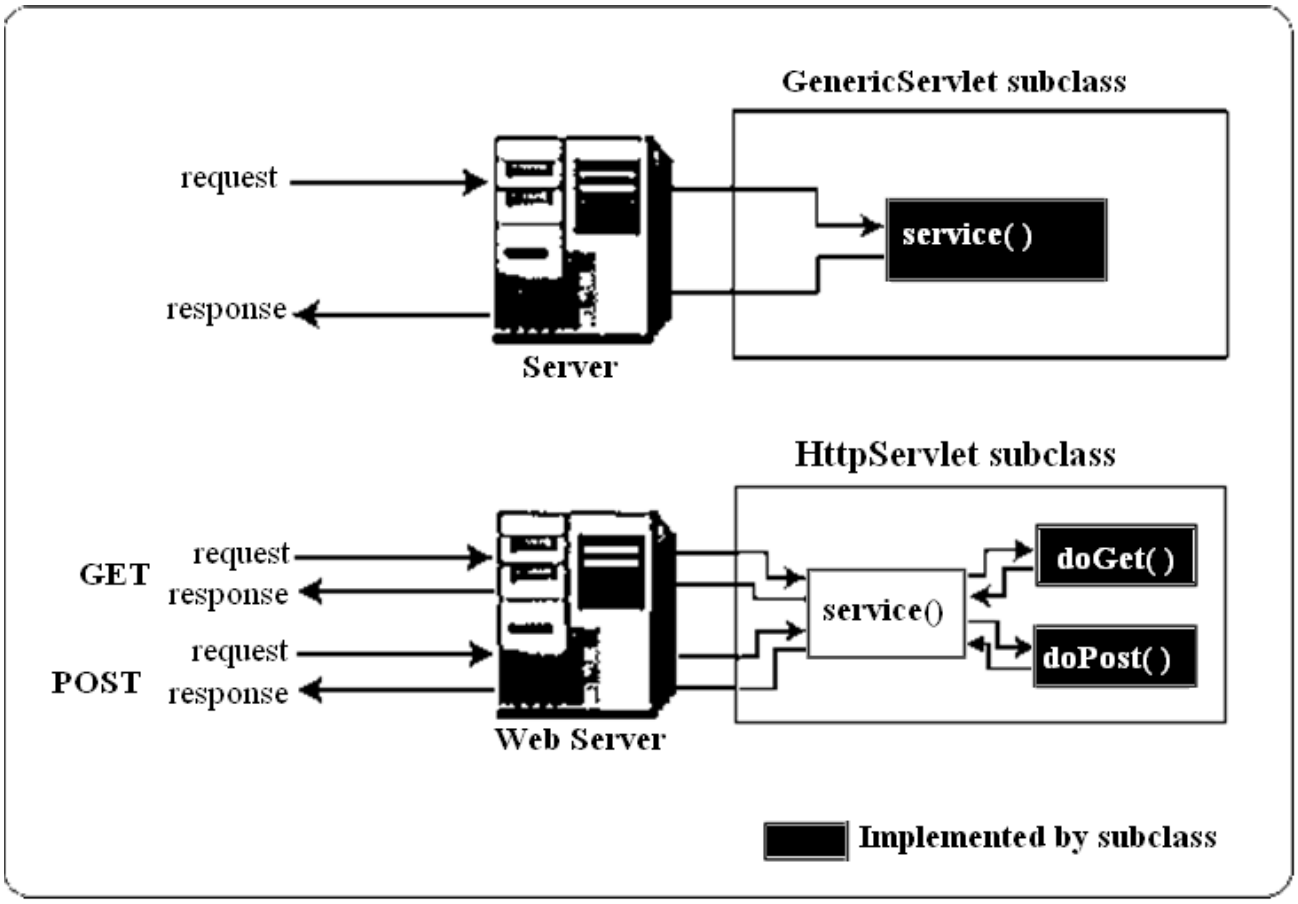
Initialize stage has the following characteristics and usage

- Executed once, when the servlet gets loaded for the first time
- Not called for each client request
- The above two points make it an ideal place to perform the startup tasks which are done in constructor in a normal class.

28.1.2Service

The service() method is the engine of the servlet, which actually processes the client's request. On every request from the client, the server spawns a new thread and calls the service() method as shown in the figure below. This makes it more efficient as compared to the technologies that use single thread to respond to requests.

The figure below show both versions of the implementation of service cycle. In the upper part of diagram, we assume that servlet is made by sub-classing from GenericServlet. (Remember, GenericServlet is used for constructing protocol independent servlets.). To provide the desired functionality, service() method is overridden. The client sends a request to the web server; a new thread is created to serve this request followed by calling the service()method. Finally a response is prepared and sent back to the user according to the request.



The second part of the figure illustrates a situation in which servlet is made using HttpServlet class. Now, this servlet can only serves the HTTP type requests. In these servlets doGet() and doPost() are overridden to provide desired behaviors. When a request is sent to the web server, the web server after creating a thread, passes on this request to service()method. The service() method checks the HTTP requests type (GET, POST etc) and calls the doGet() or doPost() method depending on how the request is originally sent. After forming the response by doGet() or doPost() method, the response is sent back to the service() method that is finally sent to the user by the web server.

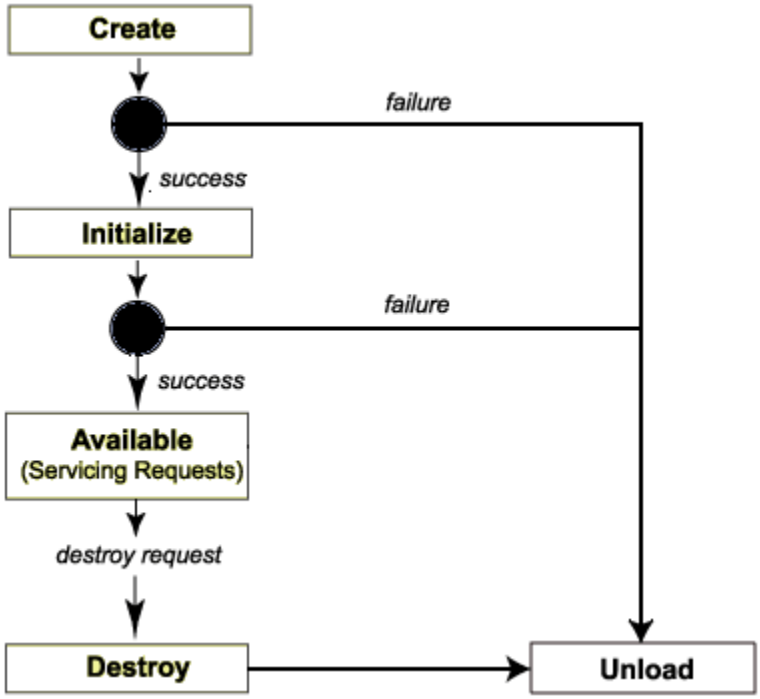
28.1.3 Destroy

The web server may decide to remove a previously loaded servlet instance, perhaps because it is explicitly asked to do so by the server administrator, or perhaps servlet container shuts down or the servlet is idle for a long time, or may be the server is overloaded. Before it does, however it calls the servlets destroy()method. This makes it a perfect spot for releasing the acquired resources.

28.2 Summary

- A Servlet is constructed and initialized. The initialization can be performed inside of init()method.
- Servlet services zero or more requests by calling service() method that may decide to call further methods depending upon the Servlet type (Generic or HTTP specific)
- Server shuts down, Servlet is destroyed and garbage is collected

The following figure can help to summarize the life cycle of the Servlet



The web sever creates a servlet instance. After successful creation, the servlet enters into initialization phase. Here, init() method is invoked for once. In case web server fails in previous two stages, the servlet instance is unloaded from the server.

After initialization stage, the Servlet becomes available to serve the clients requests and to generate response accordingly. Finally, the servlet is destroyed and unloaded from web server.

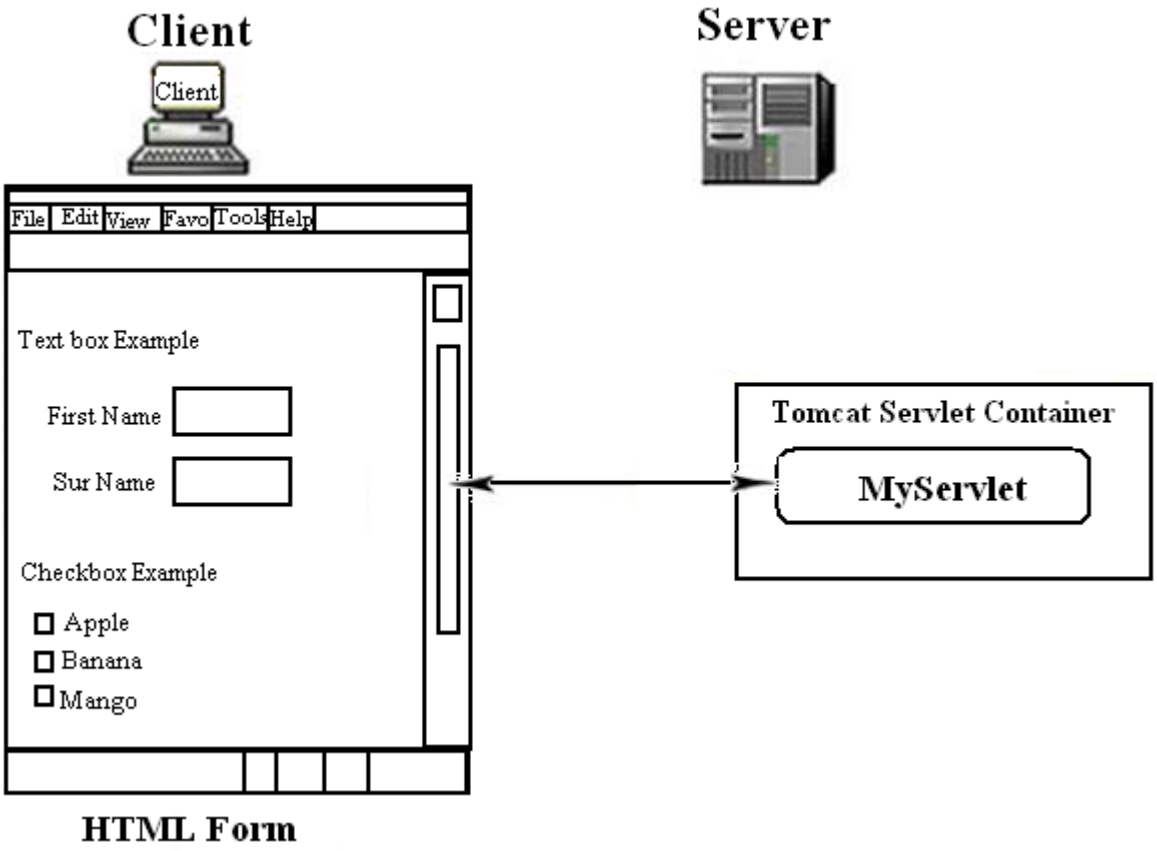
28.3 Reading HTML Form Data Using Servlets

In the second part, the required concepts and servlet technology is explored in order to read

HTML form data. To begin with, let's first identify in how many ways a client can send data

28.3.1HTML & Servlets

Generally HTML is used as a Graphics User Interface for a Servlet. In the figure below, HTML form is being used as a GUI interface for MyServlet. The data entered by the user in HTML form is transmitted to the MyServlet that can process this data once it read out. Response may be generated to fulfil the application requirements.



28.3.2 Types of Data send to Web Server

When a user submits a browser request to a web server, it sends two categories of data:

- **Form Data**

  Data that the user *explicitly* type into an HTML form. For example: registration information provided for creating a new email account.

- **HTTP Request Header Data**

Data, which is automatically, appended to the HTTP Request from the client for example, cookies, browser type, and browser IP address.

Based on our understanding of HTML, we now know how to create user forms. We also know how to gather user data via all the form controls: text, password, select, checkbox, radio buttons, etc. Now, the question arises: if I submit form data to a Servlet, how do I extract this form data from servlet? Figuring this out, provides the basis for creating interactive web applications that respond to user requests.

28.3.2.1 Reading HTML Form Data from Servlet

Now let see how we can read data from "HTML form" using Servlet. The HttpServletRequest object contains three main methods for extractingform data submitted by the user:

- getParameter(String name)
    - Used to retrieve a single form parameter and returns String corresponding to name specified.
    - Empty String is returned in the case when user does not enter any thing in the specified form field.
    - If the name specified to retrieve the value does not exist, it returns null.

  Note: You should only use this method when you are sure that the parameter has only one value. If the parameter might have more than one value, use getParamterValues().

- getParameterValues(String name)
    - Returns an array of Strings objects containing all of the given values of the given request parameter.
    - If the name specified does not exist, nullis returned

- getParameterNames()
    - If you are unsure about the parameter names, this method will be helpful
    - It returns Enumeration of String objects containing the names of the parameters that come with the request.
    - If the request has no parameters, the method returns an empty Enumeration.

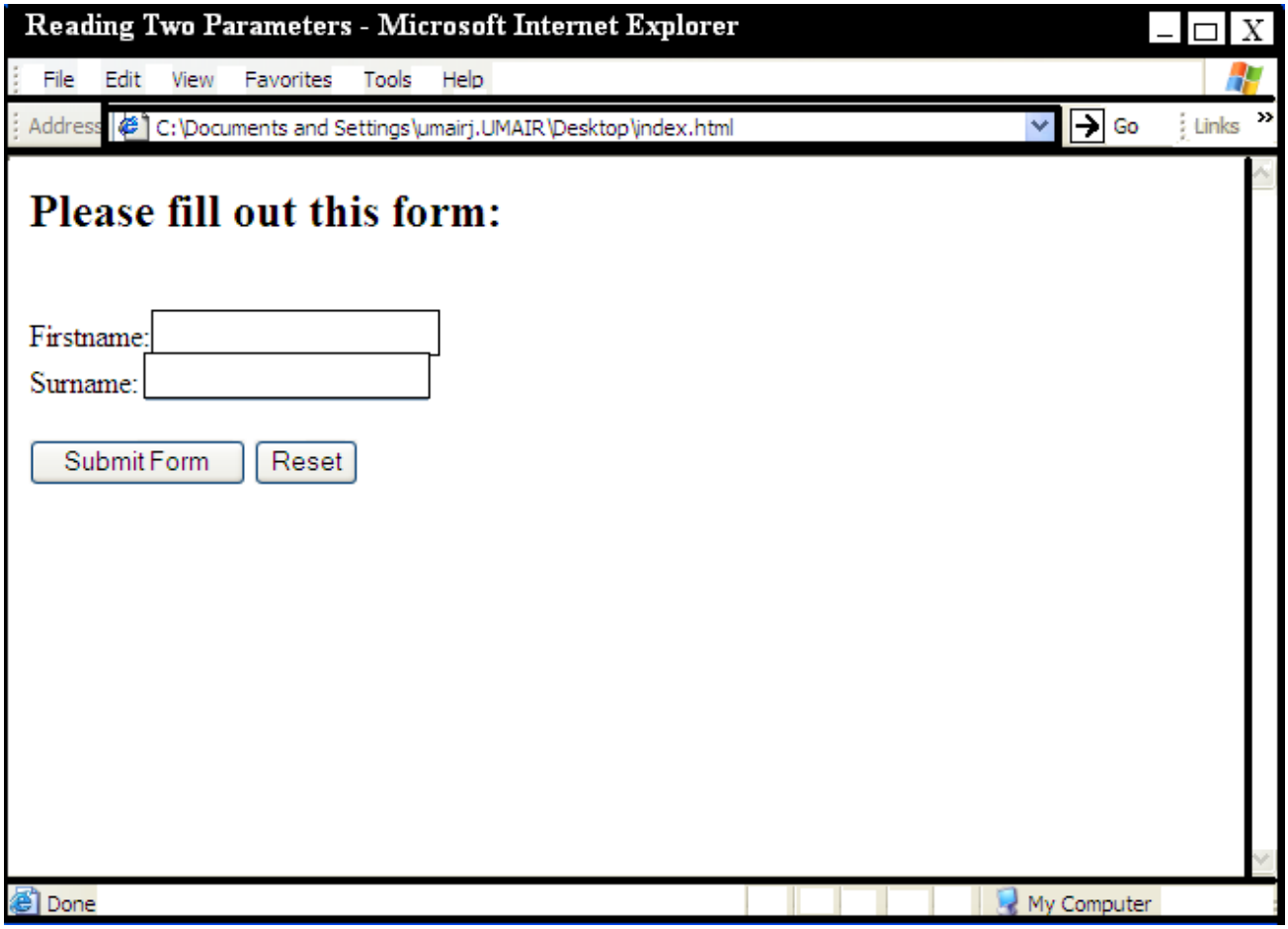
  Note: All these methods discussed above work the same way regardless of the request type(GET or POST). Also remember that form elements are case sensitive for example, "userName" is not the same as the "username."

Example Code: Reading Form Data using Servlet

This example consists of one HTML page (**index.html**), one servlet **(MyServlet.java**) and one xml file **(web.xml)** file. The HTML page contains two form parameters: firstName and surName. The Servlet extracts these specific parameters and echoes them back to the browser after appending "*Hello*".

Note: The example given below and examples later in coming handouts are built using netBeans®4.1. It's important to note that tomcat server bundled with netBeans® runs on *8084* port by default.

index.html



Let's have a look on the HTML code used to construct the above page.

```
<html>
<head>
<title> Reading Two Parameters </title> </head>
<body>
<H2> Please fill out this form: </H2>
<FORM METHOD="GET"
ACTION="http://localhost:8084/paramapp/formservlet"
NAME="myform" >
<BR> Firstname:
<INPUT TYPE = "text" NAME="firstName">
<BR> Surname:
<INPUT TYPE = "text" NAME="surName">
```

```
<BR>
<INPUT TYPE="submit" value="Submit Form">
<INPUT TYPE="reset" value="Reset">
</FORM>
</body>
</html>
```

Let's discuss the code of above HTML form. As you can see in the <FORM>tag, the attribute METHODis set to *"GET"***.** The possible values for this attribute can be GET and POST. Now what do these values mean?

- Setting the method attribite to *"GET"* means that we want to send the HTTP request using the GET method which will evantually activate the doGet() method of the servlet. In the GET method the information in the input fields entered by the user, merges with the URL as the query string and are visible to the user.
- Setting METHOD value to *"POST"* hides the entered information from the user as this information becomes the part of request body and activates doPost()method of the servlet.

Attribute ACTION of<FROM>tag is set to **http://localhost:8084/paramapp/formservlet**. The form data will be transmitted to this URL. paramapp is the name of web application created using netBeans. formservlet
is the value of <url-pattern> defined in the web.xml.The code of web.xml is given at the end.

The NAMEattribute is set to "*myform*" that helps when the same page has more than one forms. However, here it is used only for demonstration purpose.

To create the text fields where user can enter data, following lines of code come into play

<INPUT TYPE = "text" NAME="**firstName**">

<INPUT TYPE = "text" NAME="**surName**">

Each text field is distinguished on the basis of name assigned to them. Later these names also help in extracting the values entered into these text fields.

MyServlet.java

Now let's take a look at the servlet code to which HTML form data is submitted.

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
public class MyServlet extends HttpServlet
{
public void doGet(HttpServletRequest req, HttpServletResponse res)
throws ServletException, IOException
{
// reading first name parameter/textfield
String fName = req.getParameter("firstName");
// reading surname parameter/textfield
String sName = req.getParameter("surName");
// gettting stream from HttpServletResponse object
PrintWriter out = res.getWriter();
out.println("Hello: " + fName + " " + sName);
out.close();
}
}// end FormServlet
```

We started the code with importing three packages.

```
import java.io.*,
import javax.servlet.*;
import javax.servlet.http.*;
```

These packages are imported to have the access on PrintWriter, HttpServlet, HttpServletRequest, HttpServletResponse, ServletException and IOException classes.

The class MySevlet extends from HttpServlet to inherit the HTTP specific functionality. If you recall HTML code (index.html) discussed above, the value of mehtod attribute was set to *"GET"*. So in this case, we only need to override doGet() Method.

Entering inside doGet()method brings the crux of the code. These are:

String fName = req.getParameter(**"firstName"**); String sName = req.getParameter(**"surName"**);

Two String variables fNameand sNameare declared that receive String values returned by getParameter() method. As discussed earlier, this method returns String corresponds to the form parameter. Note that the values of nameattributes of input tags used in index.html have same case with the ones passed to getParameter() methods as parameters. The part of HTML code is reproduced over here again:

<INPUT TYPE = "text" NAME="**firstName**">

<INPUT TYPE = "text" NAME="**surName**">

In the last part of the code, we get the object of PrintWriterstream from the object of HttpServletResponse. This object will be used to send data back the response. Using PrintWriter object (out), the names are printed with appended *"Hello"* that becomes visible in the browser.

web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?> <web-app>
<servlet>
<servlet-name> FormServlet </servlet-name>
<servlet-class> MyServlet </servlet-class> </servlet>
<servlet-mapping>
<servlet-name> FormServlet </servlet-name>
<url-pattern> /formservlet </url-pattern>        </servlet-mapping>
</web-app>
```

The <servlet-mapping> tag contains two tags <servlet-name> and <urlpatteren> containing name and pattern of the URL respectively. Recall the value of action attribute of the <form>element in the HTML page. You can see it is exactly the same as mentioned in <url-pattern> tag.

http://localhost:8084/paramapp/**formservlet**

28.4 References:

- JAVA a Lab Course by Umair Javed
- Java API documentation
- Core Servlets and JSP by Marty Hall

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 29: More on Servlets

The objective of this handout is to learn about the use and implementation of initialization parameters for a Servlet. Moreover different ways of redirecting response and forwarding or including requests also discussed in detail.

29.1 Initialization Parameters

Some times at the time of starting up the application we need to provide some initial information e,g, name of the file where server can store logging information, DSN for database etc. Initial configuration can be defined for a Servlet by defining some string parameters in web.xml. This allows a Servlet to have initial parameters from outside. This is similar to providing command line parameters to a standard console based application.

Example: setting init parameters in web.xml

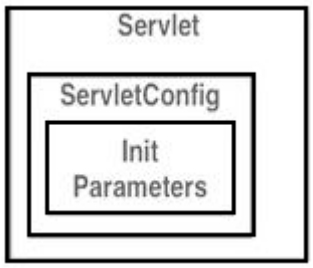
Let's have a look on the way of defining these parameters in web.xml

```
<init-param> //defining param 1
<param-name> param1 </param-name>
<param-value> value1 </param-value>
</init-param>
<init-param> //defining param 2
<param-name> param2 </param-name>
<param-value> value2 </param-value> </init-param>
```

In the above code, it is shown that for each parameter we need to define separate <initparam>tag that have two sub tags <param-name> and <param-value>, which contain the name and values of the parameter respectively.

29.1.1 ServletConfig

Every Servlet has an object called **ServletConfig** associated with it as shown in the fig. below. It contains relevant information about the Servlet like initialization parameters defined in web.xml



29.1.2 Reading Initialization Parameters

Now let's see, how we can access init parameters inside the Servlet. The method getInitParameter()of ServletConfig is usually used to access init parameters. It takes a String as parameter, matches it with <param-name> tag under all <init-param> tags and returns <param-value> from the web.xml

One way is to override init() method as shown in the code below. The ServletConfigobject can then be used to read initialization parameter.

```
public void init(ServletConfig config) throws ServletException {
String name = config.getInitParameter("paramName");
}
```

Another way to read initialization parameters out side the init () method is

- Call getServletConfig() to obtain the ServletConfig object
- Use getInitParameter() of ServletConfig to read initialization parameters

```
public void anyMethod() // defined inside servlet
{
ServletConfig config = getServletConfig();
String name = config.getInitParameter("param_name");
}
```

Example Code: Reading init parameters

MyServlet.javawill read the init parameter (log file name) defined inside web.xml. The code is given below:

```
import java.io.*;
import java.net.*;
```

```
import javax.servlet.*;
import javax.servlet.http.*;
public class MyServlet extends HttpServlet {
// attribute used to store init-parameter value
String fileName;
// overriding init() method
public void init(ServletConfig config)
throws ServletException{
super.init(config);
// reading init-parameter “logfilename” stored in web.xml
fileName = config.getInitParameter("logfilename");
}

/*
Both doGet() & doPost() methods are override over here. processRequest() is called from both these
methods. This makes possible for a servlet to handle both POST and GET requests identically.
*/
// Handles the HTTP GET request type
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
processRequest(request, response);
}
// Handles the HTTP POST request type
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
processRequest(request, response);
}
// called from doGet() & doPost()
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
PrintWriter out = response.getWriter();

// writing init-parameter value that is store in fileName
out.println(fileName);
out.close();
}
} // end MyServlet
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?> <web-app>

<servlet>
```

```
<servlet-name> MyServlet </servlet-name>
<servlet-class> MyServlet </servlet-class>
<init-param>
<param-name> logfilename </param-name>
<param-value> logoutput.txt </param-value> </init-param>
</servlet>
<servlet-mapping>
<servlet-name> MyServlet </servlet-name>
<url-pattern> /myservlet </url-pattern>          </servlet-mapping>
</web-app>
```

29.1.3 Response Redirection

We can redirect the response of the Servlet to another application resource (another Servlet, an HTML page or a JSP) but the resource (URL) must be available to the calling Servlet, in the same Servlet context (discussed later).

There are two forms of response redirection that can be possible:

- Sending a standard redirect
- Sending a redirect to an error page

29.1.4 Sending a standard Redirect

- Using response.sendRedirect("myHtml.html") method, a new request is generated which redirects the user to the specified URL.
- If the URL is of another Servlet, that second Servlet will not have access to the *original request object*. For example, if the request is redirected from servlet1 to servlet2, then servlet2 would not be able to access the request object of servlet1.
- To have access to the original request object, you must use the *request dispatching* technique (discussed later) instead of redirect.

29.1.5 Sending a redirect to an error page

Instead of using response.sendRedirect (), wecan useresponse.sendEorror () to show user an error page. This method takes two parameters, first the error number that is a predefined constant of the response class (listed below) and seconds the appropriate error message. The steps to redirect the user to an error page are:

- An error code is sent as a parameter of response.sendError (int, msg) method
- The error page is displayed with the *msg* passed to method
- The error numbers are predefined constants of the HttpServletResponse class. For example:
    - SC_NOT_FOUND (404)
    - SC_NO_CONTENT (204)
    - SC_REQUEST_TIMEOUT (408)

Example Code: Response Redirection

The example given below demonstrates a typical sign on example in which a user is asked to provide login/password, providing correct information leads to welcome page or otherwise to a registration page. This example consists of login.html, welcome.html, register.html and MyServlet.java files. Let's examine these one after another.

login.html

This page contains two text fields; one for entering username and another for password. The data from this page is submitted to MyServlet.java.

```
<html>
<body>
<h2> Please provide login details</h2>
<FORM METHOD="POST"
ACTION="http://localhost:8084/redirectionex/myservlet" NAME="myForm" >
<BR> User Id:
<INPUT TYPE="text" name="userid"/>
<BR> Password:
<INPUT TYPE="password" name="pwd"/>
<BR> <BR>
<input type="submit" value="Submit Form"/>
</form>
</body>
</html>
```

welcome.html

The user is directed to this page only if user provides correct login / password. This page only displays a successfully logged-in message to the user.

```
<html>
<body>
<h2> You have successfully logged in </h2> </body>
</html>
```

register.html

The user is redirected to this page in case of providing incorrect login/password information. The user can enter user id, address and phone number here to register.

Note: The code given below will only show fields to the user. It does not register user as no such functionality is added into this small example.

```
<html>
<body>
<h2>Your login is incorrect. Please register yourself</h2>
<FORM METHOD="POST" ACTION=""     NAME="myForm">
<BR> Name:
<INPUT TYPE="text" NAME="userid"/>
<BR> Address:
<INPUT TYPE="text" NAME="address"/>
<BR> Phone No:
<INPUT TYPE="text" NAME="phoneno"/>
<BR> <BR>
<input type="submit" value="Register"/>
</FORM>
</body>
</html>
```

MyServlet.java

MyServlet.java accepts requests from login.html and redirects the user to welcome.html or register.html based on the verification of username & password provided. Username & password are compared with fix values in this example, however you can verify these from database or from a text file etc.

```
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class MyServlet extends HttpServlet {

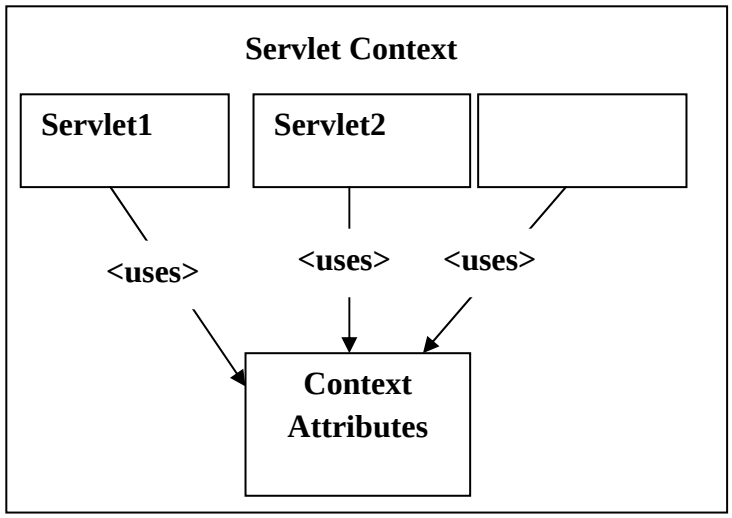
// Handles the HTTP GET request type
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
processRequest(request, response);
}
// Handles the HTTP POST request type
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
```

```
processRequest(request, response);
}
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
String id = request.getParameter("userid");
String pwd = request.getParameter("pwd");
// comparing id & password with fix values
if(id.equals("ali") && pwd.equals("vu")) {
// redirectign user to welcome.html
response.sendRedirect("welcome.html");
}
else {
// redirecting user to register.html
response.sendRedirect("register.html");
/* if you want to display an error message to the
user, you can use the following method
response.sendError( response.SC_PROXY_AUTHENTICATION_REQUIRED, "Send Error Demo" );
*/
} // end else
}
```

29.2 ServletContext

ServletContext belongs to one web application. Therefore it can be used for *sharing* resources among servlets in the same web application.

As initialization parameters, for a single servlet are stored in ServletConfig, ServetContext can store initialization parameters for the entire web application. These parameters are also called context attributes and exist for the lifetime of the application. The following figure illustrates the sharing of context attributes among all the servlets of a web application.



Note:

- There is a single ServletContext per web application
- Different Sevlets will get the same ServletContext object, when calling getServletContext() during different sessions

29.3 Request Dispatcher

RequestDispatcher provides a way to forward or include data from another source. The method getRequestDispatcher(String path) of ServletContext returns a RequestDispatcher object associated with the resource at the given path passed as a parameter.

Two important methods of RequestDispatcher are:

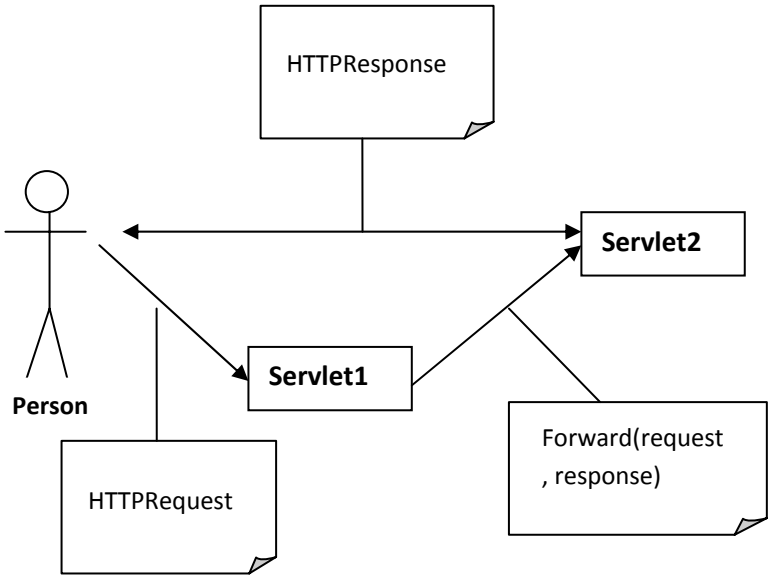
- forward(ServletRequest req, ServletResponse resp)
- include(ServletRequest req, ServletResponse resp)

29.4 RequestDispatcher: forward

Characteristics of forward methods are:

- It allows a Servlet to forward the request to another resource (Servlet, JSP or HTML file) in the same Servlet context.
- Forwarding remains transparent to the client unlike res.sendRedirect(String location). You can not see the changes in the URL.
- Request Object is available to the called resource. In other words, it remains in scope.
- Before forwarding request to another source, headers or status codes can be set, but *output content cannot be added.*

To clarify the concepts, lets take the help from following figure. User initates the request to servlet1. servlet1 forwards the request to servlet2 by calling forward(request, response). Finally a response is returned back to the user by servlet2.

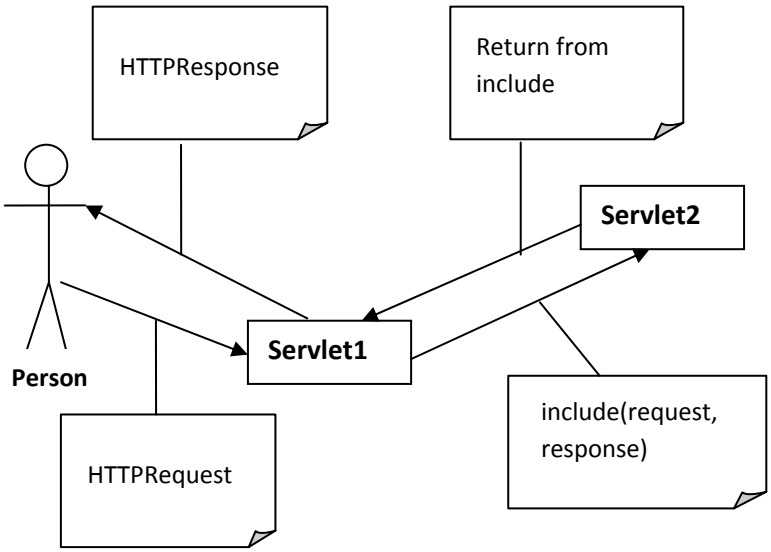


29.5 RequestDispatcher: include

It allows a Servlet to include the results of another resource in its response. The two major differences from forward are:

- Data can be written to the response before an include
- The first Servlet which receive the request, is the one which finishes the response

It will be more cleared from the following figure. User sends a HTTPRequest to Servlet1. Serlet2 is called by Servlet1 by using include(request, response) method. The response generated by Servlet2 sends back to Servlet1. Servlet1 can also add its own response content and finally send it back to user.



29.6 References:

- Java A Lab Course by Umair Javed
- Core Servlets and JSP by Marty Hall
- Java API documentation

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 30: Dispatching Requests

In this handout we will start with request dispatching techniques and give some examples related to that. Further more some methods of HttpResponse and HttpRequest will also be discussed. Finally, this handout will be concluded by discussing the importance of session racking. Before starting, let's take a look at the summery of the previous lecture.

30.1 Recap

In the previous lecture we had some discussion about Response Redirection and Request Dispatcher. We said that Response Redirection was used to redirect response of the Servlet to another application resource. This resource might be another Servlet or any JSP page.

Two forms of Response redirection were discussed. These were:

30.1.1 Sending a standard request:

Using response.sendRedirect("path of resource") method, a new request is generated which redirects the user to the given URL. If the URL is of another servlet, that second servlet will not be able to access the original request object.

30.1.2 Redirection to an error page:

An error code is passed as a parameter along with message to response.sendError(int, msg) method. This method redirects the user to the particular error page in case of occurrence of specified error.

Similarly request dispatching provides us the facility to forward the request processing to another servlet, or to include the output of another resource (servlet, JSP or HTML etc) in the response. Unlike Response Redirection, request object of calling resource is available to called resource. The two ways of Request Dispatching are:

30.1.3 Forward:

Forwards the responsibility of request processing to another resource.

30.1.4 Include:

Allows a servlet to include the results of another resource in its response. So unlike forward, the first servlet to receive the request is the one which finishes the response.

Example Code: Request Dispatching - include

Lets start with the example of include. We will see how a Servlet includes the output of another resource in its response. The following example includes a calling Servlet MyServlet and Servlet IncludeServlet, who's output will be included in the calling Servlet.

The code of MyServlet.java servlet is given below.

MyServlet.java

```
import java.io.*;
 import java.net.*;
 import javax.servlet.*;
import javax.servlet.http.*;

public class MyServlet extends HttpServlet {

/* this method is being called by both doGet() and doPost().We usually follow this practice, when we
are not sure about the type of incoming request to the servlet. So the actual processing is being done in
the processRequest().
*/
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();

out.println("<html>");
out.println("<body>");
out.println("<h1>Start of include request </h1>");
out.flush();

// getting the object of ServletContext, that will be used to
// obtain the object of RequestDispacther
ServletContext context = getServletContext();

// getting the object of RequestDispatcher by passing the path
// of included resource as a parameter
RequestDispatcher rd = context.getRequestDispatcher("/includeservlet");

// calling include method of RequestDispatcher by passing
// request and response objects as parameters. This will execute
//the second servlet and include its output in the first servlet
rd.include(request, response);

/* the statements below will be executed after including the output of the /includeservlet */

out.println("<h1>End of include request </h1>");
out.println("</body>");
out.println("</html>");
```

```
// closing PrintWriter stream out.close();
}

// This method only calls processRequest()
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
processRequest(request, response);
}

// This method only calls processRequest()
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
processRequest(request, response);
}

} // end MyServlet
```

Include Servlet

Now let's take a look at the code of IncludeServlet.java

```
import java.io.*;
 import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class IncludeServlet extends HttpServlet {
// this method is being called by both doGet() and doPost()
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
// Obtaining the object of PrintWriter, this will return the
// same PrintWriter object we have in MyServlet
PrintWriter out = response.getWriter();
// Including a HTML tag using PrintWriter
out.println("<h1> <marquee>I am included </marquee></h1>");

}
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
processRequest(request, response);
}

protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
processRequest(request, response);
}
```

```
} // end IncludeServlet
```

In the processRequest(), firstly we get the PrintWriter stream from the HttpServletResponse object. Then we include an HTML tag to the output of the calling servlet. One thing that must be considered is that PrintWriter stream is not closed in the end, because it is the same stream that is being used in the calling servlet and this stream may also be used in the calling servlet again. So, if it is closed over here, it can not be used again in the calling servlet.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
<servlet>
<servlet-name>MyServlet</servlet-name>
<servlet-class>MyServlet</servlet-class>
</servlet>
<servlet>
<servlet-name>IncludeServlet</servlet-name>
<servlet-class>IncludeServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>MyServlet</servlet-name>
<url-pattern>/myservlet</url-pattern>
</servlet-mapping>
<servlet-mapping>
<servlet-name>IncludeServlet</servlet-name>
<url-pattern>/includeservlet</url-pattern>
</servlet-mapping>
</web-app>
```

Code Example: Request Dispatcher – forward

As discussed earlier, we can forward the request processing to another resource using forward method of request dispatcher. In this example, the user enters his/her name and salary on the index.htmland submits the form to FirstServlet,which calculates the tax on salary and forwards the request to another servlet for further processing i.e. SecondServlet.

index.html

```
<html>
<body>
<form method="POST" ACTION = “firstservlet" NAME="myForm">
<h2> Enter your name</h2>
<INPUT TYPE="text" name="name"/>
<br/>
<h2> Salary</h2>
<INPUT TYPE="text" name="salary"/>
```

```
<BR/><BR/>
<INPUT type="submit" value="Submit"/>
</form>
</body>
</html>
```

FirstServlet.java

```
import java.io.*;
import java.net.*;
 import javax.servlet.*;
import javax.servlet.http.*;

public class FirstServlet extends HttpServlet {
// this method is being called by both doGet() and doPost()
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

// getting value of salary text filed of the HTML form
String salary = request.getParameter("salary");

// converting it to the integer.
int sal = Integer.parseInt(salary);
// calculating 15% tax
int tax = (int)(sal * 0.15);
// converting tax into string
String taxValue = tax + "";
// request object can store values in key-value form, later it
// can be retrieved by using getAttribute() method
request.setAttribute("tax", taxValue);

// getting object of servletContext
ServletContext sContext = getServletContext();
// getting object of request dispatcher
RequestDispatcher rd = sContext.getRequestDispatcher("/secondservlet");

// calling forward method of request dispatcher
rd.forward(request, response);
}

// This method is calling processRequest()
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
processRequest(request, response);
}
// This method is calling processRequest()
```

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
processRequest(request, response);
}
}
```

Note: It the case of Forward, it is illegal to make the reference of PrintWriter stream in the calling Servlet. Only the called resource can use PrintWriter stream to generate response

SecondServlet.java

```
import java.io.*;
import java.net.*;
 import javax.servlet.*;
import javax.servlet.http.*;
public class SecondServlet extends HttpServlet {
// this method is being called by both doGet() and doPost()
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
// obtaining values of name and salary text fields of index.html
String name = request.getParameter("name");
String salary = request.getParameter("salary");

/* getting attribute value that has been set by the calling servlet i.e. FirstServlet */
String tax = (String)request.getAttribute("tax");

// generating HTML tags using PrintWriter
out.println("<html>");
out.println("<head>");
out.println("<title>SecondServlet</title>");
 out.println("</head>");
out.println("<body>");
out.println("<h1> Welcome " + name+ "</h1>");

out.println("<h3> Salary " + salary+ "</h3>");
out.println("<h3> Tax " + tax+ "</h3>");
out.println("</body>");
out.println("</html>");
 out.close();
}
// This method is calling processRequest()
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
```

```
processRequest(request, response);
}
// This method is calling processRequest()
 protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
processRequest(request, response);
}
}
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
<servlet>
<servlet-name>FirstServlet</servlet-name>
<servlet-class>FirstServlet</servlet-class>
</servlet>
<servlet>
<servlet-name>SecondServlet</servlet-name>
<servlet-class>SecondServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>FirstServlet</servlet-name>
<url-pattern>/firstservlet</url-pattern>
</servlet-mapping>
<servlet-mapping>
<servlet-name>SecondServlet</servlet-name>
<url-pattern>/secondservlet</url-pattern>
</servlet-mapping>
</web-app>
```

30.2 HttpServletRequest Methods

Let's discuss some methods of HttpServletRequestclass

30.2.1setAttribute(String, Object)

We can put any object to the context using setAttribute() method in the key-value pair form.. These attributes are also set or reset between requests. These are often used in conjunction with Request Dispatcher. This has also been illustrated in the above example. These attributes are available every where in the same web application so that any other Servlet or JSP resource can access them by using getAttribute() method.

30.2.2 getAttribute(String)

The objects set by the setAttribute() method can be accessed using getAttribute() method. Passing the key in the form of string as a parameter to this method will return the object associated with that particular key in the context. Cast the object into its appropriate type.

30.2.3 getMethod()

This method returns the name of HTTP method which was used to send the request. The two possible returning values could be, get or post.

30.2.4 getRequestURL()

It can be used to track the source of Request. It returns the part of the request's URL with out query string.

30.2.5 getProtocol()

It returns the name and version of the protocol used.

30.2.6getHeaderNames()

It returns the enumeration of all available header names that are contained in the request.

30.2.7getHearderName()

It takes a String parameter that represents the header name and returns that appropriate header. Null value is returned if there is no header exists with the specified name.

30.3 HttpServletResponse Methods

Let's discuss some methods of HttpServletResponseclass

30.3.1setContentType()

Almost every Servlet uses this header. It is used before getting the PrintWriter Stream. It is used to set the Content Type that the PrintWriter is going to use. Usually we set "text/html", when we want to send text output or generate HTML tags on the client's browser.

30.3.2setContentLength()

This method is used to set the content length. It takes length as an integer parameter.

30.3.3addCookie()

This method is used to add a value to the Set-Cookie header. It takes a Cookie object as a parameter and adds it to the Cookie-header. We will talk more about Cookies in the session

tracking part.

30.3.4sendRedirect()

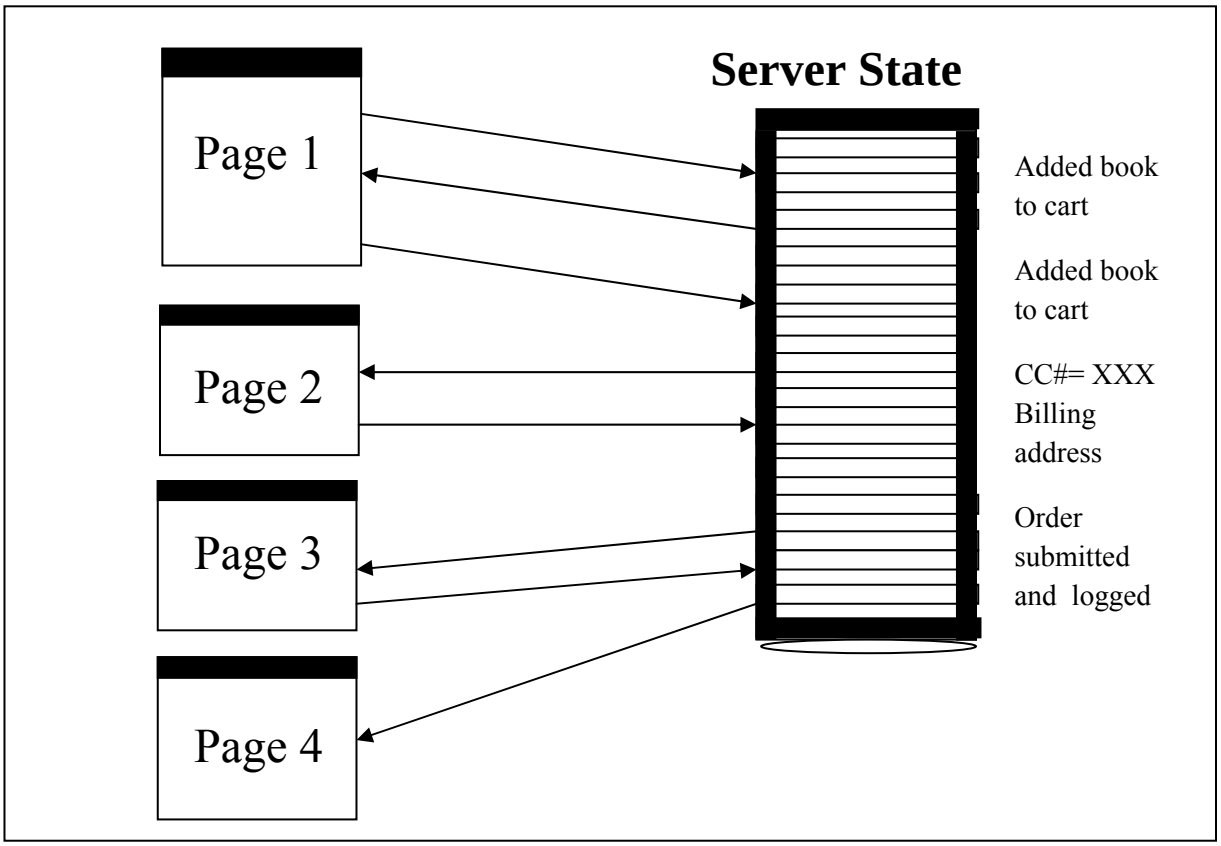
This method redirects the user to the specific URL. This method also accepts the relative URL. It takes URL string as parameter and redirects the user to that resource.

30.4 Session Tracking

Many applications require a series of requests from the same client to be associated withone another. For example, any online shopping application saves the state of a user's shopping cart across multiple requests. Web-based applications are responsible for maintaining such state, because HTTP protocol is stateless. To support applications that need to maintain state, Java Servlet technology provides an API for managing sessions and allows several mechanisms for implementing sessions.

Before looking inside the session tracking mechanism lets see the limitation of HTTP protocol to get the real picture of problems that can happen with out maintaining thesession.

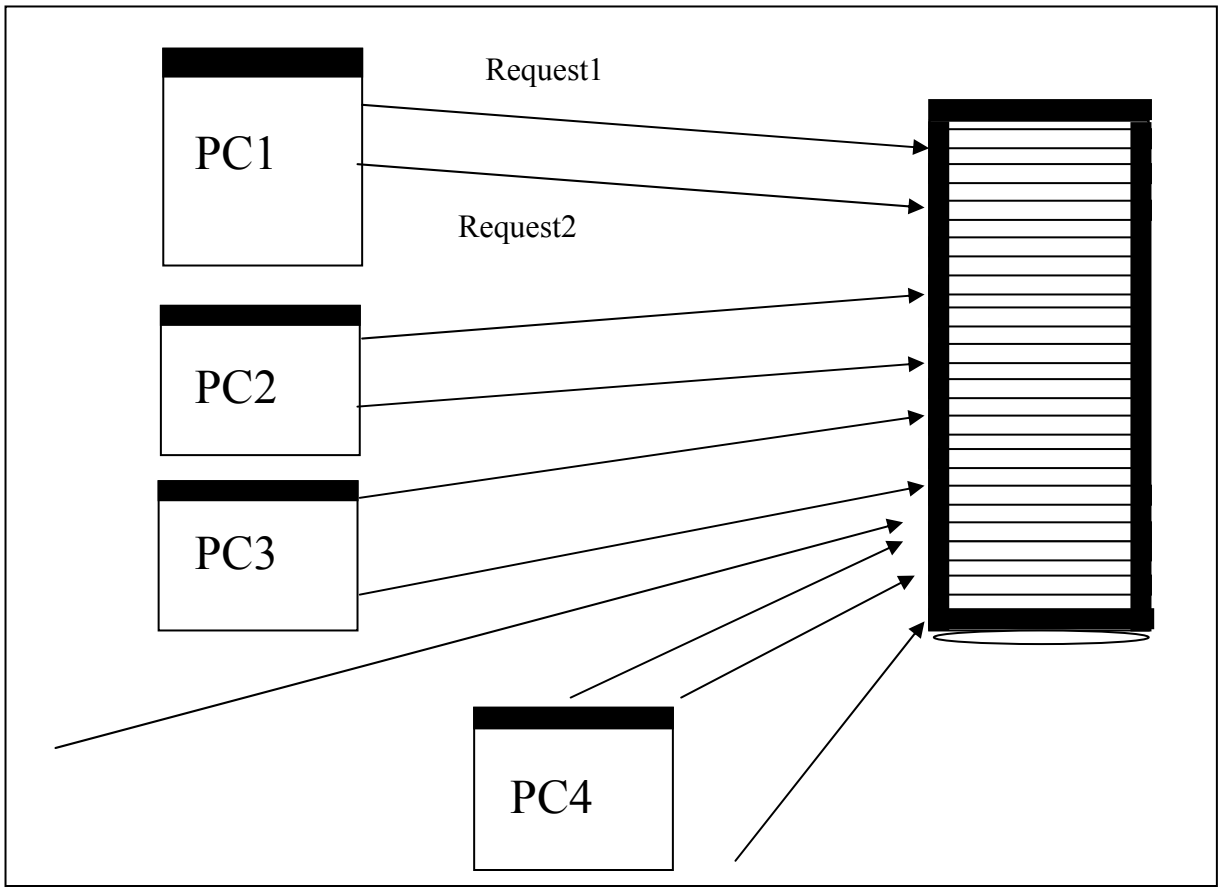
30.4.1Continuity problem- user's point of view



Suppose a user logs on to the online bookshop, selects some books and adds them to his cart. He

enters his billing address and finally submits the order. HTTP cannot track session as it is stateless in nature and user thinks that the choices made on page1 are remembered on page3.

30.4.2 Continuity problem- Server's point of view



The server has a very different point of view. It considers each request independent from other even if the requests are made by the same client.

30.5 References:

- Java A Lab Course by Umair Javed
- Core Servlet and JSP by Marty Hall

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 31: Session Tracking

We have discussed the importance of session tracking in the previous handout. Now, we'll discover the basic techniques used for session tracking. Cookies are one of these techniques and remain our focus in this handout. Cookies can be used to put small information on the client's machine and can be used for various other purposes besides session tracking. An example of simple "Online Book Store", using cookies, will also be surveyed.

As mentioned elsewhere, HTTP is a stateless protocol. Every request is considered independent of every other request. But many applications need to maintain a conversational state with the client. A shopping cart is a classical example of such conversational state.

31.1 Store State Somewhere

To maintain the conversational state, the straightforward approach is to store the state. But where? These states either can be stored on server or on client. However, both options have their merits and demerits. Let's cast a glance on these options:

Storing state on server side makes server really complicated as states needed to be stored for each client. Some one can imagine how much space and processing is required in this scenario as some web servers are hit more than hundred times in a second. E.g. Google, Yahoo etc.

What if states are stored on client side in order to maintain a conversation? Do all the clients permit you doing that? What if client (user) wiped out these states from the machine?

Concluding this discussion, state is stored neither completely on server side nor on client. States are maintained by the mutual cooperation of both client & server. Generally modern servers give the capability to store state on the server side and some information (e.g. client ID/state ID) passed from the client will relate each client with its corresponding state.

31.2 Post-Notes

In order to maintain the conversational state, server puts little notes (some text, values etc) on the client slide. When client submits the next form, it also *unknowingly* submits these little notes. Server reads these notes and able to recall who the client is.

31.3 Three Typical Solutions

Three typical solutions come across to accomplish session tracking. These are:

1. Cookies

2. URL Rewriting
3. Hidden Fields

31.3.1 Cookies

31.3.1.1 What a cookie is?

Don't be tempted? These are not, what you might be thinking off. In fact, in computer terminology, *"a cookie is a piece of text that a web server can store on a client's(user)hard disk".*

Cookies allow the web sites to store information on a client machine and later retrieve it. The pieces of information are stored as *name-value* pair on the client. Later while reconnecting to the same site (or same domain depending upon the cookie settings), client returns the same *name-value* pair to the server.

31.3.1.2 Cookie's Voyage

To reveal the mechanism of cookies, let's take an example. We are assuming here that the web application we are using will set some cookies

- If you type URL of a Web site into your browser, your browser sends a request for that web page
  - For example, when you type www.amazon.com a request is send to the Amazon's server
- Before sending a request, browser looks for cookie files that *amazon* has set
  - If browser finds one or more cookie files related to amazon, it will send it along with the request
  - If not, no cookie data will be sent with the request
- *Amazaon* web server receives the request and examines the request for cookies If cookies are received, *amazon* can use them
  - If no cookie is received, *amazon* knows that you have not visited before or the cookies that were previously set got expired.
  - Server creates a new cookie and send to your browser in the header of HTTP Response so that it can be saved on the client machine.

31.3.2 Potential Uses of Cookies

Whether cookies have more pros or cons is arguable. However, cookies are helpful in the following situations

- identifying a user during an e-commerce session. For example, this book is added into shopping cart by this client.
- Avoiding username and password as cookies are saved on your machine

- customizing a site. For example, you might like email-inbox in a different look form others. This sort of information can be stored in the form of cookies on your machine and latter can be used to format inbox according to your choice.
- Focused Advertising. For example, a web site can store information in the form of cookies about the kinds of books, you mostly hunt for.

31.3.3Sending Cookies to Browser

Following are some basic steps to send a cookie to a browser (client).

1. Create a Cookie Object

A cookie object can be created by calling the Cookieconstructor, which takes two strings: the cookie name and the cookie value.

```
Cookie c = new Cookie (“name”, “value”);
```

2. Setting Cookie Attributes

Before adding the cookie to outgoing headers (response), various characteristics of the cookie can be set. For example, whether a cookie persists on the disk or not. If yes then how long.

A cookies by default, lasts only for the current user session (i.e. until the user quits the session) and will not be stored on the disk.

Using setMaxAge(int lifetime)method indicates how much time (in seconds) should elapse before the cookie expires.

```
c.setMaxAge(60); // expired after one hour
```

3. Place the Cookie into HTTP response

After making changes to cookie attributes, the most important and unforgettable step is to add this currently created cookie into response. If you forget this step, no cookie will be sent to the browser.

```
response.addCookie(c);
```

31.3.4Reading Cookies from the Client

To read the cookies that come back from the client, following steps are generally followed.

1. Reading incoming cookies

To read incoming cookies, get them from the request object of the HttpServeltRequestby calling following method

```
Cookie cookies[] = request.getCookies();
```

This call returns an array of Cookiesobject corresponding to the name & values that came in the HTT P request header.

2. Looping down Cookies Array

Once you have an array of cookies, you can iterate over it. Two important methods of Cookie class are getName() & getValue(). These are used to retrieve cookie name and value respectively.

```
// looping down the whole cookies array
 for(int i=0; i<cookies.length; i++) {
// getting each cookie from the array
Cookie c = cookies[i];
// in search for particular cookie
if( c.getName().equals("someName") {

/* if found, you can do something with cookie or with the help of cookie.
If don't want to process further, loop can also be stopped using
break statement
*/
}
} // end for
```

Example Code1: Repeat Visitor

In the example below, servlet checks for a unique cookie, named "repeat". If the cookie is present, servlet displays *"Welcome Back"*. Absence of cookie indicates that the user is visiting this site for the first time thus servlet displays a message *"Welcome Aboard"*.

This example contains only one servlet "RepeatVisitorServlet.java"and its code is given below. A code snippet of web.xml is also accompanied.

Note: As a *reminder*, all these examples are built using netBeans4.1. This IDE will write web.xml for you. However, here it is given for your reference purpose only, or for those which are not using any IDE to strengthen their concepts

RepeatVisitorServlet.java

```
import java.io.*;
import java.net.*;
 import javax.servlet.*;
import javax.servlet.http.*;
public class RepeatVisitorServlet extends HttpServlet {
// Handles the HTTP <code>GET</code> method.
 protected void doGet(HttpServletRequest request,HttpServletResponse response)
throws ServletException, IOException
{
processRequest(request, response);
}
// Handles the HTTP <code>POST</code> method.
 protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
processRequest(request, response);
}
// called from both doGet() & doPost()
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
response.setContentType("text/html");
PrintWriter out = response.getWriter();
// writing html
out.println("<html>");
out.println("<body>");
out.println("<h2>Cookie Example </h2>");
String msg = "";
boolean repeatVisitor = false;
// reading cookies
Cookie[] cookies = request.getCookies();
// if cookies are returned from request object
if (cookies != null) {
//search for cookie -- repeat
for (int i = 0; i < cookies.length; i++) {
// retrieving one cookie out of array
Cookie c = cookies[i];
// retrieving name & value of the cookie
String name = c.getName();
String val = c.getValue();
// confirming if cookie name equals “repeat” and
// value equals “yes”
if( name.equals("repeat") && val.equals("yes"))
{
msg= "Welcome Back"; repeatVisitor = true; break;
}
```

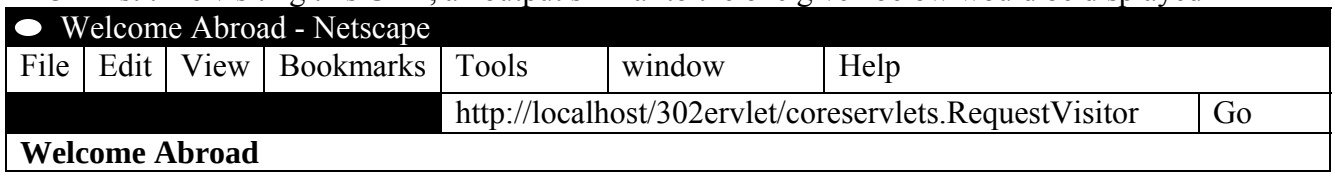
```
} // end for
} // end if
// if no cookie with name “repeat” is found
if (repeatVisitor == false)
{
// create a new cookie
Cookie c1 = new Cookie("repeat", "yes");
// setting time after which cookies expires
c1.setMaxAge(60);
// adding cookie to the response
response.addCookie(c1);
msg = "Welcome Aboard";
}
// displaying message value out.println("<h2>" + msg + "</h2>");
out.println("</body>");
out.println("</html>");
out.close();
}
}// end RepeatVisitorServlet
```

web.xml

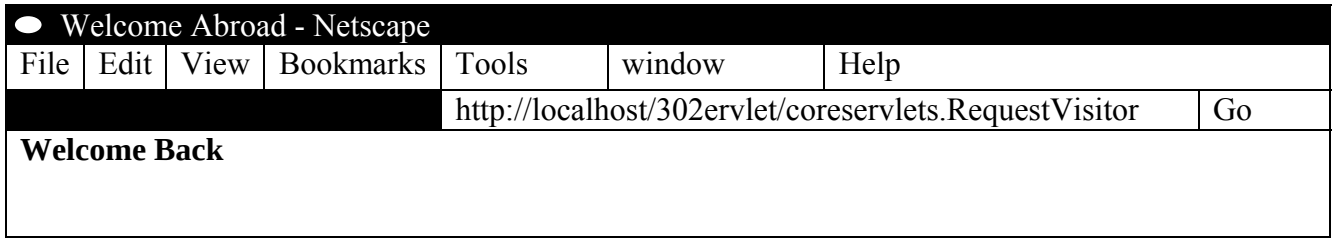
```
<?xml version="1.0" encoding="UTF-8"?> <web-app>
<servlet>
<servlet-name> RepeatVisitorServlet </servlet-name>
<servlet-class> RepeatVisitorServlet </servlet-class> </servlet>
<servlet-mapping>
<servlet-name> RepeatVisitorServlet </servlet-name>
<url-pattern> /repeatexample </url-pattern>
</servlet-mapping>
</web-app>
```

Output

On first time visiting this URL, an output similar to the one given below would be displayed



On refreshing this page or revisiting it within an hour (since the age of cookie was set to 60 mins), following output should be expected.

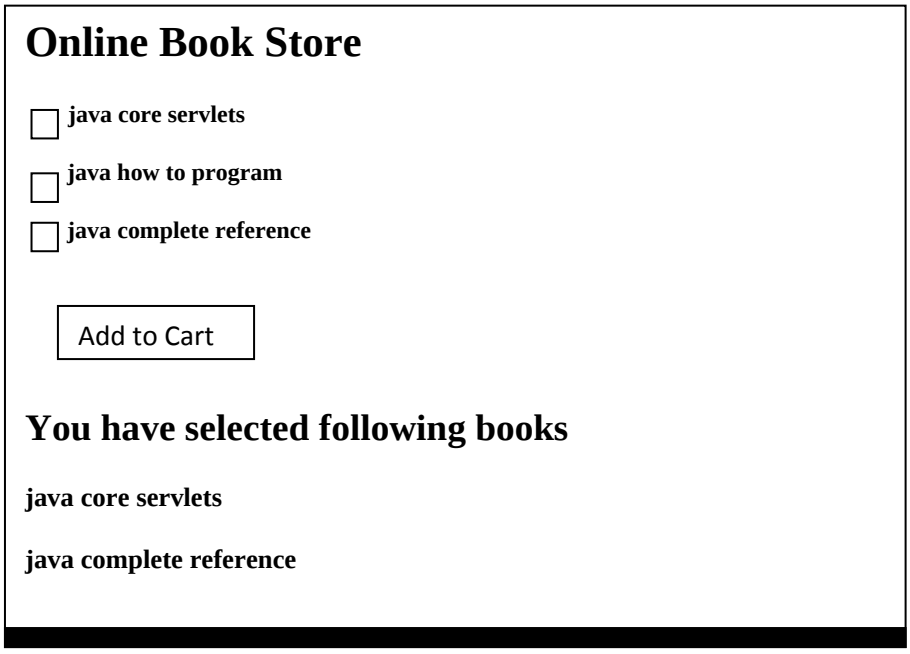


Example Code2: Online Book Store using cookies

A scale down version of online book store is going to be built using cookies. For the first time, cookies will be used to maintain the session.

Three books will be displayed to the user along with check boxes. User can select any check box to add the book in the shopping cart. The heart of the application is, it remembers the books previously selected by the user.

The following figure will help you understand the theme of this example. Books displayed under the heading of "You have selected the following books" were added to cart one after another. The important thing is server that remembers the previously added books by the same user and thus maintains the session. Session management is accomplished using cookies.



Online Book Store example revolves around one ShoppingCartServlet.java. This Servlet has one global HashMap (globalMap) in which HashMap of individual user (sessionInfo) are going to be stored. This (sessionInfo) HashMap stores the books selected by the user.

What's the part of cookies? Cookie (named JSESSIONID, with unique value) is used to keep the unique sessionID associated with each user. This sessionID is passed back and forth between user and the server and is used to retrieve the HashMap (sessionInfo) of the user from the global HashMap at the server. It should be noted here that, HashMaps of individual users are stored in a global HashMap against a sessionID.

ShoppingCartServlet.java

```
import java.io.*;
```

```
import java.net.*;
 import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
public class ShoppingCartServlet extends HttpServlet {
// used to generate a unique value which is
// used as a cookie value
public static int S_ID = 1;
// used to store HashMaps of indiviual users
public static HashMap<String, HashMap> globalMap =<String, HashMap> new HashMap();

// Handles the HTTP GET method.
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
processRequest(request, response);
}
// Handles the HTTP <code>POST</code> method.

protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
processRequest(request, response);
}
// called from both doGet() & doPost()
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
response.setContentType("text/html;charset=UTF-8");
// declaring user's HashMap
HashMap<String, String> sessionInfo = null;
String sID = "";

// method findCookie is used to determine whether browser
// has send any cookie named "JSESSIONID"
Cookie c = findCookie(request);

// if no cookies named "JSESSIONID" is recieved, means that
// user is visiting the site for the first time.
if (c == null) {
// make a unique string
sID = makeUniqueString();
// creating a HashMap where books selected by the
// user will be stored
sessionInfo = new HashMap<String, String>();
// add the user's HashMap (sessionInfo) into the
// globalMap against unique string i.e. sID
```

```
globalMap.put(sID, sessionInfo);
// create a cookie named "JSESSIONID" alongwith
// value of sID i.e. unique string
Cookie sessionCookie = new Cookie("JSESSIONID", sID);
// add the cookie to the response
response.addCookie(sessionCookie);

} else {

// if cookie is found named "JSESSIONID",
// retrieve a HashMap from the globalMap against
// cookie value i.e. unique string which is your
//sessionID
sessionInfo = (HashMap<String, String>) globalMap.get(
c.getValue() );
}

PrintWriter out = response.getWriter();

out.println("<html>");
out.println("<head>");
out.println("<title>Shooping Cart Example</title>");
out.println("</head>");

out.println("<body>");
out.println("<h1>Online Book Store</h1>");

String url = "http://localhost:8084/cookiesessionex/shoppingcartex";
// user will submit the from to the same servlet out.println("<form action=" + url +">" +
"<h3><input type=checkbox name=firstCB value=firstCB />" +
" java core servlts</h3>" + "<br>"+

“<h3><input type=checkbox name=secondCB value=secondCB />" + " java how to program</h3>" +
"<br>"+

"<h3><input type=checkbox name=thirdCB value=thirdCB />" + " java complete reference</h3>" +
"<br>"+

"<input type=submit value=\"Add to Cart\" />" + "</from>"
);

out.println("<br/>");
out.println("<h1>You have selected followig books</h1>");
out.println("<br/>");

//reteriving params of check boxes
```

```
String fBook = request.getParameter("firstCB");
String sBook = request.getParameter("secondCB"); String tBook = request.getParameter("thirdCB");

// if first book is selected then add it to
// user's HashMap i.e. sessionInfo
if ( fBook != null && fBook.equals("firstCB") ) {
sessionInfo.put("firstCB", "java core servlets");
}

// if second book is selected then add it to
// user's HashMap i.e. sessionInfo
if (sBook != null && sBook.equals("secondCB")){
sessionInfo.put("secondCB", "java how to program");
}
// if third book is selected then add it to
// user's HashMap i.e. sessionInfo
if (tBook != null && tBook.equals("thirdCB")){
sessionInfo.put("thirdCB", "java complete reference");
}
// used to display the books currently stored in
// the user's HashMap i.e. sessionInfo printSessionInfo(out, sessionInfo); out.println("</body>");
out.println("</html>");
out.close();
}          // end processRequest()
// method used to generate a unique string
public String makeUniqueString(){
return "ABC" + S_ID++;
}
// returns a reference global HashMap.
public static HashMap findTableStoringSessions(){
return globalMap;
}
// method used to find a cookie named "JSESSIONID" public Cookie findCookie(HttpServletRequest
request){

Cookie[] cookies = request.getCookies();
if (cookies != null) {

for(int i=0; i<cookies.length; i++) { Cookie c = cookies[i];
if (c.getName().equals("JSESSIONID")){
// doSomethingWith cookie
return c;
}
}
}
return null;
```

```
}
// used to print the books currently stored in
// user's HashMap. i.e. sessionInfo
public void printSessionInfo(PrintWriter out,
HashMap sessionInfo)
{
String title = "";
title= (String)sessionInfo.get("firstCB");
if (title != null){
out.println("<h3> "+ title +"</h3>");
}
title= (String)sessionInfo.get("secondCB");
if (title != null){
out.println("<h3> "+ title +"</h3>");
}
title= (String)sessionInfo.get("thirdCB");
if (title != null){
out.println("<h3> "+ title +"</h3>");
}
}
} // end ShoppingCartServlet
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?> <web-app>
<servlet>
<servlet-name> ShoppingCart </servlet-name>
<servlet-class> ShoppingCartServlet </servlet-class> </servlet>
<servlet-mapping>
<servlet-name> ShoppingCart </servlet-name>
<url-pattern> /shoppingcartex </url-pattern>   </servlet- mapping>
</web-app>
```

31.4 References:

- Java A Lab Course by Umair Javed
- Core Servlets and JSP by Marty Hall
- Stanford Course - Internet Technologies
- Java API documentation

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 32: Session Tracking 2

In the last handout we have discussed the solutions for session tracking and talked about one important mechanism *cookies* in detail. We said cookies allow the server to store information on a client machine and later retrieve it. Now we will see two more mechanisms that provide us facility to maintain a session between user's requests. These are *URL Rewriting* and *Hidden Form Fields*. After that we will discuss a session tracking API provided by java.

32.1 URL Rewriting

URL rewriting provides another way for session tracking. With URL rewriting, the parameter that we want to pass back and forth between the server and client is appended to the URL. This appended information can be retrieve by parsing the URL. This information can be in the form of:

- Extra path information,
- Added parameters, or
- Some custom, server-specific URL change

Note: Due to limited space available in rewriting a URL, the extra information is usually limited to a unique session ID.

The following URLs have been rewritten to pass the session ID **123**

- Original -http://server: port/servlet /rewrite
- Extra path information -http://server: port/servlet/rewrite/**123**
- Added parameters -http://server: port/servlet/rewrite**?id=123**
- Custom change -http://server: port/servlet/rewrite**;id123**

32.1.1 Disadvantages of URL rewriting

The following Disadvantages of URL rewriting, are considerable: -

- What if the user bookmarks the page and the problem get worse if server is not assigning a unique session id.
- Every URL on a page, which needs the session information, must be rewritten each time page is served, which can cause
    - Computationally expensive
    - Can increase communication overhead

- unlike cookies, state information stored in the URL is not persistent
- this mechanism limits the client interaction with the server to HTTP GET request.

Example Code: Online Bookstore using URL Rewriting

This is the modified version of online book store (selling two books only, however you can add in on your own) that is built using cookies in the last handout. Another important difference is books are displayed in the form of hyperlink instead of check boxes. URL rewriting mechanism is used to maintain session information.

How to make Query String

Before jumping on to example, one important technique is needed to be learned i.e. making on query string. If you ever noticed the URL of a servlet in a browser that is receiving some HTML form values, also contains the HTML fields name with values entered/selected by the user.

Now, if you want to pass some attribute and values along with URL, you can use the technique of query string. Attribute names and values are written in pair form after the **?.** For example, if you want to send attribute "name" and its value "ali", the URL will look like

- Original URL

http://server:port/servletex /register

- After adding parameters

http://server:port/servletex/**register ?name=ali**

If you want to add more than one parameter, all subsequent parameters are separated by **&** sign. For example

- Adding two parameters -

http://server:port/servletex/**register ?name=ali&address=gulberg**

URLRewriteServlet.java :

```
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
```

```
import java.util.*;
public class URLRewriteServlet extends HttpServlet {
// used to generate a unique value which is
// used as a cookie value
public static int S_ID = 1;
// used to store HashMaps of indiviual users
public static HashMap<String, HashMap> globalMap = new HashMap<String, HashMap>();

// Handles the HTTP GET method.
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
processRequest(request, response);
}
// Handles the HTTP <code>POST</code> method.
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
processRequest(request, response);
}

// called from both doGet() & doPost()
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

// declaring user's HashMap
HashMap<String, String> sessionInfo = null;

// reading sessionId
String sID = request.getParameter("JSESSIONID");
/* if parameter JSESSIONID is received, means that user is visiting the site for the first time. */
if (sID ==  null)
{
// make a unique string
sID = makeUniqueString();
// creating a HashMap where books selected by the
// user will be stored
sessionInfo = new HashMap<String, String>();

// add the user's HashMap (sessionInfo) into the
// globalMap against unique string i.e. sID
globalMap.put(sID, sessionInfo);
}else {

// if parameter "JSESSIONID" has some value
```

```
// retrieve a HashMap from the globalMap against
// sID i.e. unique string which is your sessionID
sessionInfo = (HashMap<String, String>) globalMap.get(sID);
}
response.setContentType("text/html;charset=UTF-8");
PrintWriter out = response.getWriter();
out.println("<html>");
out.println("<head>");
out.println("<title>Shopping Cart Example</title>");
out.println("</head>");
out.println("<body>");
out.println("<h1>Online Book Store</h1>");

// Making three URLS by using query string mechanism
// The attributes/parameters are JSSESSIONID and book name (like
// firstCB) along with values sID and book name respectively

String firsturl = "http://localhost:8084/urlbookstore/urlrewriteservlet?JSESSIONID= " + sID
+ "&firstCB=firstCB";
String secondurl = "http://localhost:8084/urlbookstore/urlrewriteservlet?JSESSIONID= " +
sID + "&secondCB=secondCB";
out.println("<h3><a href=" + firsturl + ">" + " java core servlts </a> </h3>" + "<br>" "<h3><a
href=" + secondurl + ">" + " java how to program </a> </h3>" + "<br>" );
out.println("<br/>");
out.println("<h1>You have selected following books</h1>");
out.println("<br/>");
//retrieving params that are emebded in URLs
String fBook = request.getParameter("firstCB"); String sBook = request.getParameter("secondCB");
// if first book is selected then add it to
// user's HashMap i.e. sessionInfo
if ( fBook != null && fBook.equals("firstCB") ) {

sessionInfo.put("firstCB", "java core servlets");
}
// if second book is selected then add it to
// user's HashMap i.e. sessionInfo


if (sBook != null && sBook.equals("secondCB")){
sessionInfo.put("secondCB", "java how to program");
}

// used to display the books currently stored in
// the user's HashMap i.e. sessionInfo printSessionInfo(out, sessionInfo);

out.println("</body>");
```

```
out.println("</html>");

out.close();
} // end processRequest()
// method used to generate a unique string
public String makeUniqueString(){
return "ABC" + S_ID++;
}

// returns a reference global HashMap.
public static HashMap findTableStoringSessions(){
return globalMap;
}
// used to print the books currently stored in
// user's HashMap. i.e. sessionInfo
public void printSessionInfo(PrintWriter out, HashMap sessionInfo)
{
String title = "";
title= (String)sessionInfo.get("firstCB");

if (title != null){
out.println("<h3> "+ title +"</h3>");
}

title= (String)sessionInfo.get("secondCB");
if (title != null){
out.println("<h3> "+ title +"</h3>");
}
}
} // end URLRewriteServlet
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?> <web-app>
<servlet>
<servlet-name>     URLRewriteServlet </servlet-name>
<servlet-class> URLRewriteServlet </servlet-class>         </servlet>
<servlet-mapping>
<servlet-name> URLRewriteServlet </servlet-name>
<url-pattern> /urlrewriteservlet </url-pattern>
</servlet-mapping>
</web-app>
```

32.2 Hidden Form Fields

HTML forms can have an element that looks like the following:

<INPUT TYPE="HIDDEN" NAME="sessionid" VALUE="123" />

Hidden Forms Fields do not affect the appearance of HTML page. They actually contain the information that is needed to send to the server. Thus, hidden fields can also be used to store information (like *sessionid*) in order to maintain session.

```
<form method="POST" action="/exec/obidos/handle-buy-box=/ref = bp_add/103-
4591077-290203">

<input type="hidden" name="colid" value="">
<input type="hidden" name="template-name" value="">
<input type="hidden" name="store-name" value="gateway">
<input type="hidden" name="maw" value="1">
<input type="hidden" name="coliid" value="">

<input type="hidden" name="dropdown-selection" value="default-address">
<table border="0" width="100%" cellspacing="0" cellpadding="6">
```

In the above figure you can see the use of Hidden form fields for storing particular information.

32.3 Java Solution for Session Tracking

Java provides an excellent solution to all the problems that occurred in tracking a session. The Servlet API provides several methods and classes specifically designed to handle session tracking. In other words, servlets have built in session tracking.

Sessions are represented by an HttpSession object. HttpSession tacking API built on top of URL rewriting and cookies. All cookies and URL rewriting mechanism is hidden and most application server uses cookies but automatically revert to URL rewriting when cookies are unsupported or explicitly disabled. Using HttpSession API in servlets is straightforward and involves looking up the session object associated with the current request, creating new session object when necessary, looking up information associated with a session, storing information in a session, and discarding completed or abandoned sessions.

32.4 Working with HttpSession

Let's have a look on HttpSession working step by step.

1. Getting the user's session object

To get the user's session object, we call the *getSession()* method of HttpServeltRequestthat returns the object of *HttpSession*

HttpSession sess = request.getSession(true);

If true is passed to the getSession() method, this method returns the current session associated with this request, or, if the request does not have a session, it

creates a new one. We can confirm whether this session object (sess) is newly created or returned by using isNew() method of HttpSession. In case of passing *false, null* is returned if the session doesn't exist.

2. Storing information in a Session

To store information in Session object (sess), we use setAttribute() method of HttpSession class. Session object works like a HashMap, so it is able to store any java object against key. So you can store number of keys and their values in pair form. For example,

sess.setAttribute("sessionid", "123");

3. Looking up information associated with a Session

To retrieve back the stored information from session object, getAttribute()method of HttpSession class is used. For example,

String sid=(String)sess.getAttribute("sessionid");

Note: - getAttribute() method returns *Object* type, so typecast is required.

4. Terminating a Session

After the amount of time, session gets terminated automatically. We can see its maximum activation time by using getMaxInactiveInterval() method of HttpSession class. However, we can also terminate any existing session manually. For this, we need to call invalidate () method of HttpSession class as shown below.

sess.invalidate()

Example Code: Showing Session Information

To understand HttpSessionAPI properly we need to have a look on an example. In this example, we will get the session object and check whether it is a new user or not. If the user is visiting for the first time, we will print "*Welcome"* and if we find the old one, we'll print "*Welcome Back"*. Moreover, we will print the session information and count the number of accesses for every user

```
import java.io.*;
 import java.net.*;
 import javax.servlet.*;
import javax.servlet.http.*;

public class ShowSessionServlet extends HttpServlet {

// Handles the HTTP <code>GET</code> method.
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
processRequest(request, response);
}

// Handles the HTTP <code>POST</code> method.
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
processRequest(request, response);

}

// called from both doGet() & doPost()
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{

// used for displaying message (like Welcomem, Newcomer) to
// user
```

```
String heading;

response.setContentType("text/html");

// Getting session object
HttpSession session = request.getSession(true);
/* Getting stored information using getAttribute() method */
Integer accessCount = (Integer)session.getAttribute("sessionCount");

/* If user comes for the first time, accessCount will be assigned null, so we can guess easily that this
a new user */

if (accessCount == null)
{
accessCount = new Integer(1);
heading = "Welcome, Newcomer";
} else
{
heading = "Welcome Back";
// Incrementing the value
accessCount = new Integer(accessCount.intValue() + 1);
}
/* Storing the new value of accessCount in the session using setAttribute() method */
session.setAttribute("sessionCount", accessCount);
// Getting the PrintWriter
PrintWriter out = response.getWriter();
/*Generating HTML tags using PrintWriter to print session info and no of times this user has
accessed this page */
out.println("<HTML>" +
" <BODY>" +
" <h1>Session Tracking Example</h1>" +
" <H2>Information on Your Session:</H2>\n" +
" <H3> Session ID: " + session.getId() + "</H3>" +
" <H3>Number of Previous Accesses: " + accessCount +
" </H3>" +
" </BODY>" + " </HTML>"
);

//Closing the PrintWriter stream
out.close();
} // end processRequest
} // end ShowSessionServlet class
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
<servlet>
<servlet-name>     ShowSession   </servlet-name>
<servlet-class> ShowSessionServlet </servlet-class>
</servlet>
<servlet-mapping>
<servlet-name> ShowSession </servlet-name>
<url-pattern> /showsession </url-pattern>
</servlet-mapping>
</web-app>
```

32.5 HttpSession – Behind the scenes

When we call getSession()method, there is a lot going on behind the scenes. For every user, a unique session ID is assigned automatically. As the server deals with lot of users at a time, this ID is used to distinguish one user from another. Now here is the question, how this ID sends to the user? Answer is, there are two options

Option 1: If the browser supports cookies, the Servlet will automatically creates a session cookie and store the session ID within that cookie.

Option 2: If the first option fails because of browser that does not support cookies then the Servlet will try to extract the session ID from the URL

32.6 Encoding URLs sent to Client

Servlet will automatically switch to URL rewriting when cookies are not supported or disabled by the client. When Session Tracking is based on URL rewriting, it requires additional help from the Servlets. For a Servlet to support session tracking via URL rewriting, it has to rewrite (encode) every local URL before sending it to the client. Now see how this encoding works

HttpServletResponseprovides two methods to perform encoding

- String encodeURL(String URL)
- String encodeRedirectURL(String URL)

If Cookies are disabled, both methods encode (rewrite) the specific URL to include the session ID and returns the new URL. However, if cookies are enabled, the URL is returned unchanged.

32.7 Difference between encodeURL() and encodeRedirectURL()

encodeURL() is used for URLs that are embedded in the webpage, that the servlet generates. For example,

String URL = ”/servlet/sessiontracker”; **String eURL = response.encodeURL(URL); out.println(“<A HREF=\” ” + eURL + ”\ ”> …… </A>”);**

Whereas encodeRedirectURL() is used for URLs that refers yours site is in sendRedirect() call. For example,

String URL = ”/servlet/sessiontracker”;
String eURL = response.encodeRedirectURL(URL);
Response.sendRedirect(eURL);

Example Code: OnlineBookStore using HttpSession

This book store is modified version of last one, which is built using URL rewriting mechanism. Here, HttpSessionwill be used to maintain session.

ShoppingCartServlet.java

```
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ShoppingCartServlet extends HttpServlet {
// Handles the HTTP GET method.
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
processRequest(request, response);
}

// Handles the HTTP <code>POST</code> method.
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
processRequest(request, response);
}
// called from both doGet() & doPost()
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
```

```
throws ServletException, IOException {
response.setContentType("text/html;charset=UTF-8");
HttpSession session = request.getSession(true);
PrintWriter out = response.getWriter();
out.println("<html>");
out.println("<head>");
out.println("<title>Shopping Cart Example</title>");
out.println("</head>");
out.println("<body>");
out.println("<h1>Online Book Store</h1>");
// First URL built using query string, representing first book
String firstURL = "http://localhost:8084/urlrewritebookstore/shoppingcart?book=firs t";

// Second URL built using query string, representing second book
// Note that parameter name is still book, so that later we need
//    to read only this parameter
String secondURL = "http://localhost:8084/urlrewritebookstore/shoppingcart?book=seco nd";

// Encoding URLs
String eURL1 = response.encodeURL( firstURL );
String eURL2 = response.encodeURL( secondURL );
out.println(
"<h3><a href=" + eURL1 + ">" +
" java core servlets </a> </h3>" + "<br>"+
"<h3><a href=" + eURL2 + ">" +
" java How to Program </a> </h3>"
);
out.println("<br/>");
out.println("<h1>You have selected following books</h1>");
out.println("<br/>");
//retrieving params that are emebded in URLs
String fBook = request.getParameter("firstCB");
String sBook = request.getParameter("secondCB");

out.println("<br/>");
out.println("<h1>You have selected following books</h1>");
out.println("<br/>");
//retrieving param that is embedded into URL
String book = request.getParameter("book");
if (book != null){
// if firstURL, value of first hyperlink is clicked
// then storing the book into session object against fBook
if (book.equals("first")){
session.setAttribute("fBook", "java core servlets");
}
// if secondURL, value of second hyperlink is clicked
```

```
// then storing the book into session object against sBook
else if(book.equals("second")){
session.setAttribute("sBook", "java how to program");
}
}//outer if ends
// used to display the books currently stored in
// the HttpSession object i.e. session
printSessionInfo(out, session);

out.println("</body>");
out.println("</html>"); out.close();
} // end processRequest()

// used to display values stored in HttpSession object
public void printSessionInfo(PrintWriter out, HttpSession session)
{
String title = "";

// reading value against key fBook from session,
// if exist displays it
title= (String)session.getAttribute("fBook");
if (title != null){
out.println("<h3> "+ title +"</h3>");
}

// reading value against key sBook from session,
// if exist displays it
title= (String)session.getAttribute("sBook");
if (title != null){
out.println("<h3> "+ title +"</h3>");
}
} // end printSessionInfo
} // end ShoppingCartServlet
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?> <web-app>
<servlet>
<servlet-name>      ShoppingCartServlet </servlet-name>
<servlet-class> ShoppingCartServlet </servlet-class>
</servlet>
<servlet-mapping>
<servlet-name> ShoppingCartServlet </servlet-name>
<url-pattern> /shoppingcart </url-pattern>
```

```
</servlet-mapping>
</web-app>
```

32.8 Some Methods of HttpSession

Now let's explore some methods of HttpSessionclass

- **setAttribute(String, Object)**
  - This method associates a value with a name.
- **getAttribute(String)**
  - Extracts previously stored value from a session object. It returns nullif no value is associated with the given name
- **removeAttribute(String)**
  - This method removes values associated with the name
- **getId( )**
  - This method returns the unique identifier of this session
- **getCreationTime( )**
  - This method returns time at which session was first created
- **getMaxInactiveInterval( ) , setMaxInactiveInterval(int)**
  - To get or set the amount of time session should go without access before being invalidated.

32.9 References:

- Java A Lab Course by Umair Javed
- Core Servlets and JSP by Marty Hall
- Stanford Course - Internet Technologies
- Java Tutorial on Servlets
  - http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Servlets11.html
- Java API documentation

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

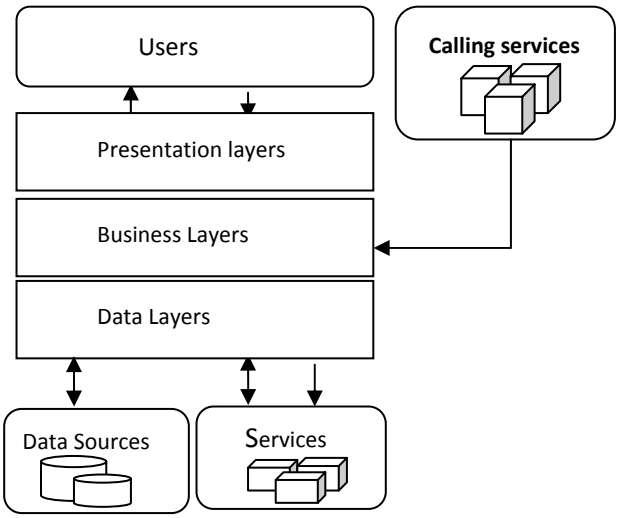
Lecture 33: Address Book Case Study Using Servlets

33.1 Design Process

In this handout, we will discuss the design process of a simple address book. A step by step procedure of creating a simple address book is given below.

33.2 Layers & Web Application

As discussed previously, normally web applications are partitioned into logical layers. Each layer performs a specific functionality which should not be mixed with other layers. For example data access layer is used to interact with database and we do not make any direct calls to database from the presentation layer. Layers are isolated from each other to reduce coupling between them but they provide interfaces to communicate with each other.



Simplified view of a web application and its layers

- **Presentation Layer**
  - Provides a user interface for client to interact with application. This is the only
  - Part of application visible to client.
- **Business Layer**
  - The business or service layer implements the actual business logic or functionality of the application. For example in case of online shopping systems this layer Handles transaction management.

- **Data Layer**
    - This layer consists of objects that represent real-world business objects such as an *Order*, *OrderLineItem*, *Product*, and so on. It also encapsulates classes which are used to interact with the data providing services such as databases, other web services etc.

In our case study of address book, we will also try to make use of the layered architecture. We will create a separate layer for working with data, and our presentation and business logic will be merged into servlets. It means that we will not have separate layers for presentation and business rather one layer (formed by servlets) will do the job of both presentation and business logic. The extent to which you divide your application into layers depends upon the size of the application and some other factors such as scalability, portability etc.

33.2.1Step 1

- Create a database (AddressBook)
- Make a table named Person according to the figure shown below. It has columns name, address, phomeNum

Person : Table

	Name	Address	phoneNum
	Saad	Model town	437546
	Usman	Defence	6342211
	Ali	Gulberg	9201211
*			
Record	I< <	1	> >I Of 3

33.2.2 Step 2

The next step is to create a class that can hold the information of a single person. Remember we have stored the information in the database, now when we extract this information from the database as a result of some search, we will require some object to store the data for that particular person. The PersonInfo class will be used at that point to store the retrieved data and transport it to presentation layer. Also we extend this application and add the functionality of "AddingNewContacts" in the database. The PersonInfoclass can be used to transport data from front end to the database.

- Make a PersonInfoclass with the following consideration

It has three three attributes: name, address, ph.

No.

It has a parameterized constructor which takes in the above mentioned parameters Override the

toString()method:

```
//File: PersonInfo.java
 public class PersonInfo {
String name;
String address;
String phoneNum;
public PersonInfo(String n, String a, String pn) {
name = n;
address = a;
phoneNum = pn;
}
public String toString( ){
return "Name: " + name + " Address: " + address + " Phone No: " + phoneNum;
}
}// end class PersonInfo
```

Note: To keep the code simple, attributes (name, address & phoneNum) are not declared as private, which is indeed not a good programming approach.

33.2.3Step 3

Now we will create a class that will be used to interact with the database for the *search*, *insert*, *update* and *delete* operations. We will call it PersonDAOwhere *DAO* stands for the "data access object". The PersonDAO along with the PersonInfo class forms the data layer of our application. As you can see that these two classes do not contain any code related to presentation or business logic (There is not much of business logic in this application anyway). So PersonDAOalong with PersonInfois used to retrieve and store data in this application. If at some stage we choose to use some other way of storing data (e.g. files) only the PersonDAO class will change and nothing else, which is a sign of better design as compared to a design in which we put everything in a single class.

So, Make a PersonDAOclass which contains:

A searchPerson(String name)method that first establishes a connection to the database and returns PersonInfo object after searching the information of the specified person from the database.

//File: PersonDAO.java

```
import java.sql.*;
```

```
public class PersonDAO {

// method searchPerson
public PersonInfo searchPerson(String sName){
PersonInfo person = null;
try {
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
String url = "jdbc:odbc:AddressBookDSN";
Connection con = DriverManager.getConnection(url);
String sql = "SELECT * FROM Person WHERE name = ?";
PreparedStatement pStmt = con.prepareStatement(sql);
pStmt.setString(1, sName);
ResultSet rs = pStmt.executeQuery();
if (rs.next( ) ) {
String name = rs.getString("name");
String add = rs.getString("address");
String pNum = rs.getString("phoneNum");
person = new PersonInfo(name, add, pNum);
con.close();
}catch(Exception ex){
System.out.println(ex);
}

return person;
}// end method
}
```

33.2.4Step 4

To find what user wants to search, we need to give user an interface through which he/she can enter the input. The SearchPesonServlet.javawill do this job for us, It will collect the data from the user and submit that data to another class. The SearchPersonServletforms the part of our *presentation layer*. As you can see that it is being used to present a form to the user and collect input.

Write SearchPersonServlet.java

Will take input for name to search in address book

Submits the request to ShowPersonServlet

//File: SearchPersonServlet.java

```
import java.io.*;
import java.net.*;
import javax.servlet.*;
```

```
import javax.servlet.http.*;
public class SearchPersonServlet extends HttpServlet {
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println(
"<html>" + "<body>" +
"<h1> Address Book </h1>" + "<form action=showperson >" +
// showperson is alias or
// url pattern of
// ShowPersonServlet
"<h2> Enter name to search </h2> <br/>" + "<input type=text name=pName /> <br/>" + "<input
type=submit value=Search Person />" + "</form>" +
"</body>" + "</html>"
);
out.close();
}
// Handles the HTTP GET method.
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
processRequest(request, response);
}
// Handles the HTTP POST method.
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
processRequest(request, response);
}
}
```

33.2.5Step 5

The data submitted by the SearchPersonServletwill be submitted to another servlet i.e. ShowPersonServlet,which will interact with the DataLayer(Business logic processing) collects the output and show it to the user. The ShowPersonServletforms the part of our presentation layer and business layer. As you can see that it is being used to do processing on the incoming data and giving it to data layer (business layer) and present data/output to the user (presentation layer)

Write ShowPersonServlet.java

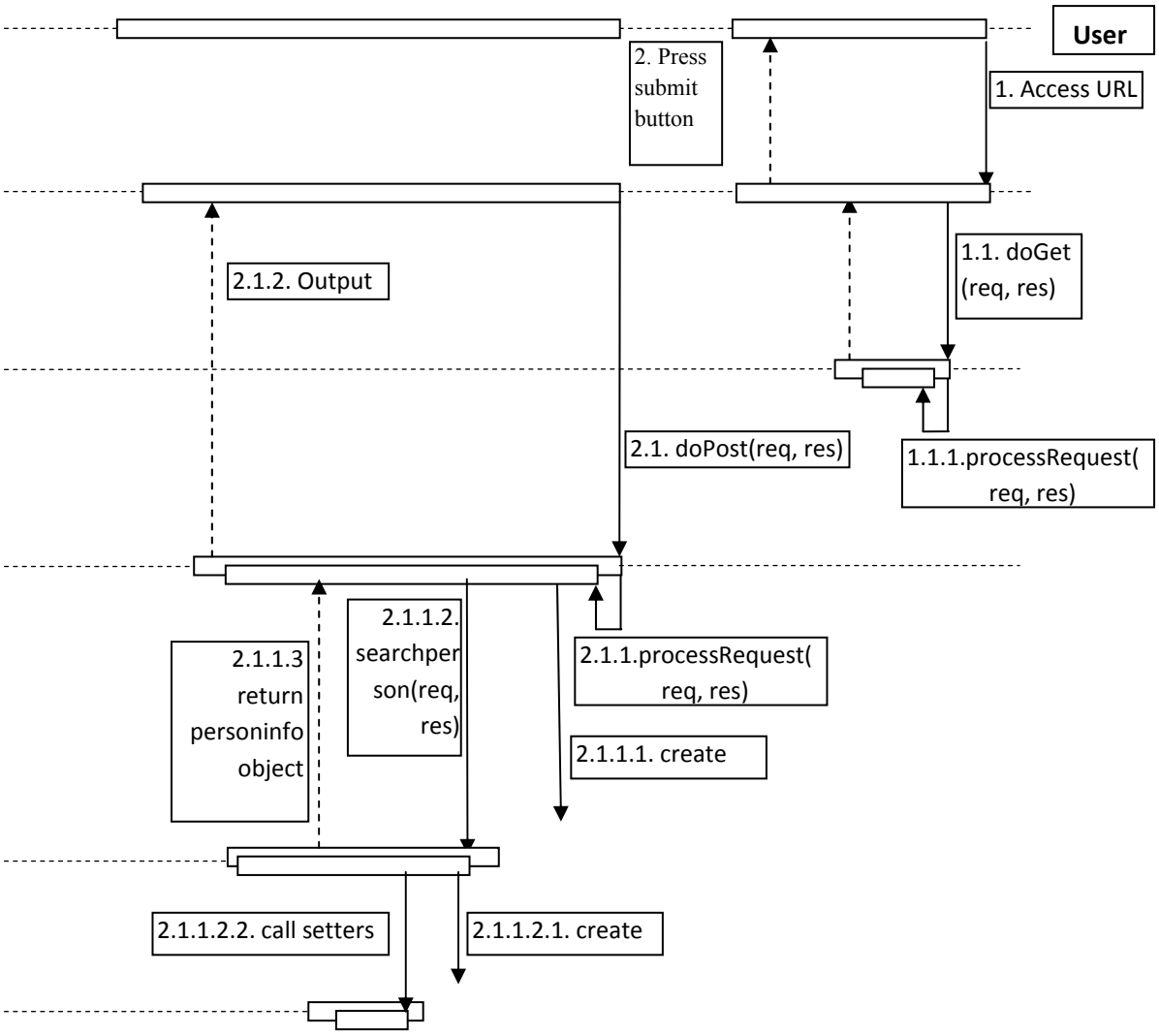
Receives request from SearchPersonServlet

Instantiate objects of PersonInfoand PersonDAOclass

Call searchPerson()method of PersonDAOclass Show results

```
import java.io.*;
import java.net.*;
 import javax.servlet.*;
 import javax.servlet.http.*;
public class ShowPersonServlet extends HttpServlet {
protected void processRequest(HttpServletRequestrequest, HttpServletResponse response)
throws ServletException, IOException{
response.setContentType("text/html");
PrintWriter out = response.getWriter();
String name = request.getParameter("pName");
// creating PersonDAO object, and calling searchPerson() method
PersonDAO personDAO = new PersonDAO();
PersonInfo person = personDAO.searchPerson(name);
out.println("<html>");
out.println("<body>");
out.println("<h1>Search Results</h1>");
if (person != null){
out.println("<h3>"+ person.toString() +"</h3>" );
}
else{
out.println("<h3>Sorry! No records found</h3>" );
}
out.println("</body>"); out.println("</html>"); out.close();
}
// Handles the HTTP GET method.
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
processRequest(request, response);
} // Handles the HTTP POST method.
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
processRequest(request, response);
}
} // end ShowPersonServlet
```

Sequence Diagram: Address Book (search use case)



33.3 Package

Many times when we get a chance to work on a small project, one thing we intend to do is to put all java files into one single directory (folder). It is quick, easy and harmless. However if our small project gets bigger, and the number of files is increasing, putting all these files into the same directory would be a nightmare for us. In java we can avoid this sort of problem by using Packages.

33.3.1 What is a package?

In simple terms, a set of Java classes organized for convenience in the same directory to avoid the name collisions. Packages are nothing more than the way we organize files into different directories according to their functionality, usability as well as category they should belong to. An obvious example of packaging is the JDK package from SUN (java.xxx.yyy) as shown below:

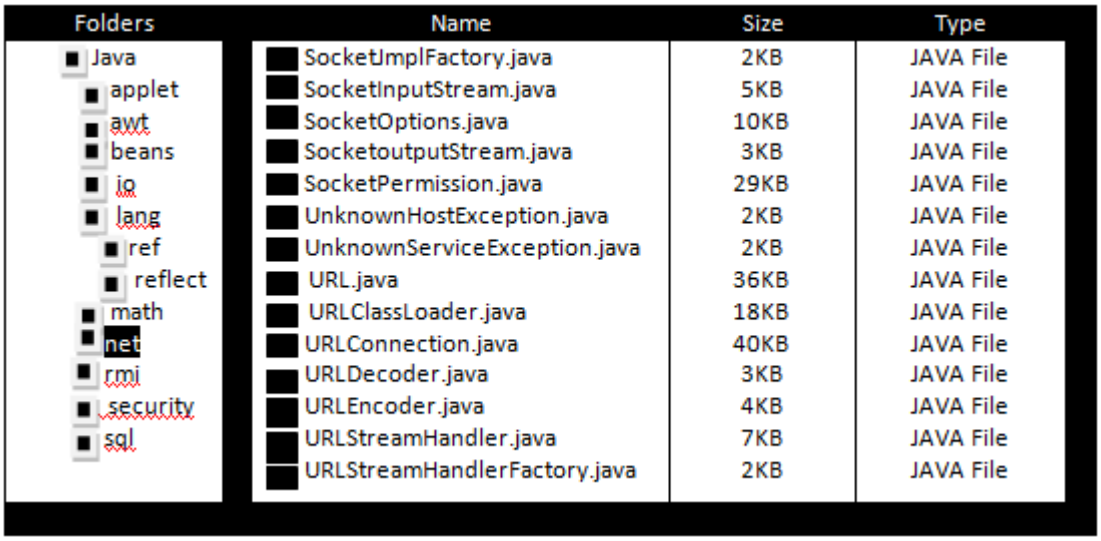


Figure. Basic structure of JDK package

Basically, files in one directory (or package) would have different functionality from those of another directory. For example, files in java.io package do something related to I/O, but files in java.net package give us the way to deal with the Network.

Packaging also helps us to avoid class *name collision* when we use the same class name as that of others. For example, if we have a class name called "ArrayList", its name would crash with the ArrayListclass from JDK. However, this never happens because JDK use

java.utilas a package name for the ArrayList class (java.util.ArrayList). So our ArrayList class can be named as "ArrayList" or we can put it into another package like com.mycompany.ArrayListwithout fighting with anyone. The benefits of using package reflect the ease of maintenance, organization, and increase collaboration among developers. Understanding the concept of package will also help us manage and use files stored in jar files in more efficient ways.

33.3.2How to create a package

Suppose we have a file called HelloWorld.java, and we want to put this file in a package **world**. First thing we have to do is to specify the keyword **package**with the name of the package we want to use (**world**in our case) on top of our source file, before the code that defines the real classes in the package, as shown in our HelloWorldclass below:

```
// only comment can be here package
world;
```

```
public class HelloWorld {
public static void main(String[] args) {
System.out.println("Hello World");
}
```

One thing you must do after creating a package for the class is to create nested subdirectories to represent package hierarchy of the class. In our case, we have the **world**package, which requires only one directory. So, we create a directory (folder) **world**and put our HelloWorld.javainto it.



Figure: HelloWorld in world package (C:\world\HelloWorld.java

33.3.3How to use package

By using "import" keyword, all class files reside only in that package can be imported.
For example,

```
// we can use any public classes inside world package
import world.*;
// import all public classes from java.util package
import java.util.*;

// import only ArrayList class (not all classes in
// java.util package)
import java.util.ArrayList;
```

Note: While working with IDEs, You don't have to create folders (packages) and to place classes at right locations. Many IDEs (like netBeans® 4.1) performs this job on your behalf.

33.4 JavaServer Pages (JSP)

Like Servlets, JSP is also a specification. JSP technology enables Web developers and designers to rapidly develop and easily maintain, information-rich, dynamic Web pages that leverage existing business systems. As part of the Java technology family, JSP technology enables rapid development of Web-based applications that are platform independent. JSP technology separates the user interface from content generation, enabling designers to change the overall page layout without altering the underlying dynamic content.

33.4.1The Need for JSP

With servlets, it is easy to

- Read form data
- Read HTTP request headers
- Set HTTP status codes and response headers
- Use cookies and session tracking
- Share data among servlets
- Remember data between requests
- Get fun, high-paying jobs

But, it sure is a pain to

- Use those println()statements to generate HTML
- Maintain that HTML

33.4.2 The JSP Framework

- Use regular HTML for most of the pages
- Mark servlet code with special tags
- Entire JSP page gets translated into a servlet (once), and servlet is what actually gets invoked (for each request)
- The Java Server Pages technology combine with Java code and HTML tags in the same document to produce a JSP file.



33.4.3 Advantages of JSP over Competing Technologies

- **Versus ASP or ColdFusion**
  - JSPs offer better language for dynamic part i.e. java
  - JSPs are portable to multiple servers and operating systems
- **Versus PHP**
  - JSPs offer better language for dynamic part
  - JSPs offer better tool support
- **Versus pure servlets**
  - JSPs provide more convenient way to create HTML
  - JSPs can use standard front end tools (e.g., UltraDev)
  - JSPs divide and conquer the problem of presentation and business logic.

33.4.4 Setting Up Your Environment

In order to create a web-application that entirely consists of JSP pages and Html based pages, the setup is fairly simple as compared to a servlet based web application.

- Set your CLASSPATH. No.
- Compile your code. No.
- Use packages to avoid name conflicts. No.
- Put JSP page in special directory, like WEB-INF for servlets No.
  - tomcat_install_dir/webapps/ROOT
  - jrun_install_dir/servers/default/default-app
- Use special URL to invoke JSP page. No
- **However**
  - If you want to use java based classes in an application along with JSPs, Previous

rules about CLASSPATH, install dirs, etc, still apply to regular classes used by JSP

33.5 References:

- Java A Lab Course by Umair Javed
- Java Package Tutorial by Patrick Bouklee http://jarticles.com/package/package_eng.html
- JavaServer Pages Overview http://java.sun.com/products/jsp/overview.html

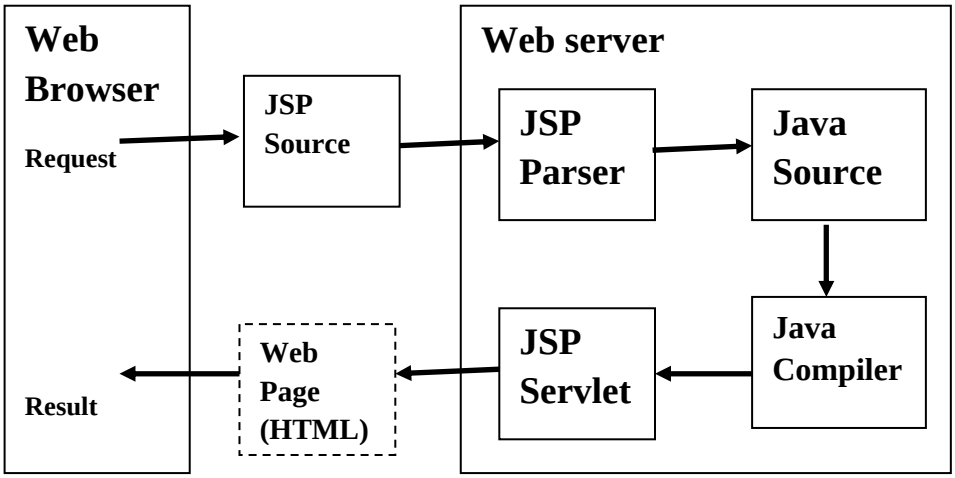
Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 34: Java Server Pages

As we concluded in our discussion on JSP, JSP is a text based document capable of returning either static or dynamic content to a client's browser. Static content and dynamic content can be intermixed. The examples of static content are HTML, XML & Text etc. Java code, displaying properties of JavaBeans and invoking business logic defined in custom tags are all examples of dynamic content.

34.1 First run of a JSP

Figure below shows what phases a JSP passed through before displaying result.



The web browser makes a request to JSP source code. This code is bifurcated into HTML and java code by the JSP parser. The java source code is compiled by the Java compiler resulting in producing a servlet equivalent code of a JSP. The servlet code is intermixed with HTML and displayed to the user. It is important to note that a JSP only passes through all these phases when it is invoked for the first time or when the changes have been made to JSP. Any later call to JSP does not undergo of compilation phase.

34.1.1 Benefits of JSP

- Convenient
  - we already know java and HTML. So nothing new to be learned to work with JSP.
  - Like servlets (as seen, ultimately a JSP gets converted into a servlet), provides an extensive infrastructure for

- Tracking sessions
- Reading and sending HTML headers
- Parsing and decoding HTML form data

- Efficient
  - Every request for a JSP is handled by a simple JSP java thread as JSP gets converted into a servlet. Hence, the time to execute a JSP document is not dominated by starting a process.
- Portable
  - Like Servlets, JSP is also a specification and follows a well standardized API.The JVM which is used to execute a JSP file is supported on many architectures and operating systems.
- Inexpensive
  - There are number of free or inexpensive Web Servers that are good for commercial quality websites.

34.1.2 JSP vs. Servlet

Let's compare JSP and Servlet technology by taking an example that simply plays current date.

First have a look on JSP that is displaying a current date. This page more looks like a HTML page except of two strangely written lines of codes. Also there are no signs of doGet(), doPost().

```
<%@ page import="java.util.*" %>
<html>
<body>
<h3>
Current Date is:<%= new Date()%>
</h3>
</body>
</html>
```

Now, compare the JSP code above with the Servlet code given below that is also displaying the current date.

//File: SearchPersonServlet.java

```
import java.io.*;
import java.net.*;
import javax.servlet.*;
```

```
import javax.servlet.http.*;
import java.util.*;
public class SearchPersonServlet extends HttpServlet {

protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println( “<html>” + “<body>” + “<h3>” +
“Current Date is:“ + new Date() +
“</h3>” + “</body>” + “</html>”
);
out.close();
}
// Handles the HTTP GET method.
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
processRequest(request, response);
}
// Handles the HTTP POST method.
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
processRequest(request, response);
}
}
```

Clearly, a lot of code is needed to be written in the case of servlet example to perform a basic job.

34.2 JSP Ingredients

Besides HTML, a JSP may contain the following elements.

- **Directive Elements**
  - Provides global control of JSP ……..……..**<%@%>**
- **Scripting Elements**
  - JSP comments ……...**<%----%>**
  - declarations……...**<%! %>**
  - Used to declare instance variables & methods
    - expressions……...**<%=%>**
  - A java code fragment which returns String
    - scriptlets……...**<%%>**
  - Blocks of java code

- **Action Elements**

- Special JSP tags ……..……..**<jsp: …../>**

We'll discuss in detail all the ingredients of JSP. This handout will cover only scripting elements, remaining ones will be discussed in next handouts.

34.3 Scripting Elements

34.3.1 Comments

Comments are ignored by JSP-to-servlet translator. Two types of comments are possibly used in JSP.

- **HTML comment:**

These comments are shown in browser, means on taking view source of the web page; these sorts of comments can be read. Format of HTML comments is like to:

<!-- comment text-->

- **JSP comment:**

These comments are not displayed in browser and have format like:

<%-- comment text --%>

34.3.2 Expressions

The format of writing a Java expression is:

<%= Java expression %>

These expressions are evaluated, after converted to strings placed into HTML page at the place it occurred in JSP page

Examples of writing Expressions are:

- <h2> Time: <% new java.util.Date() %> </h2>

will print current data & time after converting it to String

- <h2> Welcome: <% request.getParameter("name")%> </h2>

will print the name attribute

34.3.3 Scriptlets

The format of writing a scriptlet is: <% Java code %>
After opening up the scriptlet tag, any kind of java code can be written inside it. This code is inserted verbatim into corresponding servlet.

Example of writing a scriptlet is:

- <%String n = request.getParameter("name");
  out.println("welcome " + n);
  %>

The above scriptlet reads the name attribute and prints it after appending "welcome"

34.3.4Declarations

The format of writing a declaration tag is: <%! Java code %>
This tag is used to declare variables and methods at class level. The code written inside this tag is inserted verbatim into servlet's class definition.

Example of declaring a class level (attribute) variable is:

- <%!
  private int someField = 5; %>
  %>

Example of declaring a class level method is:

- <%!
  public void someMethod (……) {
  …….
  }
  %>

Code Example: Using scripting elements

The next example code consists of two JSP pages namely first.jsp and second.jsp. The user will enter two numbers on the first.jsp and after pressing the *calculate sum* button, able to see the sum of entered numbers on second.jsp

first.jsp

This page only displays the two text fields to enter numbers along with a button.

```
<html>
<body>
<h2>Enter two numbers to see their sum</h1>
```

```
<form name = "myForm" action="second.jsp" >
<h3> First Number </h3>
<input type="text" name="num1" />
<h3> Second Number </h3>
<input type="text" name="num2" /> <br/><br/>
<input type="submit" value="Calculate Sum" /> </form>
</body>
</html>
```

second.jsp

This page retrieves the values posted by first.jsp. After converting the numbers into integers, displays their sum.

```
<html>
<body>
<!-- JSP to sum two numbers -->
<%-- Declaration--%>
<%!
// declaring a variable to store sum int res;
// method helps in calculating the sum public int sum(int op1, int op2) {
return op1 + op2;
}
%>
<%-- Scripltet--%>
<%
String op1 = request.getParameter("num1"); String op2 = request.getParameter("num2"); int
firstNum = Integer.parseInt(op1);
int secondNum = Integer.parseInt(op2);
// calling method sum(), declared above in declartion tag res = sum(firstNum, secondNum);
%>
<%-- expression used to display sum --%>
<h3>Sum is: <%=res%> </h3>
</body>
</html>
```

34.4 Writing JSP scripting Elements in XML

Now days, the preferred way for composing a JSP pages is using XML. Although writing JSP pages in old style is still heavily used as we had shown you in the last example. Equivalent XML tags for writing scripting elements are given below:

- **Comments:** No equivalent tag is defined

- **Declaration:**<jsp:declartion> </jsp:declaration>
- **Expression:**<jsp:expression> </jsp:expression>
- **Scriptlet:**<jsp:scriptlet> </jsp:scriptlet>

It's important to note that every opening tag also have a closing tag too. The second.jspof last example is given below in XML style.

```
<?xml version="1.0" encoding="UTF-8"?>
<jsp:root xmlns:jsp="http://java.sun.com/JSP/Page" version="2.0">
<!-- to change the content type or response encoding change the following line
-->
<jsp:directive.page contentType="text/xml;charset=UTF-8"/>
<!-- any content can be specified here, e.g.: -->
<jsp:element name="text">
<jsp:body>
<jsp:declaration>
int res;
public int sum(int op1, int op2) {
return op1 + op2;
}
</jsp:declaration>
<jsp:scriptlet>
String op1 = request.getParameter("num1"); String op2 = request.getParameter("num2"); int
firstNum = Integer.parseInt(op1);
int secondNum = Integer.parseInt(op2);
res = sum(firstNum, secondNum);
</jsp:scriptlet>
<jsp:text> Sum is: </jsp:text>
<jsp:expression> res </jsp:expression>
</jsp:body>
</jsp:element> </jsp:root>
```

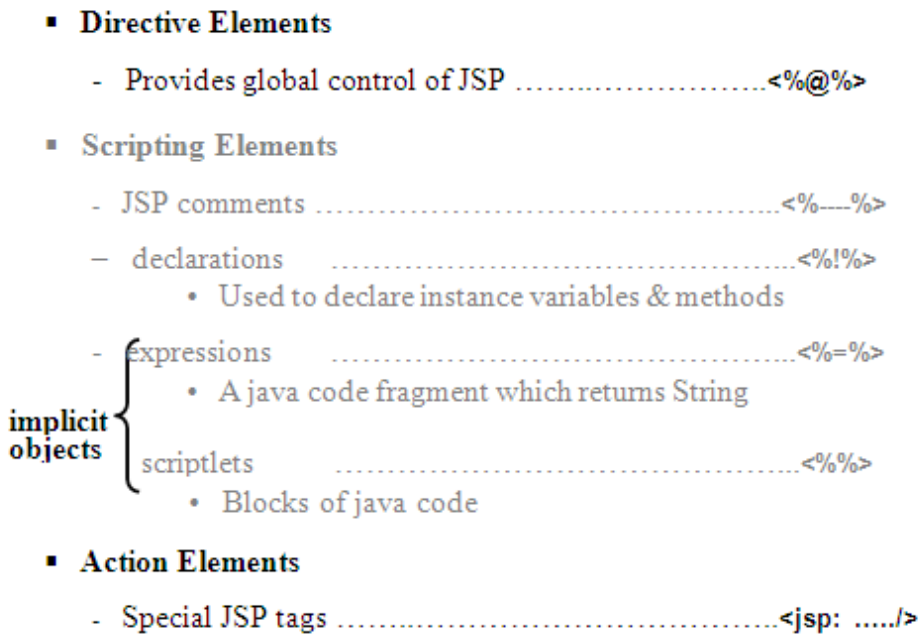
34.5 References:

- Java A Lab Course by Umair Javed
- Core Servlets and JSP by Marty Hall

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 35: JavaServer Pages

We have started JSP journey in the last handout and thoroughly discussed the JSP scripting elements. JSP directive elements and implicit objects will be discussed in this handout. Let's review JSP journey again to find out what part we have already covered.



We start our discussion from implicit objects. Let's find out what these are?

35.1 Implicit Objects

To simplify code in JSP expressions and scriptlets, you are supplied with eight automatically defined variables, sometimes called *implicit objects*. The three most important variables are request, response & out. Details of these are given below:

- **request**

  This variable is of type HttpServletRequest, associated with the request. It gives you access to the request parameters, the request type (e.g. GET or POST), and the incoming HTTP request headers (e.g. cookies etc).

-

This variable is of type HttpServletResponse, associated with the response to client. By using it, you can set HTTP status codes, content type and response headers etc.

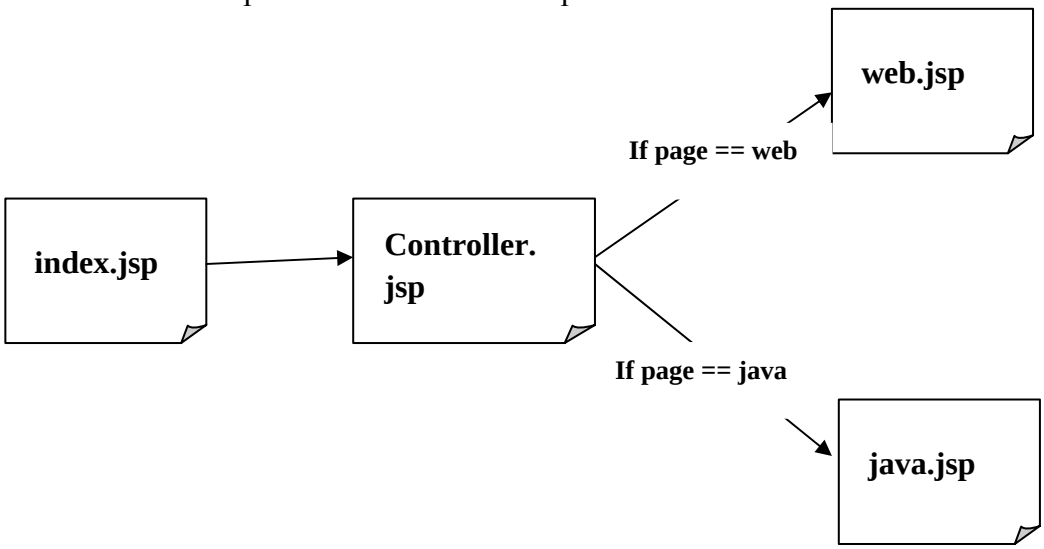
- **out**

This is the object of JspWriter used to send output to the client.

Code Example: Use of Implicit Objects

The following example constitutes of 4 JSP pages. These are index.jsp, controller.jsp, web.jsp and java.jsp. The user will select either the option of *"java"* or *"web"* from index.jsp, displayed in the form of radio buttons and submits the request to controller.jsp. Based on the selection made by the user, controller.jspwill redirect the user to respective pages (web.jsp or java.jsp).

The flow of the example is shown below in the pictorial form.



The code of these entire pages is given below.

index.jsp

```
<html>
<body>
<h2>Select the page you want to visit</h2>
<form name="myForm" action="controller.jsp" >
<h3>
<input type="radio" name = "page" value="web"/>
Web Design & Develoment
</h3>
<br>
```

```
<h3>
<input type="radio" name = "page" value="java"/>
Java
</h3>
<br>
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

controller.jsp

```
<html>
<body>
<!-- scriptlet -->
<%
// reading parameter “page”, name of radio button using
// implicit object request
String pageName = request.getParameter("page");
// deciding which page to move on based on “page” value
// redirecting user by using response implicit object
if (pageName.equals("web")) {
response.sendRedirect("web.jsp");
} else if (pageName.equals("java") )    {
response.sendRedirect("java.jsp");
}
%>
</body>
</html>
```

web.jsp

```
<html>
<body>
// use of out implicit object, to generate HTML
<%
out.println( "<h2>" +
"Welcome to Web Design & Development Page" + "</h2>"
);
%>
</body>
</html>
java.jsp
<html>
```

```
<body>
// use of out implicit object, to generate HTML
<%
out.println( "<h2>" +
"Welcome to Java Page" + "</h2>"
);
%>
</body>
</html>
```

The details of remaining 5 implicit objects are given below:

- **session**

  This variable is of type HttpSession, used to work with session object.

- **application**

  This variable is of type ServletContext. Allows to store values in key-value pair form that are shared by all servlets in same web application/

- **config**

  This variable is of type ServletConfig. Represents the JSP configuration options e.g. init-parameters etc.

- **pageContext**

  This variable is of type javax.servlet.jsp.PageContext, to give a single point of access to many of the page attributes. This object is used to stores the object values associated with this object.

- **exception**

  This variable is of type java.lang.Throwable. Represents the exception that is passed to JSP error page.

- **page**

  This variable is of type java.lang.Object. It is synonym for this.

35.2 JSP Directives

JSP directives are used to convey special processing information about the page to JSP container. It affects the overall structure of the servlet that results from the JSP page. It enables programmer to:

- Specify page settings
- To include content from other resources
- To specify custom-tag libraries

35.2.1Format

<%@ directive {attribute="val"}* %>

In JSP, there are three types of directives: page, include & taglib. The formats of using these are:

- **page:**<%@ **page**{attribute="val"}*%>
- **include:**<%@ **include**{attribute="val"}*%>
- **taglib:**<%@ **taglib**{attribute="val"}*%>

35.2.2JSP page Directive

Give high level information about servlet that will result from JSP page. It can be used anywhere in the document. It can control

- Which classes are imported
- What class the servlet extends
- What MIME type is generated
- How multithreading is handled
- If the participates in session
- Which page handles unexpected errors etc.

The lists of attributes that can be used with page directive are:

- **language** = "java"
- **extends** = "package.class"
- **import** = "package.*,package.class"
- **session** = "**true** | false"
- **info**= "text"
- **contentType** = "mimeType"
- **isThreadSafe** = "**true** | false"
- **errorPage**= "relativeURL"
- **isErrorPage** = "true | **false**"

Some example uses are:

- To import package like java.util
  <%@page import="java.util.*" info="using util package" %>
- To declare this page as an error page
  <%@ page isErrorPage = "true" %>
- To generate the excel spread sheet
  <%@ page contentType = "application/vnd.ms-excel" %>

35.2.3JSP include Directive

Lets you include (reuse) navigation bars, tables and other elements in JSP page. You can include files at

- Translation Time (by using include directive)
- Request Time (by using Action elements, discussed in next handouts)

Format

<%@include file="relativeURL"%>

Purpose

To include a file in a JSP document at the time document is translated into a servlet. It may contain JSP code that affects the main page such as response page header settings etc.

Example Code: using include directive

This example contains three JSP pages. These are index.jsp, header.jsp& footer.jsp. The header.jspwill display the text of *"web design and development"* along with current date. The footer.jspwill display only *"virtual university"*. The outputs of both these pages will be included in index.jspby using JSP include directive.

header.jsp

```
<%@page       import="java.util.*"%>
<html>
<body>
<marquee>
<h3> Web Desing & Development </h3>
<h3><%=new Date()%></h3>
```

```
</marquee>
</body>
</html>
```

footer.jsp

```
<html>
<body>
<marquee>
<h3> Virtual University </h3>
</marquee>
</body>
</html>
```

index.jsp

```
<html>
<body>
// includes the output of header.jsp
<%@include file="header.jsp" %>
<TABLE BORDER=1>
<TR><TH></TH><TH>Apples<TH>Oranges
<TR><TH>First Quarter<TD>2307<TD>4706
<TR><TH>Second Quarter<TD>2982<TD>5104
<TR><TH>Third Quarter<TD>3011<TD>5220
<TR><TH>Fourth Quarter<TD>3055<TD>5287 </TABLE>
// includes the output of footer.jsp
<%@include file="footer.jsp" %>
</body>
</html>
```

Example Code: setting content type to generate excel spread sheet

In this example, index.jsp is modified to generate excel spread sheet of the last example. The change is shown in bold face.

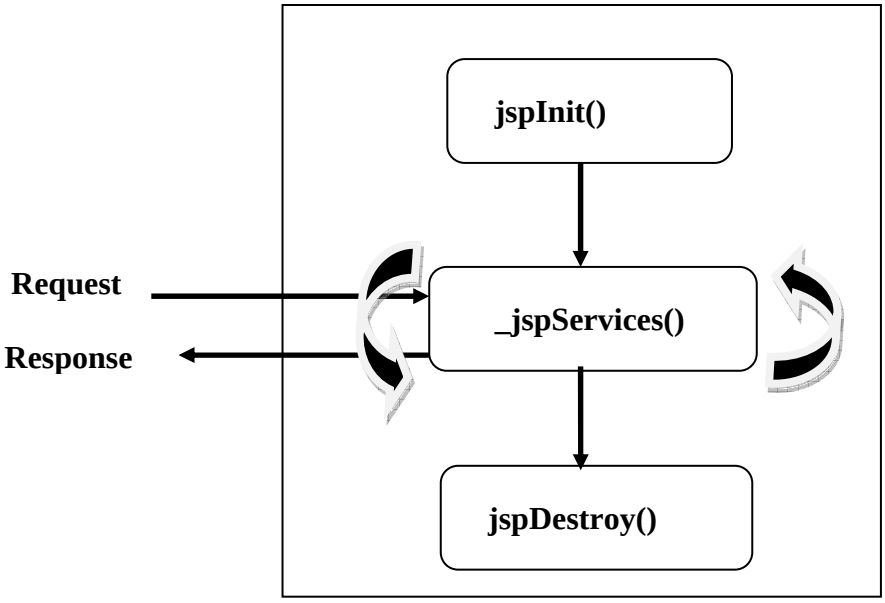
index.jsp

```
// setting content type to generate excel sheet using page directive
<%@page contentType="application/vnd.ms-excel"  %>
<html>
<body>
// includes the output of header.jsp
<%@include file="header.jsp" %>
<TABLE BORDER=1>
<TR><TH></TH><TH>Apples<TH>Oranges
<TR><TH>First Quarter<TD>2307<TD>4706
```

```
<TR><TH>Third Quarter<TD>3011<TD>5220
<TR><TH>Fourth Quarter<TD>3055<TD>5287 </TABLE>
// includes the output of footer.jsp
<%@include file="footer.jsp" %>
</body>
</html>
```

35.3 JSP Life Cycle Methods

The life cycle methods of JSP are jspInit(), _jspService() and jspDesroy(). On receiving each request, _jspService() method is invoked that generates the response as well.



35.4 References:

- Java A Lab Course by Umair Javed
- Core Servlets and JSP by Marty Hall

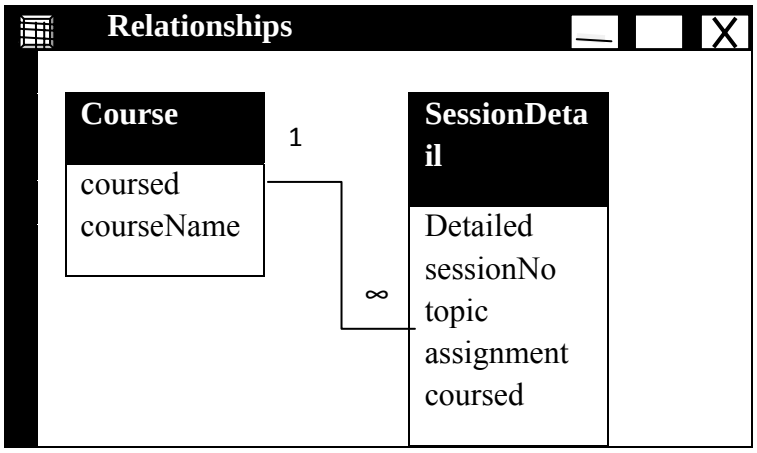
Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 36

In the last handout, we learned how to work with JSP directives and the use of implicit objects. In this handout, we'll learn about JavaBeans and what affect they produce. Before learning JavaBeans, let's start with an example that helps us to understand the impact of using JavaBeans.

Code Example: Displaying Course Outline

This example is actually the modification of the last one we had discussed in previous handout. User will select either course *"web design and development"* or *"java"*. On submitting request, course outline would be displayed of the selected course in tabular format. This course outline actually loaded from database. The schema of the database used for this example is given below:



The flow of this example is shown below:

index.jsp

This page is used to display the course options to the user in the radio button form.

```
<html>
<body>
<h2>Select the page you want to visit</h2>
<form name="myForm" action="controller.jsp" >
<h3>
<input type="radio" name = "page" value="web"/> Web Design & Develoment
</h3> <br>
<h3>
<input type="radio" name = "page" value="java"/> Java
</h3><br>
<input type="submit" value="Submit" />
```

```
</form>
</body>
</html>
```

controller.jsp

Based upon the selection made by the user, this page will redirect the user to respective pages. Those are web.jspand java.jsp

```
<html>
<body>
<!-- scriptlet -->
<%
// reading parameter named page
String pageName = request.getParameter("page");

// redirecting user based on selection made if (pageName.equals("web")) {
response.sendRedirect("web.jsp");
} else if (pageName.equals("java") )    {
response.sendRedirect("java.jsp");
}
%>
/body>
</html>
```

web.jsp

This page is used to display course outline of "*web design and development*" in a tabular format after reading them from database. The code is:

```
// importing java.sql package using page directive, to work with
// database
<%@page import="java.sql.*"%>
<html>
<body>
<center>
<h2> Welcome to Web Design & Development Page </h2>
<h3> Course Outline</h3>
<TABLE BORDER="1" >
<TR>

<TH>Session No.</TH>
<TH>Topics</TH>
```

```
</TR>
<%-- start of scriptlet --%>
<%
// establishing conection
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

String conUrl = "jdbc:odbc:CourseDSN";
Connection con = DriverManager.getConnection(conUrl);

// preparing query using join statements
String sql = " SELECT sessionNo, topic, assignment " + " FROM Course, SessionDetail" +
" WHERE courseName = ? " +
" AND Course.courseId = SessionDetail.courseID";
PreparedStatement pStmt = con.prepareStatement(sql);

// setting parameter value ”web”. pStmt.setString( 1 , "web");
ResultSet rs = pStmt.executeQuery(); String sessionNo;
String topic;
String assignment;
// iterating over resultset while (rs.next()) {
sessionNo = rs.getString("sessionNo");
topic = rs.getString("topic");
assignment = rs.getString("assignment");

if (assignment == null){
assignment = "";
}
%>
<%-- end of scriptlet --%>

<%-- The values are displayed in tabular format using expressions, however it can also be done using
out.println(sessionNo) like statements
--%>

<TR>
<TD> <%=sessionNo%> </TD>
<TD> <%=topic%> </TD>
<TD> <%=assignment%> </TD>
</TR>

<%
} // end while
%>

</TABLE >
```

```
</center>
</body>
</html>
```

java.jsp

The code of this page is very much alike of *"web.jsp"*. The only change is in making of query. Here the value is set *"java"* instead of *"web"*

```
// importing java.sql package using page directive, to work with
// database
<%@page import="java.sql.*"%>
<html>
<body>
<center>
<h2> Welcome to Java Page </h2>
<h3> Course Outline</h3>
<TABLE BORDER="1" >
<TR>
<TH>Session No.</TH>
<TH>Topics</TH>
<TH>Assignments</TH>
</TR>
<%-- start of scriptlet --%>
<%
// establishing conection
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
String conUrl = "jdbc:odbc:CourseDSN";
Connection con = DriverManager.getConnection(conUrl);
// preparing query using join statements
String sql = " SELECT sessionNo, topic, assignment " + " FROM Course, SessionDetail" +
" WHERE courseName = ? " +
" AND Course.courseId = SessionDetail.courseID";
PreparedStatement pStmt = con.prepareStatement(sql);
// setting parameter value "web". pStmt.setString( 1 , "java");
ResultSet rs = pStmt.executeQuery(); String sessionNo;
String topic;
String assignment;
// iterating over resultset while (rs.next()) {
sessionNo = rs.getString("sessionNo");
topic = rs.getString("topic");
assignment = rs.getString("assignment");
if (assignment == null){
assignment = "";
}
```

```
%>
<%-- end of scriptlet --%>
<%-- The values are displayed in tabular format using expressions, however it can also be done using
out.println(sessionNo) like statements
--%>
<TR>
<TD> <%=sessionNo%> </TD>
<TD> <%=topic%> </TD>
<TD> <%=assignment%> </TD>
</TR>
<%
} // end while
%>
</TABLE >
</center>
</body>
</html>
```

Issues with Last Example

Too much cluttered code in web.jsp and java.jsp. This makes it very difficult to understand (probably you experienced it by yourself) and to make changes/enhancements.

A single page is doing everything that is really a bad approach while making of web applications. The tasks performed by web.jspor java.jspare:

- Displaying contents (Presentation logic)
- Connecting with database (DB connectivity logic)
- Results Processing (Business Logic)

Can we simplify it? Yes, the answer lies in the use of *JavaBeans* technology.

36.1 JavaBeans

- A java class that can be easily reused and composed together in an application.
- Any java class that follows *certain design conventions* can be a JavaBean.

36.1.1 JavaBeans Design Conventions

These conventions are:

- A bean class must have a zero argument constructor
- A bean class should not have any public instance variables/attributes (fields)
- Private values should be accessed through setters/getters
    - For boolean data types, use boolean isXXX( ) & setXXX(boolean)
- A bean class must be serializable

A Sample JavaBean

The code snippet of very basic JavaBean is given below that satisfies all the conventions described above. The MyBean.java class has only one instance variable.

```
public class MyBean implements Serializable {

private String name;
// zero argument constructor
public MyBean( ){
name = “”;
}
// standard setter
public void setName(String n) {
name = n;
}
// standard getter
public String getName( ) {
return name;
}
// any other method
public void print( ) {
System.out.println(“Name is: ” + name);
}
} // end Bean class
```

Example Code: Displaying course outline by incorporating

JavaBeans

This example is made by making more enhancements to the last one. Two JavaBeans are

included in this example code. These are CourseOutlineBean& CourseDAO.

The CourseOutlineBean is used to represent one row of the table. It contains the following attributes:

- sessionNo
- topic
- assignment

The CourseDAO (where DAO stands of Data Acess Object) bean encapsulates database connectivity and result processing logic.

The web.jsp and java.jsp will use both these JavaBeans. The code of these and the JSPs used in this example are given below.

CourseOutlineBean.java

```
package vu;
import java.io.*;
public class CourseOutlineBean implements Serializable{
private int sessionNo; private String topic; private String assignment;
// no argument constructor public CourseOutlineBean() { sessionNo = 0;
topic    = "";
assignment = "";
}
// setters
public void setSessionNo(int s){
sessionNo = s;
}
public void setTopic(String t){
topic = t;
}
public void setAssignment(String a){
assignment = a;
}
// getters
public int getSessionNo( ){
return sessionNo;
}
public String getTopic( ){
return topic;
}
public String getAssignment( ){
return assignment;
}
} // end class
```

CourseDAO.java

```
package vu;
import java.io.*;
import java.sql.*;
import java.util.*;
public class CourseDAO implements Serializable{
private Connection con;
public CourseDAO() {
establishConnection();
}
//********** establishConnection method ********************
// method used to make connection with database private void establishConnection(){
try{
// establishing conection
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
String conUrl = "jdbc:odbc:CourseDSN";
con = DriverManager.getConnection(conUrl);
}catch(Exception ex){ System.out.println(ex);
}
//*********** retrieveCourseList method *********************
public ArrayList retrieveCourseList(String cName){ ArrayList courseList = new ArrayList();
try{
}
String sql = " SELECT sessionNo, topic, assignment " +
" FROM Course, SessionDetail" + " WHERE courseName = ? " +
" AND Course.courseId = SessionDetail.courseID ";
PreparedStatement pStmt = con.prepareStatement(sql);
pStmt.setString(1, cName);

ResultSet rs = pStmt.executeQuery();
int sNo; String topic; String assignment;
while ( rs.next() ) {
sNo      = rs.getInt("sessionNo");
topic     = rs.getString("topic");
assignment = rs.getString("assignment");
if (assignment == null){
assignment = "";
}
// creating a CourseOutlineBean object
CourseOutlineBean cBean = new CourseOutlineBean();
cBean.setSessionNo(sNo);
cBean.setTopic(topic);
cBean.setAssignment(assignment);
// adding a bean to arraylist
```

```
courseList.add(cBean);
}
}catch(Exception ex){ System.out.println(ex);
} finally {
// to close connection releaseResources();
}
// returning ArrayList object
return courseList;

} // end retrieveCourseOutline
//********** releaseResources method ********************
private void releaseResources(){
try{
if(con != null){
con.close();
}
}catch(Exception ex){ System.out.println();
}
}
} // end releaseResources

}// end CourseDAO
```

index.jsp

This page is used to display the course options to the user in the radio button form.

```
<html>
<body>
<h2>Select the page you want to visit</h2>
<form name="myForm" action="controller.jsp" >
<h3>
<input type="radio" name = "page" value="web"/>
Web Design & Develoment
</h3> <br>
<h3>
<input type="radio" name = "page" value="java"/> Java
</h3><br>
<input type="submit" value="Submit" />
```

```
</form>
</body>
</html>
```

controller.jsp

Based on user selection, redirects the user to desired page.

```
<html>
<body>

<!-- scriptlet -->
<%
String pageName = request.getParameter("page");
if (pageName.equals("web")) {
response.sendRedirect("web.jsp");
} else if (pageName.equals("java") )    {
response.sendRedirect("java.jsp");
}
%>
</body>
</html>
```

web.jsp

This page is used to display course outline of "*web design and development*" in a tabular format after reading them from database. Moreover, this page also uses the JavaBeans (CourseOutlineBean & CourseDAO).

```
<%@page import="java.util.*" %>
<%-- importing vu package that contains the JavaBeans--%>
<%@page import="vu.*" %>
<html>
<body>
<center>
<h2> Welcome to Web Design & Development Course </h2>
<h3> Course Outline</h3>
<TABLE BORDER="1" >
<TR>
<TH>Session No.</TH>
<TH>Topics</TH>
<TH>Assignments</TH>
</TR>
<%-- start of scriptlet --%>
```

```
// creating CourseDAO object
CourseDAO courseDAO = new CourseDAO();
// calling retrieveCourseList() of CourseDAO class and
// passing “web” as value. This method returns ArrayList
ArrayList courseList = courseDAO.retrieveCourseList("web"); CourseOutlineBean webBean = null;
// iterating over ArrayList to display course outline
for(int i=0; i<courseList.size(); i++){
webBean = (CourseOutlineBean)courseList.get(i);
%>
<%-- end of scriptlet --%>
<TR>
<TD> <%= webBean.getSessionNo()%> </TD>
<TD> <%= webBean.getTopic()%> </TD>
<TD> <%= webBean.getAssignment()%> </TD>
</TR>
<%
} // end for
%>
</TABLE >
</center>
</body>
</html>
```

java.jsp

The code contains by this page is almost same of web.jsp. Here, “java” is passed to retieveCourseList( ) method. This is shown in boldface.

```
<%@page import="java.util.*" %>
<%-- importing vu package that contains the JavaBeans--%>
<%@page import="vu.*" %>
<html>
<body>
<center>
<h2> Welcome to Java Course </h2>
<h3> Course Outline</h3>
<TABLE BORDER="1" >
<TR>
<TH>Session No.</TH>
<TH>Topics</TH>
<TH>Assignments</TH>
</TR>
<%-- start of scriptlet --%>
<%
// creating CourseDAO object
CourseDAO courseDAO = new CourseDAO();
```

```
// calling retrieveCourseList() of CourseDAO class and
// passing “java” as value. This method returns ArrayList
ArrayList courseList = courseDAO.retrieveCourseList("java");
CourseOutlineBean javaBean = null;
// iterating over ArrayList to display course outline
for(int i=0; i<courseList.size(); i++){
javaBean = (CourseOutlineBean)courseList.get(i);
%>
<%-- end of scriptlet --%>
<TR>
<TD> <%= javaBean.getSessionNo()%> </TD>
<TD> <%= javaBean.getTopic()%> </TD>
<TD> <%= javaBean.getAssignment()%> </TD>
</TR>
<%
} // end for
%>
</TABLE >
</center>
</body>
</html>
```

36.2 References:

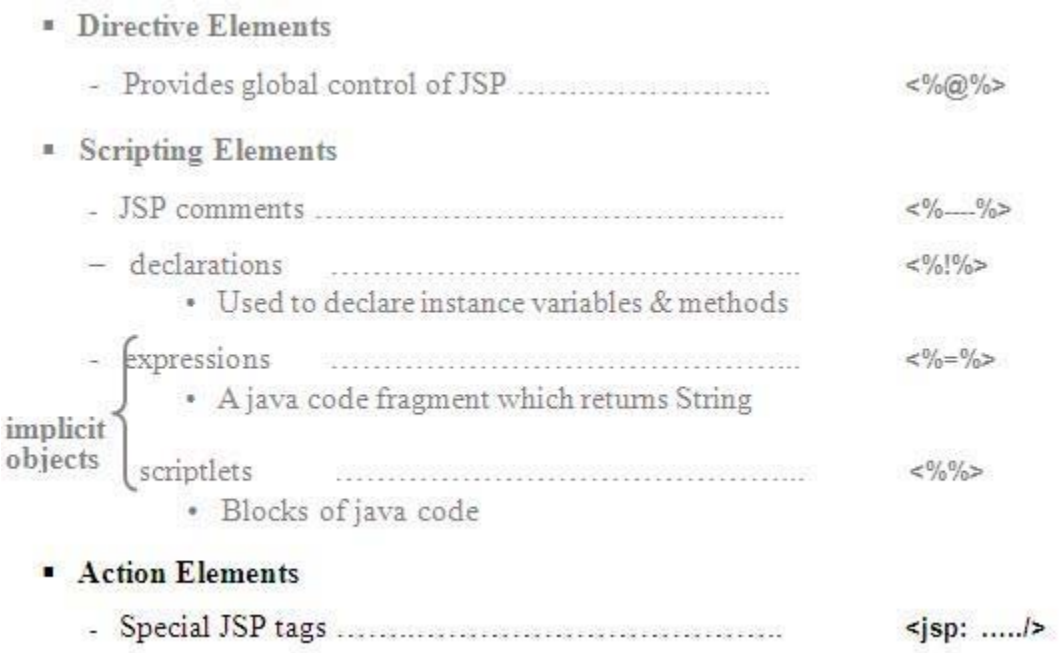
- Entire material for this handout is taken from the book **JAVA A Lab Course** by **Umair Javed**. This material is available just for the use of VU students of the course Web Design and Development and not for any other commercial purpose without the consent of author.

Note: Coding exercises in working condition for this lecture are also available on **“Downloads”** section of LMS.

Lecture 37: JSP Action Elements and Scope

The journey we had started of JSP is very much covered except of JSP action elements. In this handout, we'll study the use of JSP action elements. Further also learn how and where to store JavaBean objects that can be shared among JSP pages.

Let's first quickly look on the JSP journey to find out where we have reached.



37.1 JSP Action Elements

JSP action elements allow us to work with JavaBeans, to include pages at request time and to forward requests to other resources etc.

Format

Expressed using XML syntax

- **Opening tag<jsp:actionElement attribute="value" ….. >**
- **Body** body
- **Closing tag</jsp:actionElement>**

Empty tags (without body) can also be used like

<jsp:actionElement attribute="value" ….. >

Some JSP Action Elements

- To work with JavaBeans
  -<jsp:useBean />
  -<jsp:setProperty />
  -<jsp:getProperty />
- To include resources at request time
  -<jsp:include />
- To forward request to another JSP or Servlet
  -<jsp:forward />
- To work with applets
  -<jsp:plugin />

37.2 Working with JavaBeans using JSP Action Elements

The three action elements are used to work with JavaBeans. These are discussed in detail below:

37.2.1 JSP useBean Action Element

It is used to obtain a reference to an existing JavaBean object by specifying id(name of object) and scope in which bean is stored. If a reference is not found, the bean is instantiated.

The format of this action element is:

<jsp:useBean id = "name"
scope = "**page**|request|session|application" class="package.Class "
/>

The id attribute specifies the name of the JavaBean object that is also used for later references. The scopeattribute can have one possible value out of *page*, *request*, *session* and *application*. If this attribute is omitted, the default value of scope attribute is page. We'll discuss in detail about scope shortly.

The classattribute specifies the type of object is going to be created.

jsp:useBeanis being equivalent to building an object in scriptlet. For example to build an object of MyBeanusing scriptlet is:

```
<%
        MyBean m = new MyBean( );
%>
```

Achieving above functionality using jsp:useBean action element will look like this:

```
<jsp:useBean    id = “m”
scope = “page”
class=“vu.MyBean”                    />
```

In the above code snippet, we are assuming that MyBeanlies in vu package.

37.2.2 JSP setProperty Action Element

To set or change the property value of the specified bean. String values are converted to types of properties by using the related conversion methods.

The format of this action element is:

```
<jsp:setProperty name = “beanName or id” property = “name” value
=“value”/>
```

The name attribute should match the id given in jsp:useBean. The property attribute specifies the name of the property to change and the valueattribute specifies the new value.

jsp:setPropertyis being equivalent to following code of scriptlet. For example to change the *name* property of m (instance of MyBean) using scriptlet is:

```
<%
        m.setProperty(“ali”);
%>
```

Achieving above functionality using jsp:setPropertyaction element will look like this:

```
<jsp:setProperty name = “m” property = “name” value = “ali” />
```

37.2.3 JSP getProperty Action Element

Use to retrieves the value of property, converts it to String and writes it to output stream.

The format of this action element is:

```
<jsp:getProperty name = “beanName or id”property = “name”/>
```

jsp:getProperty is being equivalent to following code of scriptlet. For example to retrieve the *name* property of m (instance of MyBean) followed by writing it to output stream, scriptlet code will look like:

```
<%
String name = m.getName( ); out.println(name);
%>
```

Achieving above functionality using jsp:getProperty action element will look like this:

```
<jsp:getProperty name = "m"property = "name" />
```

Example Code: Calculating sum of two numbers by using action elements and JavaBean

This example contains index.jspand result.jspand one JavaBean i.e. SumBean. User will enter two numbers on index.jspand their sum will be displayed on result.jsp. Let's examine these one after another

SumBean.java

The SumBean has following attributes

- firstNumber
- secondNumber
- sum

The *firstNumber* and *secondNumbers* are *"write-only"* properties means for these only setters would be defined. Whereas *sum* is a *"read-only"* property as only getter would be defined for it.

The SumBeanalso contain one additional method for calculating sum i.e. calulateSum(). After performing addition of *firstNumber* with *secondNumber*, this method will assign the result to *sum* attribute.

```
package vu;
import java.io.*;
public class SumBean implements Serializable{
private int firstNumber;
 private int secondNumber;
private int sum;
// no argument constructor
 public SumBean() {
```

```
firstNumber = 0;
secondNumber = 0;
sum = 0;
}
// firstNumber & secondNumber are writeonly properties
// setters
public void setFirstNumber(int n){
firstNumber = n;
}
public void setSecondNumber(int n){
secondNumber = n;
}
// no setter for sum
// sum is a read only property
public int getSum( ){
return sum;
}
// method to calculate sum
public void calculateSum() {
sum = firstNumber + secondNumber;
}
}
```

index.jsp

This page will display two text fields to enter number into them.

```
<html>
<body>

<h2>Enter two numbers to calculate their sum</h2>
<form name="myForm" action="result.jsp">
<h3>
Enter first number
<input type="text" name="num1" />
<br/>
Enter second number
<input type="text" name="num2" />
<br/>
<input type="submit" value="Calculate Sum" />
</h3>
</form>
</body>
</html>
```

result.jsp

This page will calculate the sum of two entered numbers by the user and displays the sum back to user. The addition is performed using SumBean

```
<%-- importing vu package that contains the SumBean --%>
<%@page import="vu.*"%>
<html>
<body>
<h2>The sum is:
<%-- instantiating bean using action element -- %>
<%--
//Servlet equivalent code of useBean
SumBean sBean = new SumBean();
--%>

<jsp:useBean id="sBean" class="vu.SumBean" scope="page"/>
<%-- setting firstNumber property of sBean using action elements
-- %>
<%-- implicit conversion from string to int as num1 is of type
String and firstNumber is of type int
--%>
<%--
//Servlet equivalent code of setProperty for num1
int no = Integer.parseInt(request.getParameter("num1"));
sBean.setFirstNumber(no);
--%>
<jsp:setProperty name="sBean"
property="firstNumber" param="num1" />
<%--
//Servlet equivalent code of setProperty for num2
int no = Integer.parseInt(request.getParameter("num2"));
sBean.setSecondNumber(no);
--%>
//Servlet equivalent code of setProperty for num2
int no = Integer.parseInt(request.getParameter("num2"));
sBean.setSecondNumber(no);
<jsp:setProperty name="sBean"
property="secondNumber" param="num2" />
<%
// calling calculateSum() method that will set the value of
// sum attribute
sBean.calculateSum();
%>
```

```
<%--
// servlet equivalent code of displaying sum
int res = sBean.getSum();
out.println(res);
--%>
<jsp:getProperty name="sBean" property="sum" />
</h2>
</body>
</html>
```

37.3 Sharing Beans & Object Scopes

So far, we have learned the following techniques to create objects.

- Implicitly through JSP directives
- Explicitly through actions
- Directly using scripting code

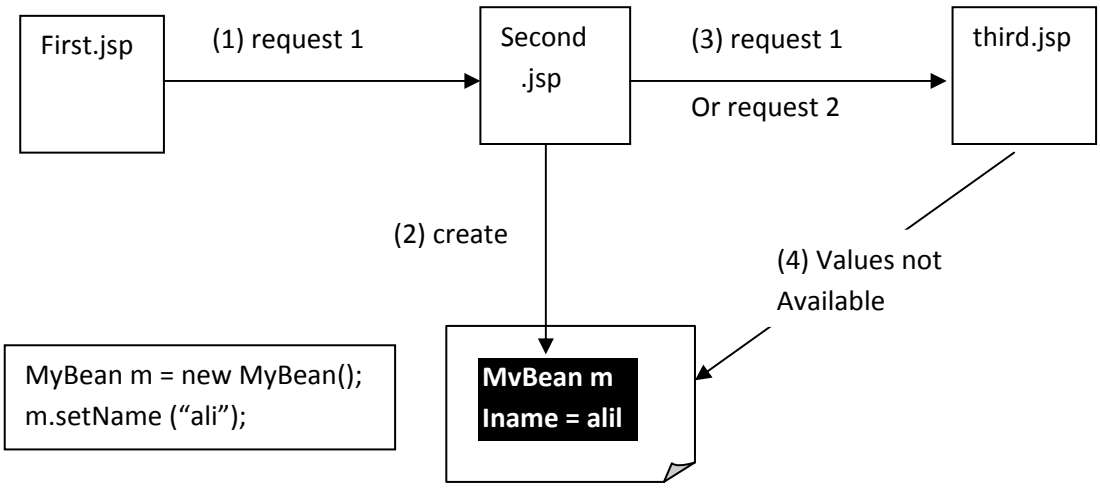
Although the beans are indeed bound to local variables, that is not the only behavior. They are also stored in four different locations, depending on the value of the optional scopeattribute of jsp:useBean. The scope attribute has the following possible values: page, request, sessionand application.

Let's discover what impact these scopes can produce on JavaBeans objects which are stored in one of these scopes.

37.3.1page

This is the default value of scope attribute, if omitted. It indicates, in addition to being bound to local variable, the bean object should be placed in the pageContext object. The bean's values are only available and persist on JSP in which bean is created.

In practice, beans created with page scope are always accessed (their values) by jsp:getProperty, jsp:setProperty, scriptlets or expressions later in the same page. This will be more cleared with the help of following diagram:

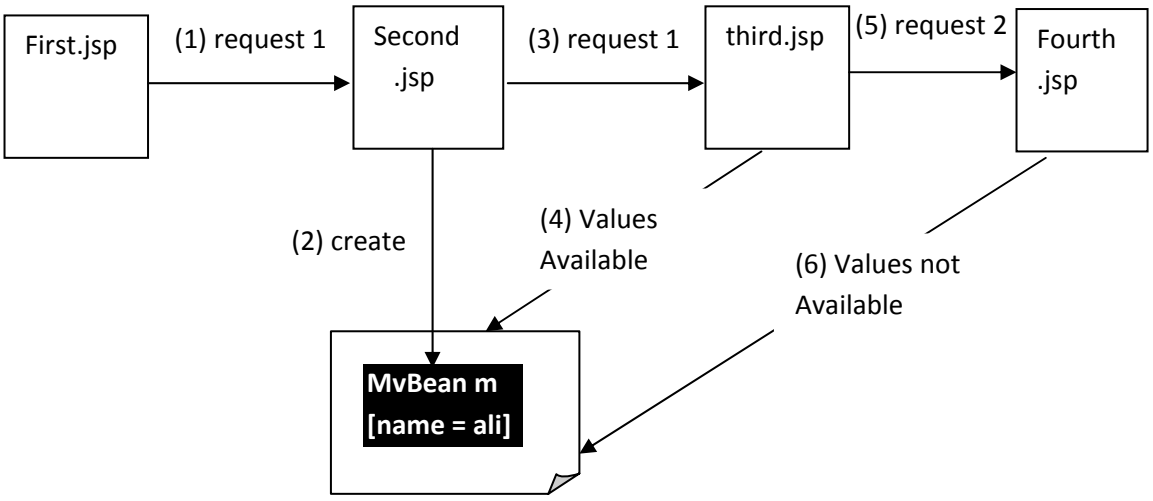


In the diagram above, *first.jsp* generates a request *"request 1"* that is submitted to *second.jsp*. Now, *second.jsp* creates an object m of MyBean by calling its default constructor and stores a value *"ali"* for the *name* property by making a call to appropriate setter method. Since, the scope specified in this example is "page" when the object of MyBean is instantiated using jsp:useBean action element. Therefore, object (m) of MyBeanis stored in PageContext.

Whether, second.jspforwards the same request *(request 1)* to third.jspor generates a new request *(request 2),* at third.jsp, values (e.g. ali) stored in MyBeanobject m, are not available. Hence, specifying scope "page" results in using the object on the same page where they are created.

37.3.2request

This value signifies that, in addition to being bound to local variable, the bean object should be placed in ServletRequest object for the duration of the current request. In other words, until you continue to forward the request to another JSP/servlet, the beans values are available. This has been illustrated in the following diagram.

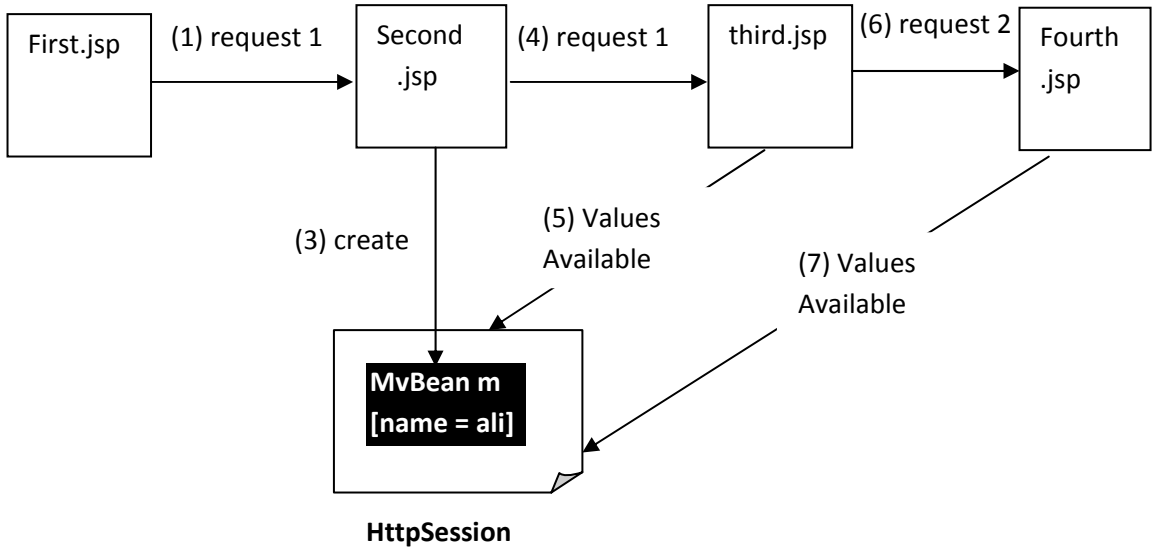


In the diagram above, MyBeanis instantiated by specifying scope =“request” that results in storing object in ServletRequest. A value *“ali”* is also stored in m using setter method.

second.jspforwards the same request *(request 1)* to third.jsp, since scope of m(object of MyBean) is request, as a result third.jsp can access the values(e.g. ali) stored in m. According to the figure, third.jspgenerates a new request *(request 2)* and submits it to fourth.jsp. Since a new request is generated therefore values stored in object m(e.g. ali) are not available to fourth.jsp.

37.3.3 session

This value means that, in addition to being bound to local variable, the bean object will be stored in the HttpSession object associated with the current request. As you already know, object’s value stored in HttpSession persists for whole user’s session. The figure below helps in understanding this concept.

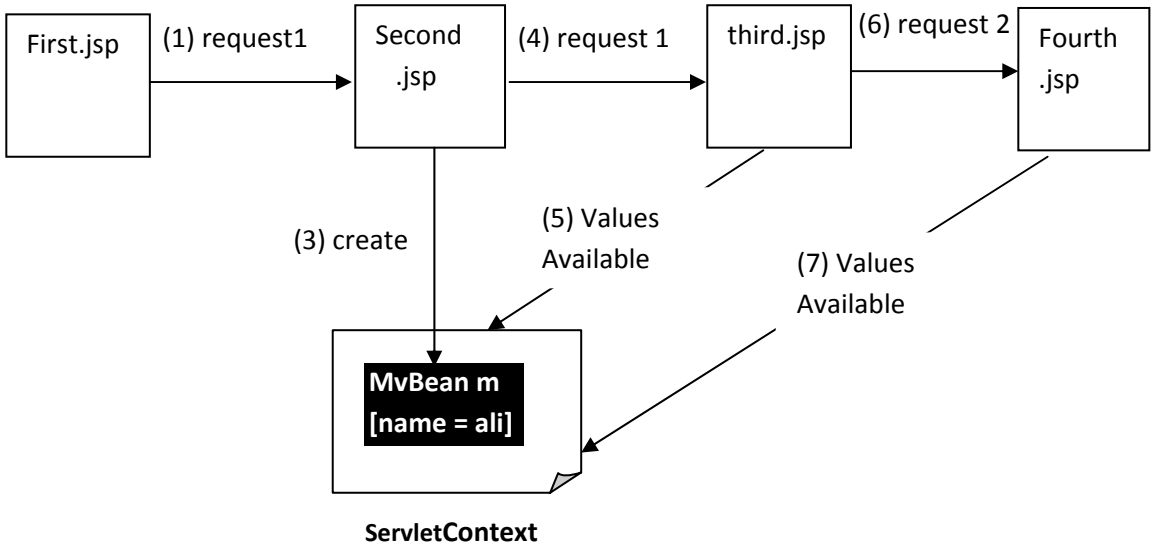


In the diagram above, MyBeanis instantiated by specifying scope = "session" that results in storing object in HttpSession. A value *"ali"* is also stored in m using setter method.

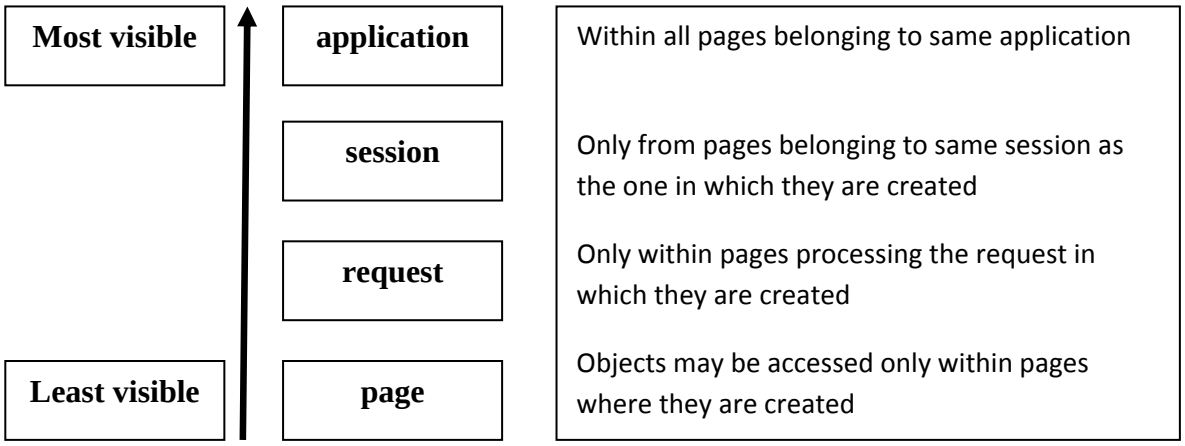
Irrespective of request forwarding or new request generation from second.jsp to other resources, the values stored in HttpSession remains available until user's session is ended.

37.3.4 Application

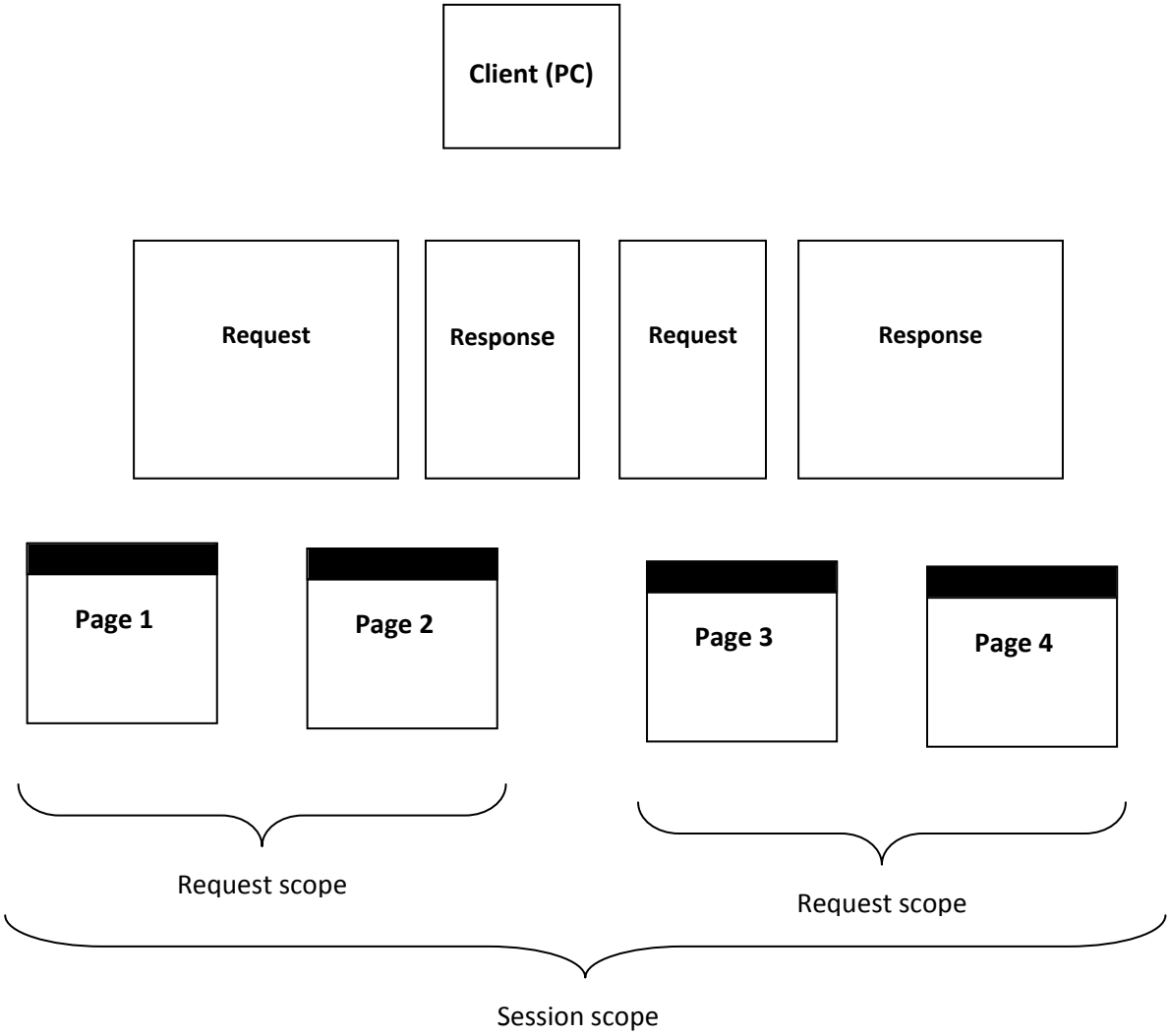
This very useful value means that, in addition to being bound to local variable, the bean object will be stored in ServletContext. The bean objects stored in ServletContextis shared by all JSPs/servlets in the same web application. The diagram given below illustrates this scenario:



37.4 Summary of Object's Scopes



Let's take another view of session, request & page scopes in the next figure that helps us to understand the under beneath things.

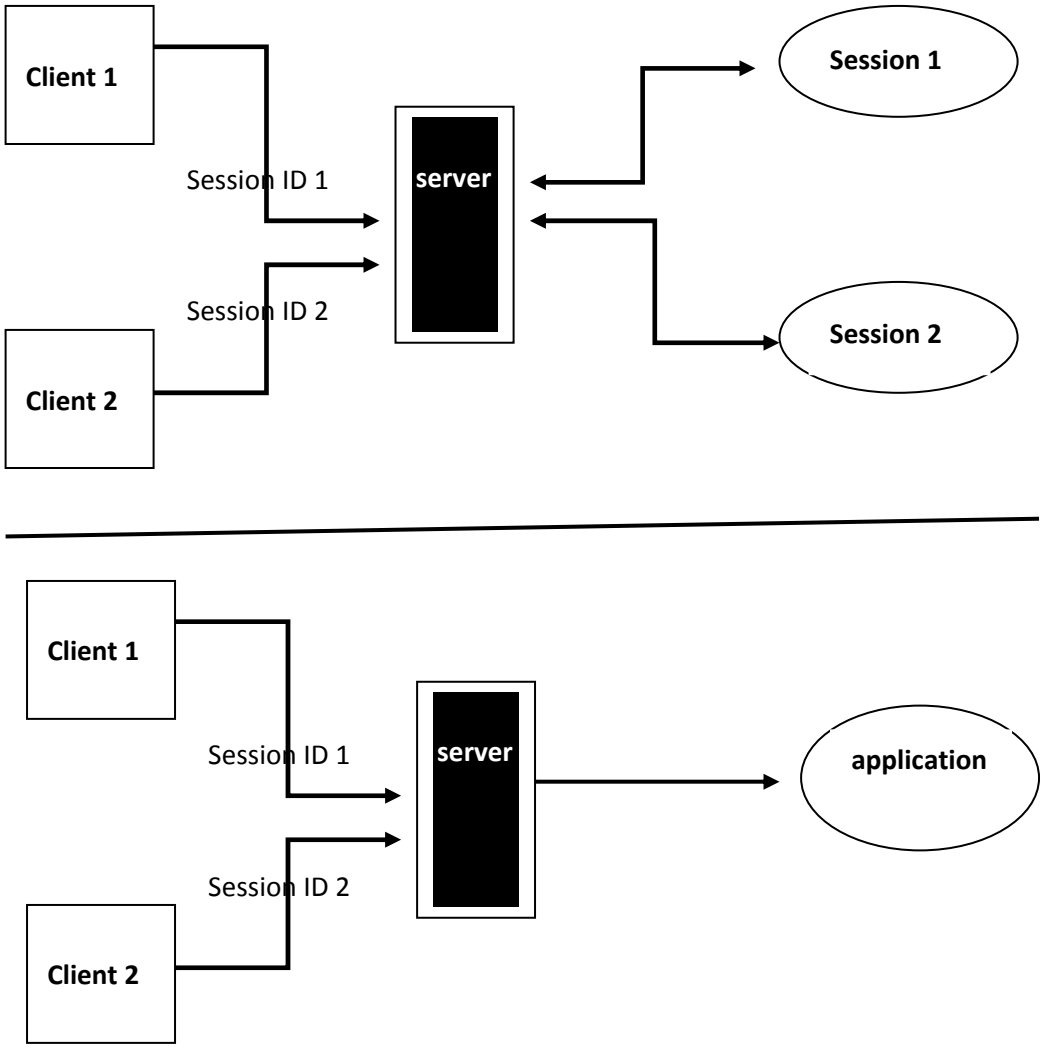


The figure shows four JavaServer Pages. Each page has its own page scope. Therefore objects stored in page scope are only available to same pages on which they are created.

Suppose page1 forwards the request to page2. Objects stored in request scope remains available to page1 as well to page 2. Similar case is true for page 3 & page 4.

If user makes a visit to all these pages in one session, object's values stored in session scope remains available on all these pages.

To understand the difference between sessions & application scope, consider the following figure:



As you can conclude from the figure, for each user (client), objects are stored in different sessions. However, in the case of application scope, all users stores objects in single place.

37.5 More JSP Action Elements

Let's talk about two important action elements. These are include& forward.

37.5.1 JSP include action Element

It is used to include files at request time. For example, to reuse HTML, JSP or plain text content. It's important to note that JSP content cannot affect main page (in which output is included); only output of included JSP is used. It also allows updating of the included content without changing the main JSP.

The jsp:includeaction element requires two attributes: page& flush.

- page: a relative URL of the file to be included.
- flush: must have the value *"true"*

<jsp:include page = "relative URL" flush = "true" />

jsp:include is being equivalent to following code of scriptlet. For example to include the output of *one.jsp* , scriptlet code will look like:

```
<%
  RequestDispatcher rd =request.getRequestDispatcher("one.jsp");
  rd.include(request, response);
%>
```

Achieving above functionality using jsp:include action element will look like this:

<jsp:include page = "one.jsp" flush = "true" />

37.5.2 JSP forward action Element

It is used to forward request to another resource. The format of jsp:forward action is:

<jsp:forward page = "one.jsp" />

jsp:forward is being equivalent to following code of scriptlet. For example to forward the request to *one.jsp* , scriptlet code will look like:

```
<%
   RequestDispatcher rd = request.getRequestDispatcher("one.jsp");
   rd.forward(request, response);
%>
```

37.6 References:

- Java A Lab Course by Umair Javed.
- Core Servlets and JavaServer Pages by Marty Hall

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 38: JSP Custom Tags

To begin with, let's review our last code example of lecture 36 i.e. Displaying course outline. We incorporated JavaBeans to minimize the database logic from the JSP. But still, we have to write some lines of java code inside java.jsp & web.jsp. As discussed earlier, JSPs are built for presentation purpose only, so all the other code that involves business and database logic must be shifted elsewhere like we used JavaBeans for such purpose.

There is also another problem attached to it. Generally web page designers which have enough knowledge to work with HTML and some scripting language, faced lot of difficulties in writing some simple lines of java code. To overcome these issues, java provides us the mechanism of custom tags.

38.1 Motivation

To give you an inspiration, first have a glance over the code snippet we used in JSP of the course outline example of last lecture. Of course, not all code is given here; it's just for your reference to give you a hint.

```
<%
CourseDAO courseDAO = new CourseDAO(); ……
// iterating over ArrayList for
(…… ) {
……
…… //displaying courseoutline
}
        ……
%>
```

Can we replace all the above code with one single line? Yes, by using custom tag we can write like this:

```
<mytag:coursetag          pageName="java" />
```

By only specifying the course/page name, this tag will display the course outline in tabular format. Now, you must have realized how significant changes custom tags can bring on.

38.2 What is a Custom Tag?

- In simplistic terms, "a user defined component that is used to perform certain action".

This action could be as simple as displaying *"hello world"* or it can be as complex as displaying course outline of selected course after reading it from database.

- It provides mechanism for encapsulating complex functionality for use in JSPs. Thus facilitates the non-java coders.
- We already seen & used many built in tags like:
    - o < jsp:useBean …… />
    - o < jsp:include …… />
    - o < jsp:forward …… /> etc.

38.3 Why Build Custom Tag?

- We introduced action <jsp:useBean> and JavaBeans to incorporate complex, encapsulated functionality in a JSP.
- However, JavaBeans cannot manipulate JSP content and Web page designers must have some knowledge to use JavaBeans in a page
- With Custom tags, it is possible for web page designers to use complex functionality without knowing any java

38.4 Advantages of using Custom Tags

- Provides cleaner separation of processing logic and presentation, than JavaBeans.
- Have access to all JSP implicit objects like out, requestetc.
- Can be customized by specifying attributes.

38.5 Types of Tags

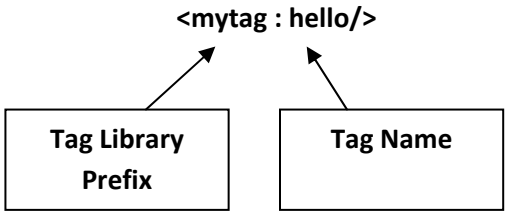
Three types of can be constructed. These are:

1. Simple Tag
2. Tag with Attribute
3. Tag with Body

38.5.1 Simple Tag

A simple tag has the following characteristics:

- Start and End of tag
- No body is specified within tag
- No attributes
- For example



38.5.2Tag with Attributes

A tag with attributes has the following characteristics:

- Start and End of tag
- Attributes within tag
- No body enclosed
- For example

< mytag:hello **attribute = "value"** />

38.5.3Tag with Body

A tag with body has the following characteristics:

- Start and End of tag
- May be attributes
- Body enclosed within tag
- For example

< mytag:hello optional_attributes …… >

some body

</ mytag:hello >

38.6 Building Custom Tags

So far, we have used many built-in tags. Now the time has come to build your own one. Custom tags can be built either by using JSP 1.2 specification or JSP 2.0 (latest) specification.

To develop custom tags using JSP 1.2 involves lot of cumbersome (too difficult for James Gossling also). However, JSP 2.0 brings lots of goodies like

- Simple tag extensions to build custom tags
- Integrated Expression Language (will be discussed in coming lecture)
- Also provides an alternate mechanism for building custom tags using tag files (.tag)
- Improved XML syntax etc.

38.6.1 Steps for Building Custom Tags

The following steps are used in order to develop your own custom tag. These are:

1. Develop the Tag Handler class
2. Write Tag library Descriptor (.tld) file
3. Deployment

38.6.2 Develop the Tag Handler class

- Tag Handler is also a java class that is implicitly called when the associated tag is encountered in the JSP.
- Must implement SimpleTaginterface
- Usually extend from SimpleTagSupport class that has already implemented SimpleTaginterface.
- For example,

```
public class MyTagHandler extends SimpelTagSupport {

……
……
}
```

- **doTag()**method
  - By default does nothing
  - Need to implement / override to code/write functionality of tag
  - Invoked when the end element of the tag encountered.
- JSP implicit objects (e.g. out etc) are available to tag handler class through pageContextobject.
- pageContextobject can be obtained using getJspContext()method.
- For example to get the reference of implicit outobject, we write.
  - PageContext pc = (PageContext) getJspContext();
  - JspWriter out = pc.getOut();

38.6.3 Write Tag Library Discriptor (.tld) file

- It is a XML based document.

- Specifies information required by the JSP container such as:
  - Tag library version
  - JSP version
  - Tag name
  - Tag Handler class name
  - Attribute names etc.

Note: If you are using any IDE (like netBeans® 4.1, in order to build custom tags, the IDE will write .tldfile for you.

38.6.4 Deployment

- Place Tag Handler class in myapp/WEB-INF/classes folder of web application.
- Place .tldfile in myapp/WEB-INF/tldsfolder of web application.

Note: Any good IDE will also perform this step on your behalf

38.7 Using Custom Tags

Use taglibdirective in JSP to refer to the tag library. For example

<%@ taglib uri="TLD file name" prefix="mytag" %>

The next step is to call the tag by its name as defined in TLD. For example, if tag name is *hello* then we write:

< mytag:hello />

where *mytag* is the name of prefix specified in taglibdirective.

What actually happened behind the scenes? Container calls the doTag() method of appropriate tag handler class. After that, Tag Handler will write the appropriate response back to the page.

Example Code: Building simple tag that displays "Hello World"

Enough we have talked about what are custom tags, their types. Now, it is a time to build a custom tag that displays *"Hello World"*.

Approach

- Extend Tag Handler class from SimpleTagSupport class and override

doTag()method
- Build TLD file
- Deploy

Note: As mentioned earlier, if you are using any IDE (like netBeans® 4.1), the last two steps will be performed by the IDE.

WelcomeTagHandler.java

```
package vu;
// importing required packages
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

// inheriting from SimpleTagSupport class
public class WelcomeTagHandler extends SimpleTagSupport {
// overriding doTag() method
public void doTag() throws JspException {
// obtaining the reference of out implicit object
PageContext pageContext = (PageContext)getJspContext(); JspWriter out =
pageContext.getOut();
try {
out.println(" Hello World ");
} catch (java.io.IOException ex) {
throw new JspException(ex.getMessage());
}
} // end doTag() method
} // end WelcomeTagHandler class
```

customtags.tld

If using IDE, this file will be written automatically. In this file you specify the tag name along with Tag Handler class.

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web- jsptaglibrary_2_0.xsd">
<tlib-version>1.0</tlib-version>
<short-name>mytag</short-name>
<!—the value of uri will be used in JSP to refer to this tld -->
<uri>/WEB-INF/tlds/customtags</uri>
<!—
Specifying the tag name and tag class. Also mentioning that this tag has no body
-->
<tag>
<name>welcome</name>
```

```
<tag-class>vu.WelcomeTagHandler</tag-class>
<body-content>empty</body-content>
</tag>

</taglib>
```

index.jsp

```
<%--
using taglib directive, specifying the tld file name as well as
prefix. Note that you you use any value for the prefix attribtute
--%>
<%@taglib uri="/WEB-INF/tlds/customtags.tld" prefix="mytag" %>

<html>
<body>
<h2>A Simple Tag Example</h2>
<h3>
<%-- calling welcome tag with the help of prefix --%>
<mytag:welcome />
</h3>
</body>
</html>
```

38.8 Building tags with attributes

If you want to build a tag that can also take attributes, for example

<mytag:hello **attribute=”value”** />

To handle attributes, you need to add

Instance variables and Corresponding *setter* methods

Behind the scenes, container will call these setter methods implicitly and pass the value of the custom tag attribute as an argument.

Example Code: Building tag with attribute

In this example, we will modify our course outline example to incorporate tags. Based on attribute value, the tag will display the respective course outline in tabular format.

Approach

- Extend Tag Handler class from SimpleTagSupportclass
  - Add instance variable of type String
  - Write setter method for this attribute
  - Override doTag()method
- Build TLD file
- Deploy

CourseOutlineBean.java

This is the same file used in the last example

```
package vubean;
import java.io.*;
public class CourseOutlineBean implements Serializable{
private int sessionNo;
private String topic;
private String assignment;
// no argument constructor
public CourseOutlineBean() {
sessionNo = 0;
topic     = "";
assignment = "";
}
// setters
public void setSessionNo(int s){
sessionNo = s;
}
public void setTopic(String t){
topic = t;
}
public void setAssignment(String a){
assignment = a;
}
// getters
public int getSessionNo( ){
return sessionNo;
}
public String getTopic( ){
return topic;
}

public String getAssignment( ){
```

```
}
} // end class
```

CourseDAO.java

No changes are made to this file too.

```
package vu;
import java.io.*;
import java.sql.*;
import java.util.*;
import vubean.*;
public class CourseDAO implements Serializable{
private Connection con;
public CourseDAO() {
establishConnection();
}
//********** establishConnection method ********************
// method used to make connection with database
private void establishConnection(){
try{
// establishing conection
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
String conUrl = "jdbc:odbc:CourseDSN";
con = DriverManager.getConnection(conUrl);
}catch(Exception ex){
System.out.println(ex);
}
}
//*********** retrieveCourseList method ********************
public ArrayList retrieveCourseList(String cName){
ArrayList courseList = new ArrayList();
try{
String sql = " SELECT sessionNo, topic, assignment " + " FROM Course, SessionDetail" +
" WHERE courseName = ? " +
" AND Course.courseId = SessionDetail.courseID ";
PreparedStatement pStmt = con.prepareStatement(sql);
pStmt.setString(1, cName);
ResultSet rs = pStmt.executeQuery();
int sNo; String topic; String assignment;
while ( rs.next() ) {
sNo      = rs.getInt("sessionNo");
topic     = rs.getString("topic");
assignment = rs.getString("assignment");
```

```
if (assignment == null){
assignment = "";
}
// creating a CourseOutlineBean object
CourseOutlineBean cBean = new CourseOutlineBean();

cBean.setSessionNo(sNo);
cBean.setTopic(topic);
cBean.setAssignment(assignment);
// adding a bean to arraylist
courseList.add(cBean);
}
}catch(Exception ex){
System.out.println(ex);
} finally {
// to close connection
releaseResources();
}
// returning ArrayList object
return courseList;
} // end retrieveCourseOutline
//********** releaseResources method ********************
private void releaseResources(){
try{
if(con != null){
con.close();
}
}catch(Exception ex){
System.out.println();
}
} // end releaseResources
}// end CourseDAO
```

MyTagHandler.java

The tag handler class uses JavaBeans (CourseOutlineBean.java& CourseDAO.java), and includes the logic of displaying course outline in tabular format.

```
package vutag;
// importing package that contains the JavaBeans
import vubean.*;
import vu.*;
import javax.servlet.jsp.tagext.*;
```

```
import javax.servlet.jsp.*;
import java.util.*;
public class MyTagHandler extends SimpleTagSupport {
/*
Declaration of pageName property.
*/
private String pageName;

public void doTag() throws JspException {

CourseDAO courseDAO = new CourseDAO();
ArrayList courseList = courseDAO.retrieveCourseList(pageName);
// to display course outline in tabular form, this method is
// used – define below
display(courseList);
}
/*
Setter for the pageName attribute.
*/
public void setPageName(java.lang.String value) {
this.pageName = value;
}
/*
display method used to print courseoutline in tabular form
*/
private void display(ArrayList courseList)throws JspException{
PageContext pc = (PageContext)getJspContext();
JspWriter out = pc.getOut();
try{
// displaying table headers
out.print("<TABLE BORDER=1 >");
out.print("<TR>");
out.print("<TH> Session No </TH>");
out.print("<TH> Topic</TH>");
out.print("<TH> Assignment </TH>");
out.print("</TR>");
// loop to iterate over courseList
for (int i=0; i<courseList.size(); i++){
CourseOutlineBean courseBean = (CourseOutlineBean)courseList.get(i);
// displaying one row
out.print("<TR>");
out.print("<TD>" + courseBean.getSessionNo() + "</TD>"); o
ut.print("<TD>" + courseBean.getTopic() + "</TD>");
out.print("<TD>" + courseBean.getAssignment() + "</TD>");
```

```
out.print("</TR>");
}catch(java.io.IOException ex){
throw new JspException(ex.getMessage());
}
}
} // end clas MyTagHandler.java
```

mytaglibrary.tld

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web- jsptaglibrary_2_0.xsd">
<tlib-version>1.0</tlib-version>
<short-name>mytaglibrary</short-name>
<!—the value of uri will be used in JSP to refer to this tld -->
<uri>/WEB-INF/tlds/mytaglibrary</uri>
<!—
Specifying the tag name and tag class. Also mentioning that this tag has no body
-->
<tag>
<name>coursetag</name>
<tag-class>vutag.MyTagHandler</tag-class>
<body-content>empty</body-content>
<!—
Specifying the attribute name and its type
-->
<attribute>
<name>pageName</name>
<type>java.lang.String</type>
</attribute>
</tag>
</taglib>
out.print("</TABLE>");
```

index.jsp

This page is used to display the course options to the user in the radio button form.

```
<html>
<body>
<h2>Select the page you want to visit</h2>
<form name="myForm" action="controller.jsp" >
<h3>
```

```
<input type="radio" name = "page" value="web"/> Web Design & Development
</h3>
<br>
<h3>
<input type="radio" name = "page" value="java"/>
Java
</h3>
<br>
<input type="submit" value="Submit" />
</form>
</body>
</html>
```

controller.jsp

Based upon the selection made by the user, this page will redirect the user to respective pages. Those are web.jspand java.jsp

```
<html>
<body>
<!-- scriptlet -->
<%
String pageName = request.getParameter("page");
if (pageName.equals("web")) {
response.sendRedirect("web.jsp");
} else if (pageName.equals("java") )    {
response.sendRedirect("java.jsp");
}
%>
</body>
</html>
```

java.jsp

```
<%-- using taglib directive, specifying the tld file and prefix -
-%>
<%@taglib uri="/WEB-INF/tlds/mytaglibrary.tld" prefix="mytag"%>

<html>
<body>
<center>
<h2> Welcome to Java Learning Center </h2>
<h3> Course Outline</h3>
```

```
<%--
calling coursetag and specifying java as attribute
value
--%>
<mytag:coursetag pageName="java" />
</center>
</body>
</html>
```

web.jsp

```
<%-- using taglib directive, specifying the tld file and prefix -
-%>
<%@taglib uri="/WEB-INF/tlds/mytaglibrary.tld" prefix="mytag"%>
<html>
<body>
<center>
<h2> Welcome to Java Learning Center </h2>
<h3> Course Outline</h3>
<%--
calling coursetag and specifying java as attribute value
--%>
<mytag:coursetag pageName="java" />
</center>
</body>
</html>
```

38.9 References:

- Java A Lab Course by Umair Javed.
- Core Servlets and JavaServer Pages by Marty Hall

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 39: MVC + Case Study

We have covered an adequate amount of Servlets and JSPs in detail. Now, the time has come to learn different architectures that are most commonly used for the sake of web development. These architectures also help us to understand where these components best fit in. In this handout, we'll cover the most widely used/popular architecture i.e. *Model*

View Controller
(MVC).

A small case study *"Address Book"* is also part of this handout that is based on MVCModel 1. Before moving on to MVC, let's see what *error pages* are and how they are used?

39.1 Error Page

Error Pages enables you to customize error messages. You can even hide them from the user's view entirely, if you want. This also makes possible to maintain a consistent look and feel throughout an application, even when those dreaded error messages are thrown.

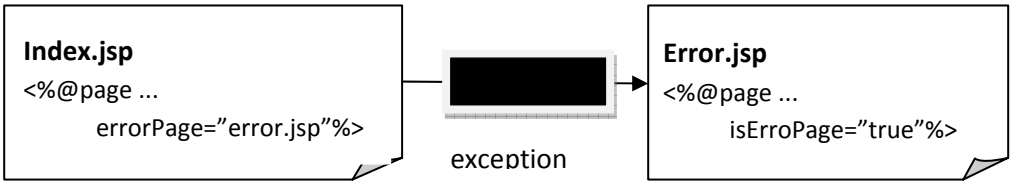
By means of *page directive*, a JSP can be given the responsibility of an Error page. An Error JSP is called by the web server when an uncaught exception gets occurred. This exception is passed as an instance of java.lang.Throwableto Error JSP (also accessible via implicit exceptionobject).

39.1.1Defining and Using Error Pages

isErrorPageattribute of a *page directive* is used to declare a JSP as an error page.

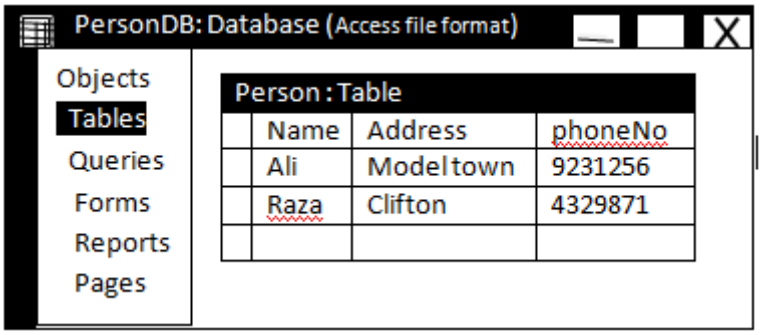
JSP pages are informed about the error page by setting errorPageattribute of page directive

In the figure below, error.jspis defined as JSP Error page and index.jspis informed to call error.jsp if any uncaught exception rose. This is done by setting attributes errorPage and isErrorPageof the page directive on these JSPs.



39.2 Case Study – Address Book

What we have learned is going to be implemented in this Address Book example. Here MS-Access is being used as DBMS. This database will have only one table, *Person* with following attributes



39.2.1 Ingredients of Address Book

Java Beans, Java Server Pages and Error Page that are being used in this Address Book Example are: -

Java Beans

- **PersonInfo –** Has following attributes:
  - name
  - address
  - phoneNum
- **PersonDAO**
  - Encapsulates database logic.
  - Therefore, it will be used to save and retrieve PersonInfo data.

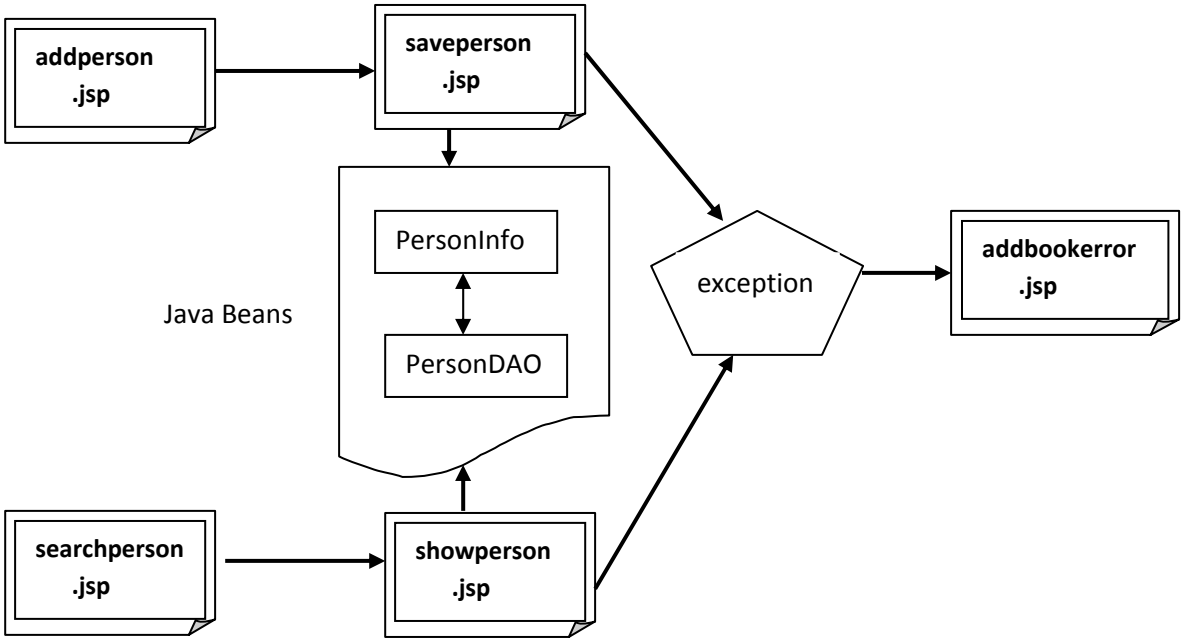
Java Server Pages

- **addperson.jsp**
  - Used to collect new person info that will be saved in database.
- **saveperson.jsp**
  - Receives person info from addperson.jsp
  - Saves it to database
- **searchperson.jsp**
  - Used to provide search criteria to search Person's info by providing name
- **showperson.jsp**
  - This page receive person's name from searchperson.jspto search in
  - database
  - Retrieves and displays person record found against person name

Error Page

- **addbookerror.jsp**
  - This page is declared as an error page and used to identify the type of exception.

- In addition to that, it also displays the message associated with the received exception to the user.



addperson.jsp takes person's information from the user and sends it to saveperson.jsp. After receiving request, saveperson.jspmakes an object of PersonInfousing received information and saves it into the database using PersonDAO Java bean.

Similarly, searchperson.jsptakes search criteria (name) from the user and passes it to showperson.jspthat searches the record in database using PersonDAOand shows the results to the user.

If any uncaught exception is generated on these JSP, addbookerror.jspis called implicitly, which displays an appropriate message to the user after identifying the exception type.

Code for the Case Study

Let's have a look on the code of each component used in the case study; first start from JavaBeans.

PersonInfo

PersonInforepresents the record of one person and its objects are used to interrupt the information about persons.

```
package vu;
import java.io.*;
public class PersonInfo implements Serializable{
```

```
private String name;
private String address;
private int phoneNum;
// no argument constructor
public PersonInfo() {
name = "";
address = "";
phoneNum = 0;
}
// setters
public void setName(String n){
name = n;
}
public void setAddress(String a){
address = a;
}
public void setPhoneNum(int pNo){
phoneNum = pNo;
}
// getters
public String getName( ){
return name;
}
public String getAddress( ){
return address;
}

public int getPhoneNum( ){
return phoneNum;
}
} // end class PersonInfo
```

PersonDAO

This class will help in retrieving and storing person's records in database. The code is given below:

```
package vu;
import java.util.*;
import java.sql.*;
public class PersonDAO{
private Connection con;
// default constructor
public PersonDAO() throws ClassNotFoundException , SQLException
{
```

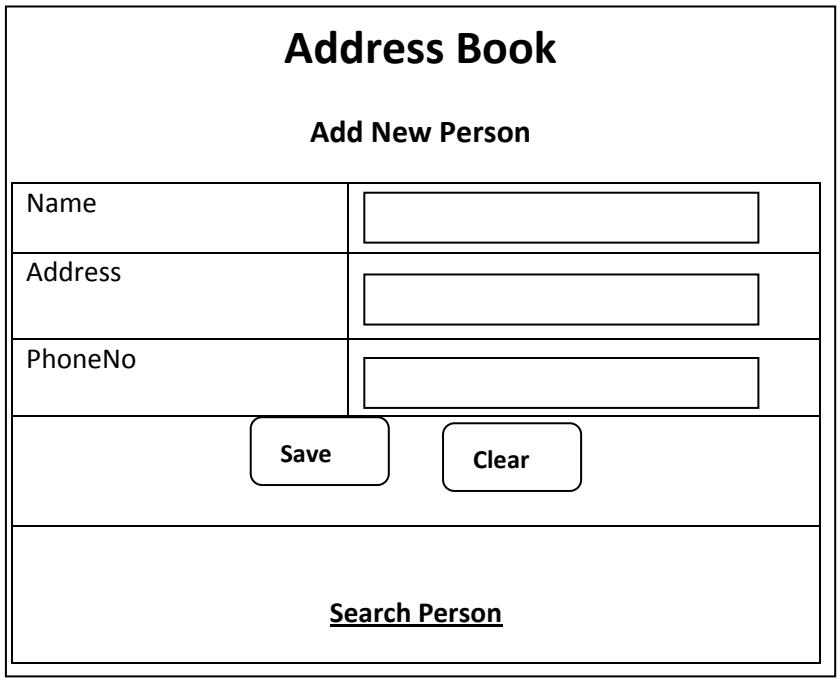
```
establishConnection();
}
// method used to establish connection with db
private void establishConnection() throws ClassNotFoundException , SQLException
{
// establishing conection
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
String conUrl = "jdbc:odbc:PersonDSN";
con = DriverManager.getConnection(conUrl);
}
// used to search the person records against name and returns
// the ArrayList that contains only those PersonInfo objects
// which matches the search criteria i.e. name
public ArrayList retrievePersonList(String pName) throws SQLException
{
ArrayList personList = new ArrayList();
// preparing query
String sql = " SELECT * FROM Person WHERE name = ?";
PreparedStatement pStmt = con.prepareStatement(sql);
pStmt.setString( 1, pName);
// executing query
ResultSet rs = pStmt.executeQuery();
String name; String add; int pNo;
while ( rs.next() ) {
name = rs.getString("name");
add = rs.getString("address");
pNo = rs.getInt("phoneNumber");
// creating a CourseOutlineBean object
PersonInfo personBean = new PersonInfo();
personBean.setName(name);
personBean.setAddress(add);
personBean.setPhoneNum(pNo);
// adding a bean to arraylist
personList.add(personBean);
} // end while
return personList;
} // end retrievePersonList
// this method accepts an object of PersonInfo, and stores it
// into the database
public void addPerson(PersonInfo person) throws SQLException{
String sql = " INSERT INTO Person(name, address, phoneNumber) VALUES (?, ?, ?)";
PreparedStatement pStmt = con.prepareStatement(sql); String name = person.getName();
String add = person.getAddress();
```

```
int pNo = person.getPhoneNum();
pStmt.setString( 1 , name );
pStmt.setString( 2 , add );
pStmt.setInt( 3 , pNo );
pStmt.executeUpdate();
} // end addPerson
// overriding finalize method to release acquired resources
public void finalize( ) {
try{
if(con != null){
con.close();
}
}catch(SQLException sqlex){
System.out.println(sqlex);
}
} // end finalize
} // end PersonDAO class
```

Now let's take a look at the code for JSP pages

addperson.jsp

This JSP page gets person record's information from the user. It contains three Input Fields for name, address and phone number as shown in the diagram. This page sends this information to saveperson.jspfor further processing.

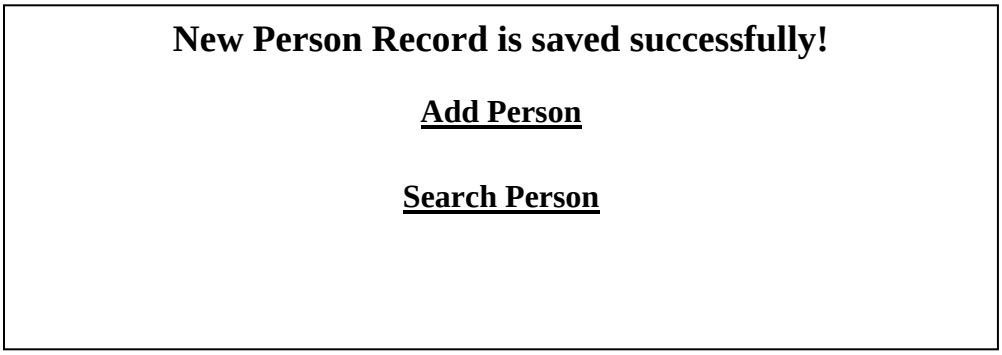


The code that is used to generate the above page is given below:

```
<%--
Although there are no chances of exception to arise on this page, for consistency, error page is defined
on top of all JSPs
--%>
<%@page errorPage="addbookerror.jsp" %>
<html>
<body>
<center>
<h2> Address Book </h2>
<h3> Add New Person</h3>
<%-- Form that contains Text input fields and sending it to saveperson.jsp
--%>
<form name ="register" action="saveperson.jsp" />
<TABLE BORDER="1" >
<TR>
<TD> <h4 > Name </h4> </TD>
<TD> <input type="text" name="name" />   </TD>
</TR>
<TR>
<TD> <h4> Address </h4> </TD>
<TD> <input type="text" name="address" />          </TD>
</TR>
<TR>
<TD> <h4>Phone Number</h4> </TD>
<TD> <input type="text" name="phoneNum" />      </TD>
</TR>
<TR>
<TD COLSPAN="2" ALIGN="CENTER" >
<input type="submit" value="save" />
<input type="reset" value="clear" />
</TD>
</TR>
</TABLE>
</form>
<h4>
<%-- A link to searchperson.jsp --%>
<a href="searchperson.jsp" > Search Person </a>
</h4>
</center>
</body>
</html>
```

saveperson.jsp

This JSP page gets data from the addperson.jsp, makes an object of PersonInfo and saves it to the database using PersonDAOclass. Apart from these, it also displays an informative message to the user if new person record is saved successfully into thedatabase and two hyperlinks to navigate on to the desired pages as shown in the following diagram:



The code of this page is given below:

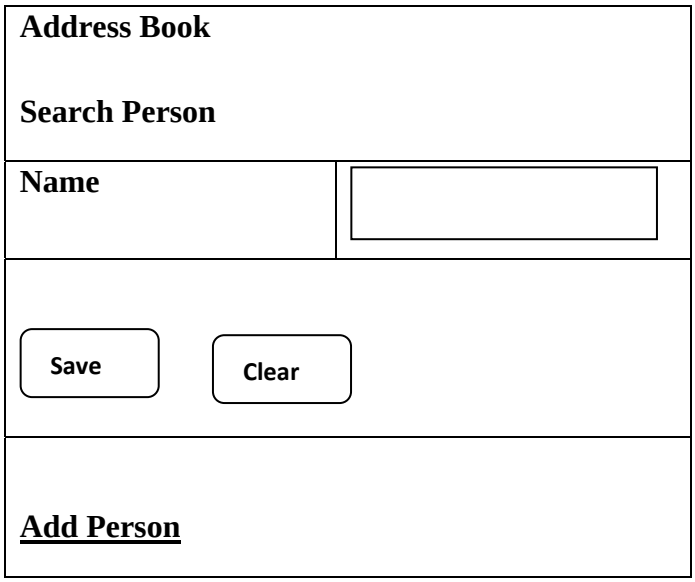
```
<%-- defining error page --%>
<%@page errorPage="addbookerror.jsp" %>
<%@ page import="java.sql.*" %>
<html>
<body>
<%-- creating PersonDAO object and storing in page scope --%>
<jsp:useBean id="pDAO" class="vu.PersonDAO" scope="page" />
<%-- creating PersonBean object and storing in page scope --%>
<jsp:useBean id="personBean" class="vu.PersonInfo" scope = "page"
/>
<%--
setting all properties of personBean object with input parameters using *
--%>
<jsp:setProperty name="personBean" property="*" />

<%--
to save Person record into the database, calling addperson
method of PersonDAO
--%>
<%
pDAO.addPerson(personBean);
%>
<center>
```

```
<h3> New Person Record is saved successfully!</h3>
<h4>
<a href="addperson.jsp" > Add Person </a>
</h4>
<h4>
<a href="searchperson.jsp" > Search Person </a>
</h4>
</center>
</body>
</html>
```

searchperson.jsp

It gets search criteria from the user (i.e. name) and sends it to showperson.jspto display the search results. The outlook of the page is given below:



used to generate the above page given page is:

```
<%-- defining error page --%>
<%@page errorPage="addbookerror.jsp" %>

<html>
<body>
<center>
<h2> Address Book </h2>
<h3> Search Person</h3>

<%--
Form that contains Text input field and sending it to showperson.jsp
--%>
```

```
<form name ="search" action="showperson.jsp" />
<TABLE BORDER="1" >
<TR>
<TD> <h4 >Name</h4>        </TD>
<TD> <input type="text" name="name" />   </TD>
</TR>
<TR>
<TD COLSPAN="2" ALIGN="CENTER"">
<input type="submit" value="search" />
<input type="reset" value="clear" />
</TD>
</TR>
</TABLE>
</form>
<h4>
<a href="addperson.jsp" > Add Person </a>
</h4>
</center>
</body>
</html>
```

showperson.jsp

showperson.jspreceives search criteria (i.e. *name*) from the searchperson.jsp, that is entered by the user to find the matching record. This page retrieves the complete list of matching records from the database using PersonDAO, and shows them to the user.

This following figure gives you the sight, when person named “saad” is searched.

Address Book

Following Result meet your research criteria

Name	Address	PhoneNo.
Saad	Gulberg	9700234

Add Person

Search Person

Below, the code of showperson.jsp is given:

```
<%-- defining error page --%>
<%@page errorPage="addbookerror.jsp" %>
<%-- importing required packages --%>
<%@page import="java.util.*" %>
<%@page import="vu.*" %>
<html>
<body>
<center>
<h2> Address Book </h2>
<h3> Following results meet your search criteria</h3>
<TABLE BORDER="1" >
<TR>
<TH> Name </TH>
<TH> Address </TH>
<TH> PhoneNum </TH>
</TR>
<jsp:useBean id="pDAO" class="vu.PersonDAO" scope="page" />
<%
// getting search criteria sent by searchperson.jsp
String pName = request.getParameter("name");
// retrieving matching records from the Database using
// retrievePersonList() method of PersonDAO
ArrayList personList = personDAO.retrievePersonList(pName);
PersonInfo person = null;
// Showing all matching records by iterating over ArrayList
for(int i=0; i<personList.size(); i++) {
person = (PersonInfo)personList.get(i);
%>
<TR>
<TD> <%= person.getName()%> </TD>
<TD> <%= person.getAddress()%> </TD>
<TD> <%= person.getPhoneNum()%> </TD>
</TR>
<%
} // end for
%>
</TABLE >
<a href="addperson.jsp" > Add Person </a>
<a href="searchperson.jsp" > Search Person </a>
</center>
</body>
```

```
</html>
```

addbookerror.jsp

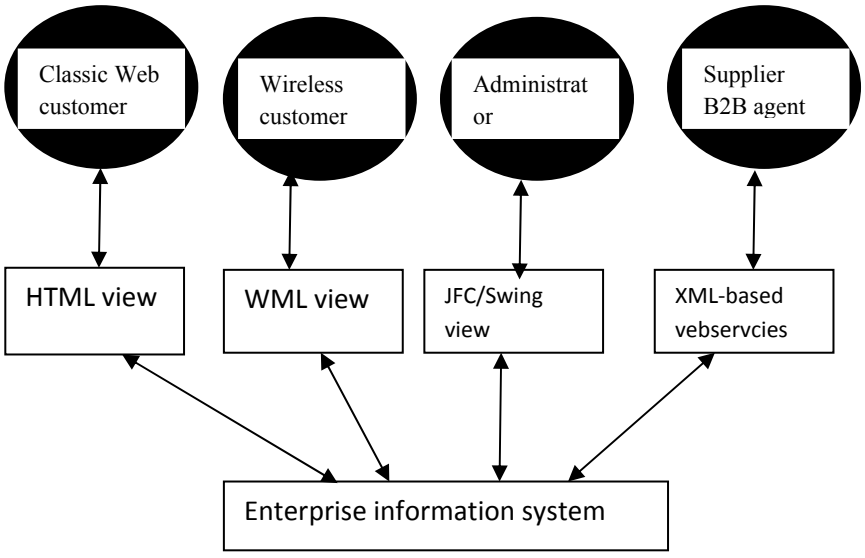
This JSP error page is called implicitly by all other JSP pages whenever any uncaught/unhandled exception occurs. It also finds out the type of the exception that is generated,and shows an appropriate message to the user:

```
<%-- indicating that this is an error page --%>
<%@page isErrorPage="true" %>
<%-- importing class --%>
<%@page import = "java.sql.SQLException" %>
<html>
<head>
<title>Error</title>
</head>
<body>
<h2>
Error Page
</h2>
<h3>
<%-- scriptlet to determine exception type --%>
<%
if (exception instanceof SQLException) {
%>
An SQL Exception
<%
} else if (exception instanceof ClassNotFoundException){
%>
A Class Not Found Exception
<%
} else {
%>
A Exception
<%
} // end if-else
%>
<%-- end scriptlet to determine exception type --%>
occured while interacting with the database
</h3>
<h3>
The Error Message was
```

```
<%= exception.getMessage() %>
</h3>
<h3 >  Please Try Again Later!        </h3>
<%--
hyperlinks to return back to addperson.jsp or
searchperson.sjp
--%>
<h3>
<a href="controller.jsp?action=addperson" >
Add Person
</a>
<a href="controller.jsp?action=searchperson" >
Search Person
</a>
</h3>
</body>
</html>
```

39.3 Model View Controller (MVC)

Now, more than ever, enterprise applications need to support multiple types of users with multiple types of interfaces. For example, an online store may require an HTML front for Web customers, a WML front for wireless customers, a JavaTM (JFC) / Swing interface for administrators, and an XML-based Web service for suppliers



Also, several problems can arise when applications contain a mixture of data access code,business

interdependencies between all of the components cause strong ripple effects whenever a change is made anywhere. High coupling makes classes difficult or impossible to reuse because they depend on so many other classes. Adding new data views often requires re-implementing or cutting and pasting business logic code, which then requires maintenance in multiple places. Data access code suffers from the same problem, being cut and pasted among business logic methods.

The Model-View-Controller architecture solves these problems by decoupling data access, business logic, and data presentation and user interaction. Such separation allows multiple views to share the same enterprise data model, which makes supporting multiple clients easier to implement, test, and maintain.

39.3.1 Participants and Responsibilities

The individual's responsibility of three participants (model, view & controller) is given below:

- **Model**

The model represents the state of the component (i.e. its data and the methods required to manipulate it) independent of how the component is viewed or rendered.

- **View**

The view renders the contents of a model and specifies how that data should be presented. There can be multiple views for the same model within single applications or model may have different views in different applications or operating systems.

- **Controller**

The controller translates interactions with the view into actions to be performed by the model. In a web application, they appear as GET and POST HTTP requests. The actions performed by the model include activating business processes or changing the state of the model. Based on the user interactions and the outcome of the model actions, the controller responds by selecting an appropriate view.

39.3.2 Evolution of MVC Architecture

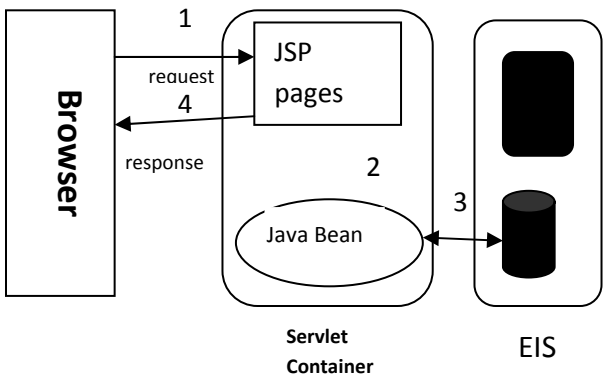
In the beginning, we used no MVC. Then we had *MVC Model 1* and *MVC Model 2* architectures. And people came up with so called web application frameworks such as *Apache Struts* based on Model 2 architecture. And finally we have a standard web based application framework i.e. *JavaServer Faces (JSF).*

In this handout, we'll only talk about MVC Model 1.

39.3.2.1 MVC Model 1

A Model 1 architecture consists of a Web browser directly accessing Web-tier JSP pages.The JSP pages access JavaBeans that represent the application model. And the next view to display

(JSP page, servlet, HTML page, and so on) is determined either by hyperlinks selected in the source document or by request parameters.



In Model 1 architecture, view selection is decentralized, because the current page being displayed determines the next page to display. In addition, each JSP page or servlet processes its own inputs (parameters from GET or POST). And this is hard to maintain, for example, if you have to change the view selection, then several JSP pages need to be changed. In some Model 1 architectures, choosing the next page to display occurs in scriptlet code, but this usage is considered poor form.

In MVC Model 1 architecture, the JSP page alone is responsible for processing the incoming request and replying back to the client. There is still separation of presentation from content, because all data access is performed using JavaBeans.

Although the Model 1 architecture should be perfectly suitable for simple applications, it may not be desirable for complex implementations. Random usage of this architecture usually leads to a significant amount of scriptlets or Java code embedded within the JSP page, especially if there is a significant amount of request processing to be performed. While this may not seem to be much of a problem for Java developers, it is certainly an issue if your JSP pages are created and maintained by designers which are only aware of HTML and some scripting language.

Note: Probably some of you must be thinking about the case study discussed earlier inthis handout. Indeed, it is based on MVC Model 1 architecture.

39.4 References:

- Java A Lab Course by Umair Javed
- Java BluePrints - J2EE Patterns
  http://java.sun.com/blueprints/patterns/MVC-detailed.html
- Exploring the MVC Design Pattern
  http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html

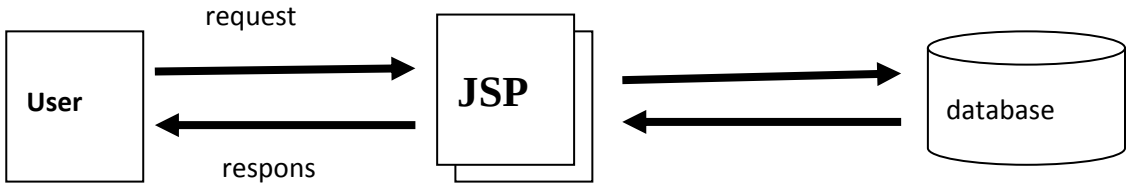
Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 40: Model 2 Architecture MVC

We have studied *page-centric approach* and *page-with-bean approach* until now. You must be wondering when we had covered these. Probably these buzz words are new one for you but we already covered these topics. Let's review these once again.

40.1 Page-Centric Approach

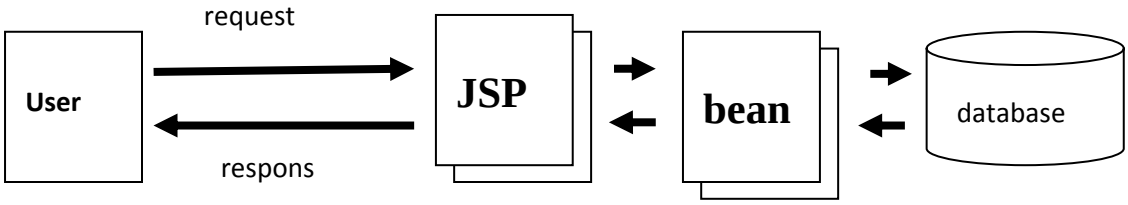
A web application that is collection of JSPs. Generally this approach is followed to get started with developing web applications. This approach is represented in the following diagram:



The page-centric approach has lot of draw backs such as the code becomes a mixture of presentation, business and data access logic. The maintenance and up-gradation of the application becomes a nightmare. Scaling of such kind of application is also difficult and lots of code is also get duplicated.

40.1.1 Page-with-Bean Approach (MVC Model1)

This approach is different from page-centric approach in a way that all the business logic goes into JavaBeans. Therefore, the web application is a collection of JSPs and JavaBeans. But still this approach is insufficient to separate different kind of logics. We have made an address book example in the last handout using this approach.



40.2 MVC Model 2 Architecture

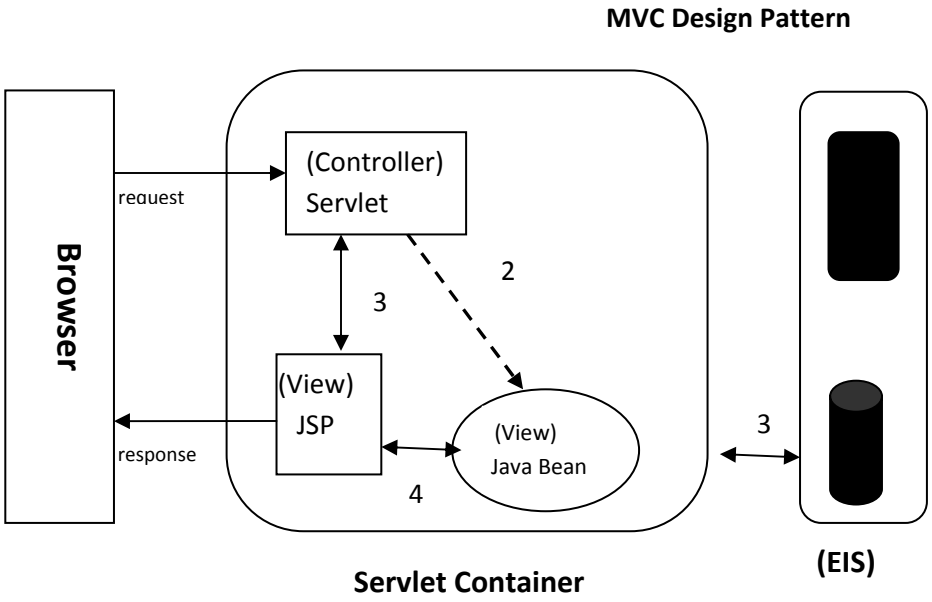
This architecture introduces a controller. This controller can be implemented using JSP or servlet. Introducing a controller gives the following advantages:
It centralizes the logic for dispatching requests to the next view based on:

- The Request URL
- Input Parameters
- Application state

It gives the single point of control to perform security checks and to record logging information It also encapsulates the incoming data into a form that is usable by the back-end MVC model.

We'll discuss it with the help of an example.

The following figure will help you to understand the architecture and functioning of the application that is built using MVC Model 2 architecture.



The client (browser) sends all the requests to the controller. Servlet/JSP acts as the *Controller* and is in charge of the request processing and creation of any beans or objects*(Models)* used by the JSP.

JSP is working as *View* and there is not much processing logic within the JSP page itself,it is simply responsible for retrieving objects and/or beans, created by the Servlet,extracting dynamic content from them and put them into the static templates.

40.3 Case Study: Address Book using MVC Model 2

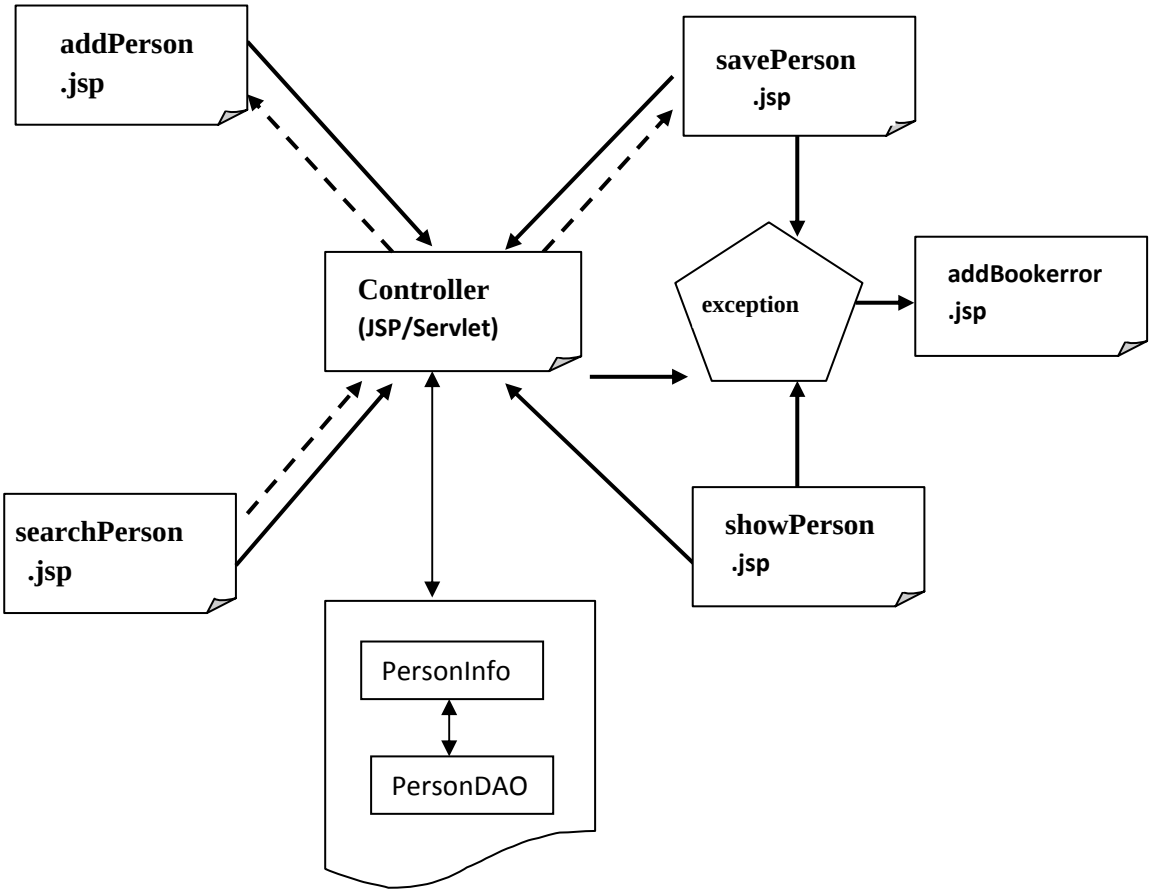
The address book example that is built using page-with-bean approach will be modified to incorporate controller. We'll show you how to implement controller using JSP as well as with servlet. Let's first incorporate controller using JSP.

40.3.1 Introducing a JSP as Controller

Add another JSP (controller.jsp) that

- Acts as a controller Recieves requests from addperson.jsp& searchperson.jsp
- Identifies the page which initiates the request
- Uses JavaBeans to save or search persons to/from database
- Forwards or redirects the request to appropriate (saveperson.jspor showperson.jsp) page.

The program flow of this example is shown in the following diagram:



As you can see in the diagram that all the requests are submitted to controller which uses the JavaBeans and forwards/redirects the user to another view (JSP)? If any exception arises on controller or JSPs, the control would automatically be transferred to addbookerror.jspto display an appropriate message.

40.3.2How controller differentiates between requests?

Most likely, you must be thinking about it. The simplest solution lies in using the consistent name (e.g. *action*) of the submit button across all the pages but with different and unique values.

The same rule applies to hyperlinks that send the *action* parameter along with value by using query string technique.

This eases the controller's job to identify which page is actually generated the request and what to do next. The controller simply retrieves the value of *action* parameter using request.getParameter()method. Now, if-else structure can be used to compare the possible values of *action* to act upon the requested task.

Now, let's first see the code of JavaBean that is used in this example.

PersonInfo

This JavaBean is used to represent one person record. The code is given below:

```
package vu;
import java.io.*;
public class PersonInfo implements Serializable{
private String name;
private String address;
private int phoneNum;
// no argument constructor
public PersonInfo() {
name = "";
address = "";
phoneNum = 0;
}

// setters
public void setName(String n){
name = n;
}
public void setAddress(String a){
address = a;
}
public void setPhoneNum(int pNo){
phoneNum = pNo;
}
// getters
public String getName( ){
return name;
```

```
}
public String getAddress( ){
return address;
}
public int getPhoneNum( ){
return phoneNum;
}
} // end class PersonInfo
```

PersonDAO

This class will help in retrieving and storing person's records in database. The code is given below:

```
package vu;
import java.util.*;
import java.sql.*;
public class PersonDAO{
private Connection con;
// default constructor
public PersonDAO() throws ClassNotFoundException , SQLException
{
establishConnection();
}
// method used to establish connection with db
private void establishConnection() throws ClassNotFoundException ,SQLException
{
// establishing conection
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
String conUrl = "jdbc:odbc:PersonDSN";
con = DriverManager.getConnection(conUrl);
}
// used to search the person records against name and returns
```

```
// the ArrayList that contains only those PersonInfo objects
// which matches the search criteria i.e. name
public ArrayList retrievePersonList(String pName) throws SQLException
{
ArrayList personList = new ArrayList();
// preparing query
String sql = " SELECT * FROM Person WHERE name = ?";
PreparedStatement pStmt = con.prepareStatement(sql);
pStmt.setString( 1, pName);
// executing query
ResultSet rs = pStmt.executeQuery();
String name; String add; int pNo;
while ( rs.next() ) {
name = rs.getString("name");
add = rs.getString("address");
pNo = rs.getInt("phoneNumber");
// creating a CourseOutlineBean object
PersonInfo personBean = new PersonInfo();

personBean.setName(name);
personBean.setAddress(add);
personBean.setPhoneNum(pNo);

// adding a bean to arraylist
personList.add(personBean);
} // end while
return personList;
} // end retrievePersonList

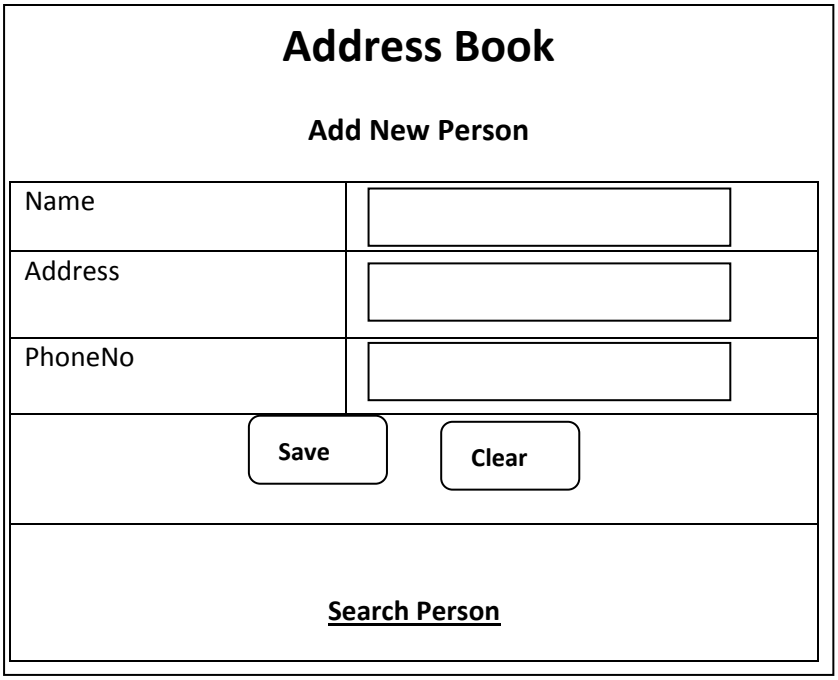
// this method accepts an object of PersonInfo, and stores it
// into the database
```

```
public void addPerson(PersonInfo person) throws SQLException{
String sql = " INSERT INTO Person(name, address, phoneNumber) VALUES (?, ?, ?)";
PreparedStatement pStmt = con.prepareStatement(sql);
String name = person.getName();
String add = person.getAddress();
int pNo = person.getPhoneNum();
pStmt.setString( 1 , name );
pStmt.setString( 2 , add );
pStmt.setInt( 3 , pNo );
pStmt.executeUpdate();
} // end addPerson
// overriding finalize method to release acquired resources
public void finalize( ) {
try{
if(con != null){
con.close();
}
}catch(SQLException sqlex){
System.out.println(sqlex);
}
} // end finalize
} // end PersonDAO class
```

addperson.jsp

This page is used for entering a new person record into the database. Note that a hyperlink is also given at the bottom of the page that takes the user to searchperson.jsp.

Note: Since we are following MVC model 2 architecture, so all the hyperlinks will also sends the request to controller first which redirects the user to requested page.



The code of above page is given below:

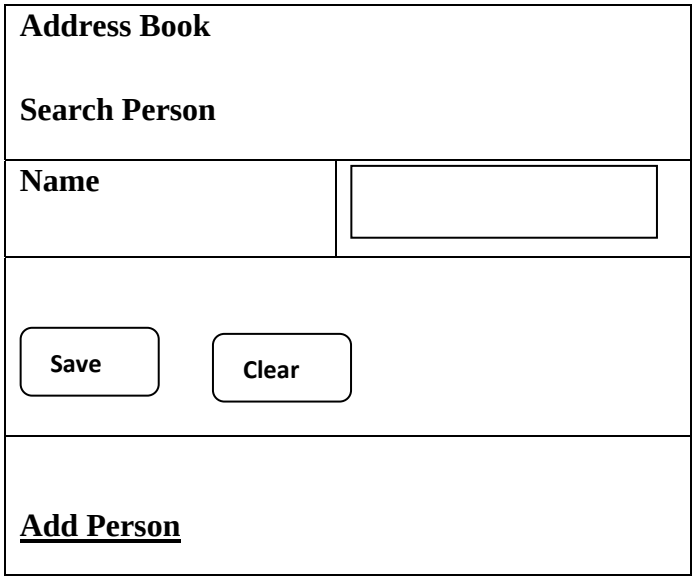
```
<%--
Although there are no chances of exception to arise on this page, for consistency, error page is defined
on top of all JSPs
--%>
<%@page errorPage="addbookerror.jsp" %>
<html>
<body>
<center>
<h2> Address Book </h2>
<h3> Add New Person</h3>
<%--
As mentioned in MVC2, all the requests are submitted to
controller, that's why action's contains the value of
"controller.jsp"
--%>
<form name ="register" action="controller.jsp" />
```

```
<TABLE BORDER="1" >
<TR>
<TD> <h4> Name </h4>        </TD>
<TD> <input type="text" name="name" />     </TD>
</TR>
<TR>
<TD> <h4> Address </h4>     </TD>
<TD> <input type="text" name="address" /> </TD>
</TR>
<TR>
<TD> <h4>Phone Number</h4>        </TD>
<TD> <input type="text" name="phoneNum" /> </TD>
</TR>
<TR>
<TD COLSPAN="2" ALIGN="CENTER">
<%--
As described above the technique to differentiate between the requests, the name of the button is
“action” with value “save”.
--%>
<input type="submit" name ="action" value="save" />
<input type="reset" value="clear" />
</TD>
</TR>
</TABLE>
</form>
<h4>
<%--
The hyperlink will also sends the request to controller
Note the action parameter with its value are also part of hyperlink using the query string technique.
--%>
<a href="controller.jsp?action=searchperson" > Search Person
```

```
</a>
</h4>
</center>
</body>
</html>
```

searchperson.jsp

This JSP is used to search the person record against name given in the text field. A hyperlink is also given at the bottom of addperson.jsp.



The code that is used to generate that above page is given below:

```
<%-- defining error page --%>
<%@page errorPage="addbookerror.jsp" %>
<html>
<body>
<center>
<h2> Address Book </h2>
<h3> Search Person</h3>
<form name ="search" action="controller.jsp" />
<TABLE BORDER="1" >
<TR>
```

```
<TD> <h4> Name </h4></TD>
<TD> <input type="text" name="name" />     </TD>
</TR>
<TR>
<TD COLSPAN="2" ALIGN="CENTER">
<%--
The name of the button is still “action” but with
different value “search”.
--%>
<input type="submit" name ="action" value="search" />
<input type="reset" value="clear" />
</TD>
</TR>
</TABLE>
</form>
<h4>
<%--
The action parameter with different value “addperson” are
part of hyperlink here as well.
--%>
<a href="controller.jsp?action=addperson" >
Add Person
</a>
</h4>
</center>
</body>
</html>
```

controller.jsp

As mentioned earlier that controller.jspidentifies the page which initiates the request and use JavaBeans to save/search persons to/from database. Also its job list includes redirecting the user

to appropriate page.

Since this JSP is doing only processing therefore no view available. Let's check it out its code:

```
<%-- defining error page --%>
<%@page errorPage="addbookerror.jsp" %>

<%-- importing required packages. package vu contains JavaBeans -
-%>
<%@page import ="java.util.*" %>
<%@page import = "vu.*" %>
<html>
<body>
<%-- declaring PersonDAO object--%>
<jsp:useBean id="pDAO" class="vu.PersonDAO" scope="page" />
<%--
scriptlet to identify JSP for redirection purpose if request comes from hyperlinks
--%>
<%
// retrieving action parameter value
// Remember that "action" is the name of buttons as well
// it is used in hyperlinks in making of query string
String action = request.getParameter("action");
// if "Add Person" hyperlink is clicked if (action.equals("addperson") ){
response.sendRedirect("addperson.jsp");
// if "Search Person" hyperlink is clicked
} else if (action.equals("searchperson")){
response.sendRedirect("searchperson.jsp");
// if "save" button is clicked of addperson.jsp
}else if (action.equals("save")) {
%>
// declaring PersonInfo obeject
```

```
<jsp:useBean id="personBean" class="vu.PersonInfo" scope="page"/>
<%--
setting all properties of personBean object with input parameters using *
--%>
<jsp:setProperty name="personBean" property="*" />
<%-- to insert record into database--%>
<%
pDAO.addPerson(personBean);
// redirecting user to saveperson.jsp response.sendRedirect("saveperson.jsp");
%>
<%-- if "search" button is clicked on searchperson.jsp --%>
<%
}else if (action.equals("search") ) {
String pName = request.getParameter("name");
ArrayList personList = pDAO.retrievePersonList(pName);
// storing personList(contains PersonInfo objects) into
// request hashmap
request.setAttribute("list", personList);
%>
<%--
forwarding request to showperson.jsp to retrieve stored arraylist
(“list”)
--%>
<jsp:forward page="showperson.jsp"  />
<%
} // end if page == search
%>
</body>
</html>
```

saveperson.jsp

This page displays a successful message indicating that person record is saved. Its also give the options to the user to move on to addperson.jspor searchperson.jsp through hyperlinks. Note that these hyperlinks also first take the user to controller.jspthen on to requested page.

New Person Record is saved successfully!

Add Person

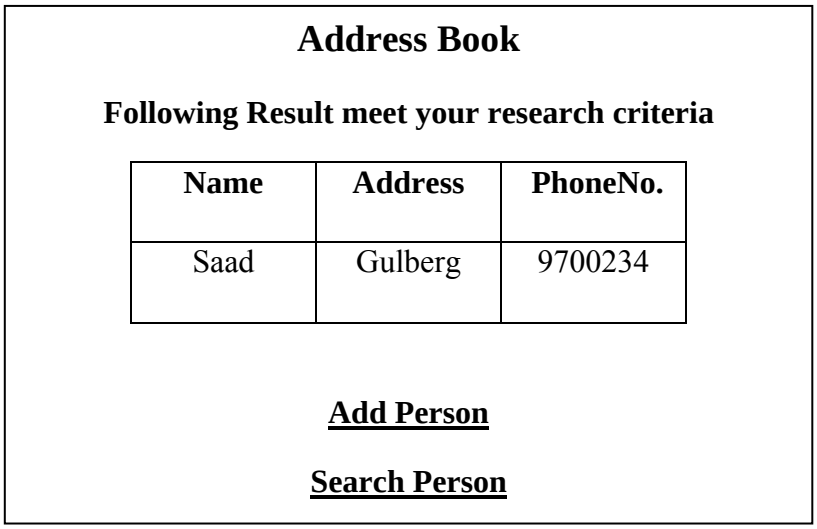
Search Person

The code of saveperson.jspis given below:

```
<%-- defining error page --%>
<%@page errorPage="addbookerror.jsp" %>
<html>
<body>
<center>
<h3> New Person Record is saved successfully!</h3>
<h4>
<a href="controller.jsp?action=addperson" > Add Person
</a>
</h4>
<h4>
<a href="controller.jsp?action=searchperson" >
Search Person
</a>
</h4>
</center>
</body>
</html>
```

showperson.jsp

This following figure gives you the view when name “saad” is searched.



Address Book

Following Result meet your research criteria

Name	Address	PhoneNo.
Saad	Gulberg	9700234

Add Person

Search Person

Below, the code of showperson.jspis given:

```
<%-- defining error page --%>
<%@page errorPage="addbookerror.jsp" %>
<%-- importing required packages --%>
<%@page import="java.util.*" %>
<%@page import="vu.*" %>
<html>
<body>
<center>
<h2> Address Book </h2>
<h3> Following results meet your search criteria</h3>
<TABLE BORDER="1" >
<TR>
<TH> Name  </TH>
<TH> Address </TH>
<TH> PhoneNum </TH>
```

```
</TR>
<%
// retrieving arraylist stored on controller.jsp to display
// PersonInfo objects
ArrayList personList = (ArrayList)request.getAttribute("list");
PersonInfo person = null;
for(int i=0; i<personList.size(); i++) {
person = (PersonInfo)personList.get(i);
%>
<%-- displaying PersonInfo details--%>
<TR>
<TD> <%= person.getName()%> </TD>
<TD> <%= person.getAddress()%> </TD>
<TD> <%= person.getPhoneNum()%> </TD>
</TR>
<%
} // end for
%>
</TABLE >
<h4>
<a href="controller.jsp?action=addperson"> Add Person </a>
<a href="controller.jsp?action=searchperson">Search Person</a>
</h4>
</center>
</body>
</html>
```

addbookerror.jsp

User will view this page only when any sort of exception is generated. The code of this page is given below:

```
<%-- indicating that this is an error page --%>
```

```
<%@page isErrorPage="true" %>
<%-- importing class --%>
<%@page import = "java.sql.SQLException" %>
<html>
<head>
<title>Error</title>
</head>
<body>
<h2>
Error Page
</h2>
<h3>
<%-- scriptlet to determine exception type --%>
<%
if (exception instanceof SQLException) {
%>
An SQL Exception
<%
} else if (exception instanceof ClassNotFoundException){
%>
A Class Not Found Exception
<%
} else {
%>
A Exception
<%
} // end if-else
%>
<%-- end scriptlet to determine exception type --%>
occured while interacting with the database
</h3>
```

```
<h3>
The Error Message was
<%= exception.getMessage() %>
</h3>
<h3 >   Please Try Again Later!          </h3>
<%--
hyperlinks to return back to adperson.jsp or
searchperson.sjp
--%>
<h3>
<a href="controller.jsp?action=addperson" > Add Person
</a>
<a href="controller.jsp?action=searchperson" >
Search Person
</a>
</h3>
</body>
</html>
```

JSP is the Right Choice as a Controller?

Since JSP that is performing the job of controller is doing only processing and there is no view available of it. It includes the logic of selecting JSP and to retrieve/store records from/to dataset using JavaBeans.

But remember the reason for introducing JSPs? JavaServer Pages are built for *presentation (view)* only so JSP is really not a good place for such kind of logic. Concluding, what's the option we have? The answer is, use Servlets as controller.

Introducing a Servlet as Controller

Remove the controller.jspfrom the previous example code and add ControllerServlet.java (a servlet) into this example.This ControllerServlet.java performs the same job that was previously performed by controller.jsp.
Besides adding ControllerServlet.java,you have to modify all the addresses which

are previously pointing to controller.jsp. For example the value of actionattribute of formtag & the address of hyperlink in all concerned pages.

If controller is defined in web.xml as an alias of ControllerServlet.java, consider the following fragment of code which shows the value of action attribute of form tag before and after introducing change.

When controller.jsp is acting as a controller

<form name ="register" **action="controller.jsp"** />

When ControllerServlet.java is acting as a controller then value of action attribute becomes:

<form name ="register" **action="controller"** />

Similarly, the following comparison shows the code of hyperlinks used in the previous example before and after making changes

When controller.jsp is acting as a controller

<a **href="controller.jsp?**action=searchperson" >
Search Person

When ControllerServlet.java is acting as a controller

<a **href="controller?**action=searchperson" >
Search Person

Passing Exceptions to an Error JSP from a Servlet

Servlet can use existing error pages (like addbookerror.jsp) to pass on the exceptions. Set the requestattribute to javax.servlet.jsp.JspExcpetion with the exception object you want to pass. After that forwards the request to error page.

For example, the following code snippet is taken from ControllerServlet.java to demonstrate how to pass SQLExceptionto addbookerror.jsp

……
……
……
……
}catch (**SQLException sqlex**){

// setting SQLException instance
request.setAttribute("javax.servlet.jsp.JspException" , sqlex);

RequestDispatcher rd =
request.getRequestDispatcher("addbookerror.jsp");

} // end catch

ControllerServlet.java

The following code is of servlet that is acting as a controller

```
package controller;
import vu.*;
import java.io.*;
import java.net.*;
import java.sql.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ControllerServlet extends HttpServlet {
// This method only calls
processRequest() protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
processRequest(request, response);
}
// This method only calls processRequest() protected void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
processRequest(request, response);
}
protected void processRequest(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException{
// retrieving value of action parameter
String userAction = request.getParameter("action");
// if request comes to move to addperson.jsp from hyperlink if (userAction.equals("addperson") ) {
response.sendRedirect("addperson.jsp");
// if request comes to move to searchperson.jsp from hyperlink
```

```
} else if (userAction.equals("searchperson")) {
response.sendRedirect("searchperson.jsp");
//          if “save” button clicked on addperson.jsp to add new record
}          if (userAction.equals("save")) {
// this method defined below
addPerson(request,response);
// if “search” button clicked on searchperson.jsp for search
} else if (userAction.equals("search"))
{
// this method defined below
searchPerson(request,response);
}
} // end processRequest()
// if request comes to add/save person
private void addPerson(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException{
try
{
// creating PersonDAO object
PersonDAO pDAO = new PersonDAO();
// creating PersonInfo object
PersonInfo person = new PersonInfo();
// setting properties of Person object
// setting name property
String pName = request.getParameter("name");
person.setName(pName);
// setting address propertyt
String add = request.getParameter("address");
person.setAddress(add);
// setting phoneNumb property
```

```
String pNo = request.getParameter("phoneNum");
int phoneNum = Integer.parseInt(pNo);
person.setPhoneNum(phoneNum);
// calling PersonDAO method to save data into database
pDAO.addPerson(person);
// redirecting page to saveperson.jsp
response.sendRedirect("saveperson.jsp");
}catch (SQLException sqlex){
// setting SQLException instance request.setAttribute("javax.servlet.jsp.JspException" , sqlex);
RequestDispatcher rd = request.getRequestDispatcher("addbookerror.jsp");
rd.forward(request, response);
}catch (ClassNotFoundException cnfe){
// setting ClassNotFoundException instance
 request.setAttribute("javax.servlet.jsp.JspException" , cnfe);
RequestDispatcher rd = request.getRequestDispatcher("addbookerror.jsp");
 rd.forward(request, response);
}
}// end addperson()
// if request comes to search person record from database
private void searchPerson(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException{
try {
// creating PersonDAO object
PersonDAO pDAO = new PersonDAO();
String pName = request.getParameter("name");
// calling DAO method to retrieve personlist from database
// against name
ArrayList personList = pDAO.retrievePersonList(pName);
request.setAttribute("list", personList);
// forwarding request to showpeson, so it can render personlist
RequestDispatcher rd = request.getRequestDispatcher("showperson.jsp");
```

```
rd.forward(request, response);
}catch (SQLException sqlex){
// setting SQLException instance
request.setAttribute("javax.servlet.jsp.JspException" , sqlex);
RequestDispatcher rd = request.getRequestDispatcher("addbookerror.jsp");
rd.forward(request, response);
}catch (ClassNotFoundException cnfe){
// setting ClassNotFoundException instance
request.setAttribute("javax.servlet.jsp.JspException" , cnfe);
RequestDispatcher rd = request.getRequestDispatcher("addbookerror.jsp");
rd.forward(request, response);
}
}// end searchPerson()
} // end ControllerServlet
```

web.xml

As you already familiar, for accessing a servlet, you need to define a URL pattern in web.xml. This is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
<servlet>
<servlet-name> ControllerServlet </servlet-name>
<servlet-class> controller.ControllerServlet </servlet-class>
</servlet>
<servlet-mapping>
<servlet-name> ControllerServlet </servlet-name>
<url-pattern> /controller </url-pattern>
</servlet-mapping>
</web-app>
```

40.4 References:

- Java A Lab Course by Umair Javed.
- Java E-commerce course at Stanford

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 41: Layers and Tiers

How do you structure an application to support such operational requirements as maintainability, reusability, scalability and robustness? The answer lies in using Layers and Tiers? What different technologies Java provides to support layered or tiered architectures. The answer to these questions will remain our focus in this handout. A small case study will also be used to comprehend the concept of layers.

41.1 Layers vs. Tiers

Layers are merely logical grouping of the software components that make up the application or service, whereas Tiers refer to the physical residence of those layers.

In general,

Layers – represents the *logical view* of application

Tiers – represents *physical view* of application

However, both terms are used intractably very often. You must be confused what does logical & physical view mean? Let's elaborate layers and tiers further in detail to differentiate between them.

41.1.1 Layers

The partitioning of a system into layers such that each layer performs a specific type of functionality and communicates with the layer that adjoins it.

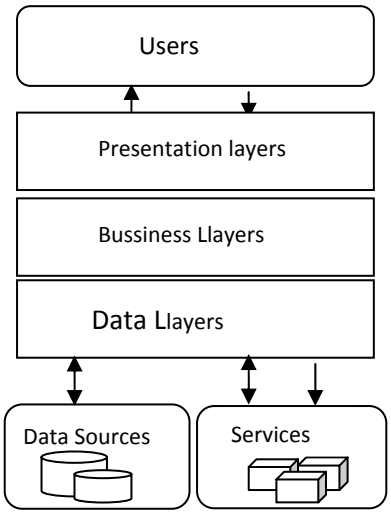
The separation of concerns minimizes the impact of adding services/features to an application. The application developed in layers also enables tiered distribution(discussed later). Furthermore easier maintenance, reuse of code, high cohesion & loose coupling sort of additional benefits are also enjoyed by the use of tiered architecture.

To begin with, layered architecture based on three layers. These are

- Presentation Layer
- Business Layer
- Data Layer

Note: However, there is no upper limit of number of layers an application can have. Each layer can also be further break down into several layers depending upon the requirements and size of the application.

The figure given below shows a simplified view of an application and its layers.



As you can see in the figure, users can only interact with the presentation layer. The presentation layer passes the user request to the business layer, which further passes the request to the data layer. The data layer communicates with the data sources (like Database etc.) or other external services in order to accomplish the user request.

Let's discuss each layer's responsibility in detail:

41.1.1.1 Presentation Layer

It provides a user interface to the client/user to interact with the application. This is the only part of the application visible to client.

Its job list includes collecting user's input, validating user's input (on client side using JavaScript like technologies OR on server side), presenting the results of the request made by the user and controlling the screen flow (which page/view will be visible to the user).

41.1.1.2 Business Layer

Also called *application layer*, it is only concerned with the application specific functionality. It is used to implement business rules and to perform business tasks.

For example, in a banking system, this layer will provide the functionality of banking functions such as opening an account, transferring of balance from one account to another, calculation of taxes etc.

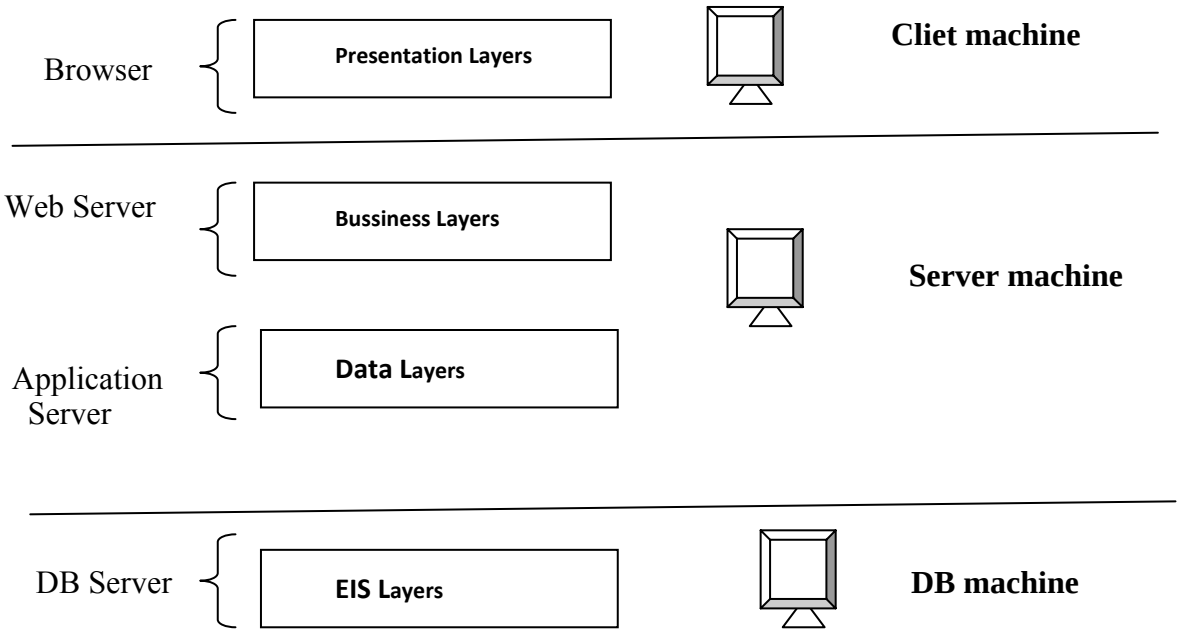
41.1.1.3 Data Layer

It is concerned with the management of the data & data sources of the system. Data sources can be database, XML, web services, flat file etc. Encapsulates data retrieval & storage logic For example, the address book application needs to retrieve all person records from a database to display them to the user.

41.1.2Tiers

As mentioned, layers help in building a tiered architecture. Like layers, there is no restriction on using number of tiers. An application can be based on *Single-tier*, *Two-tier*,*Three-tier* or *N-Tier* (application which have more than three tiers). The choice of using a tiered architecture is contingent to the business requirements and the size of the application etc.

Tiers are physically separated from each other. Layers are spread across tiers to build up an application. Two or more layers can reside on one tier. The following figure presents a three-tier architectural view of an application.



The client tier represents the client machine where actually web browser is running and usually displays HTML. You can think of a Presentation as of two parts; one is on client side, for example, HTML. There is also a presentation layer that is used to generate the client presentation often called *server presentation*. We'll discuss about it later.

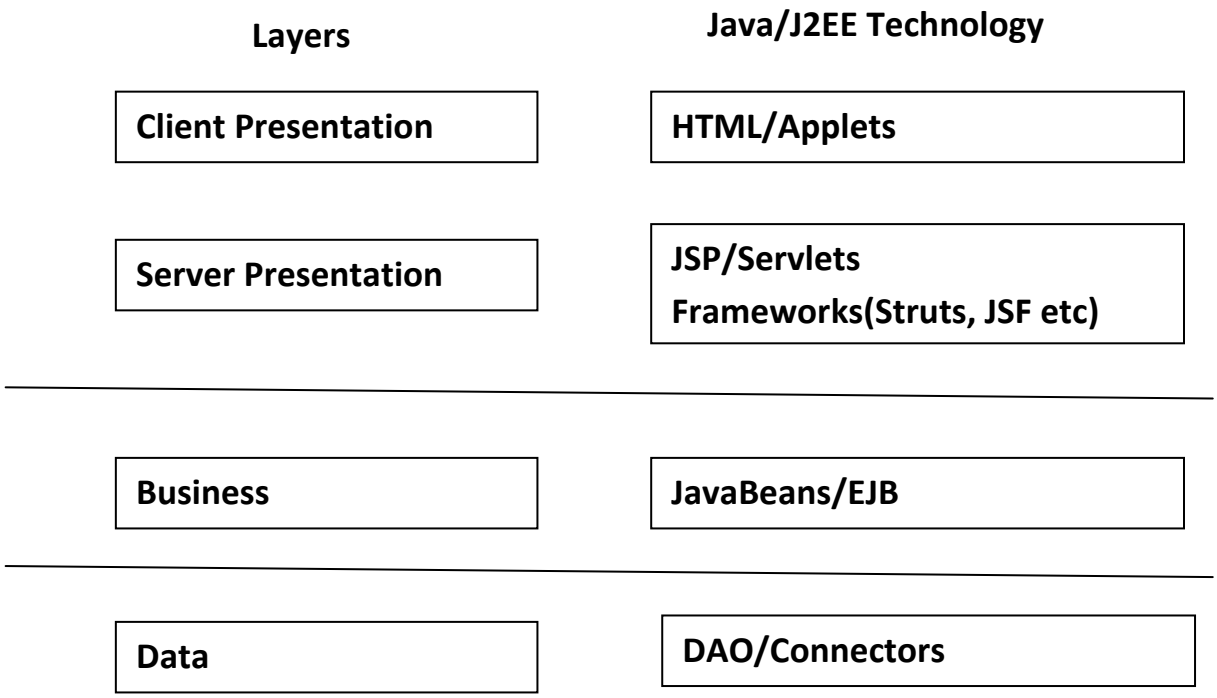
The server machine can consist on a single server machine or more. Therefore, it is possible *web server* is running on one server machine while *application server* on another. Web server is used to execute web pages like JSPs whereas application server is used to run special business objects like *Enterprise JavaBeans* (discussed later). The *web layer* and *applications server* can be on two separate machines or they can be on same tier as shown in the diagram.

The database server is often running on a separate tier, i.e. DB machine often called Enterprise information tier.

41.2 Layers Support in Java

The secret of wide spread use of Java lies in providing specific technology for each layer.This not only eases the development by freeing the programmer for caring operational features but only reduces the production time of the software.

In the following figure, Presentation is bifurcated into two layers. These are *Client Presentation layer* and *Server Presentation Layer*. What client sees in a browser forms *client presentation* layer while *server presentation* layer includes the Java technology components (JSP and Servlets etc.) that are used to generate the client presentation.



On business layer, JavaBeans (also referred as Plain Old Java Objects (POJO)) can be used. While moving towards a bigger architecture, the J2EE provides the special class that fits in business layer i.e. Enterprise JavaBean (EJB).

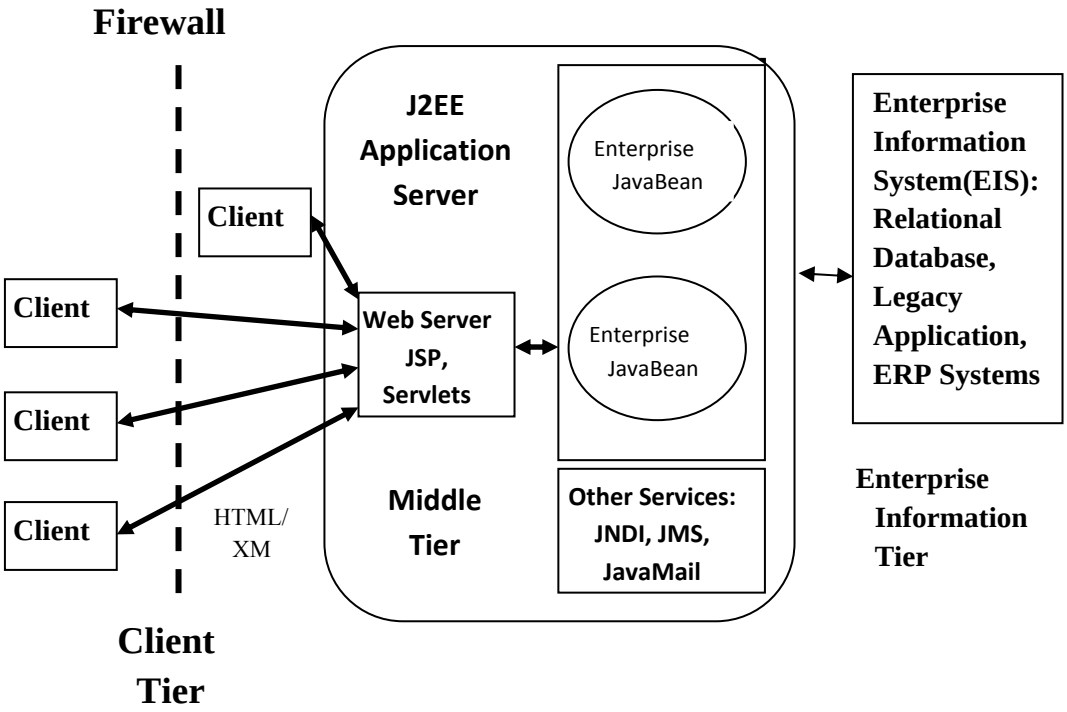
EJBs are special java classes that are used to encapsulate business logic. They provide additional benefits in building up an application such as scalability, robustness,scalability etc.

On data layer, Data Access Objects (DAO) can be used. Similarly you can use *connectors*. There are other different specialized components provided in java that ease the development of data layer.

41.3 J2EE Multi-Tiered Applications

In a typical J2EE Multi-Tiered application, a client can either be a swing based application or a web based. As you can see in the following figure, clients can access the web server from behind the firewall as well.

Suppose, our client is HTML based. Client does some processing on HTML and transports it to web server. JSP and Servlets are possible technologies that can be used in a web server. However, there are some Frameworks such as JSF etc that can be used in a web server. The classes which form the presentation layer reside on web server and of course controllers are also used over here.



If web server, wants to perform some business process, it usually gets help from some business layer components. The business layer component can be a simple JavaBean (POJO) but in a typical J2EE architecture, EJBs are used. Enterprise JavaBeans interacts with the database or information system to store and retrieve data.

EJBs and JSP/Servlets works in two different servers. As you already know, JSP and Servlets runs in a *web server* where as EJBs requires an *application server*. But, generally application server contains the web server as well.

Application server including web server generally resides on a single tier (machine),which is often called middle tier. This tier stores and retrieves data from the *Enterprise Information Tier (EIS)* which is a separate tier. The response sends back to the client by the middle tier can be HTML, XML etc. This response can be seen on the separate tier know as client tier.

41.4 Case Study: Matrix Multiplication using Layers

Problem Statement

Calculate product of two matrices of order 2 * 2
Result of multiplication should be stored in DB as well as shown to the user.

Format

- **Input format**
  - input will be in 4,2,6,5 format separated by commas where 4,2 represents entries of the first row
- **Display format**
  - Displays the matrix as a square
- **Storage format for DB**
  - Matrix will be stored as a string in the database along with the order of the matrix

- The following figure shows the table design that will be used to store the results.

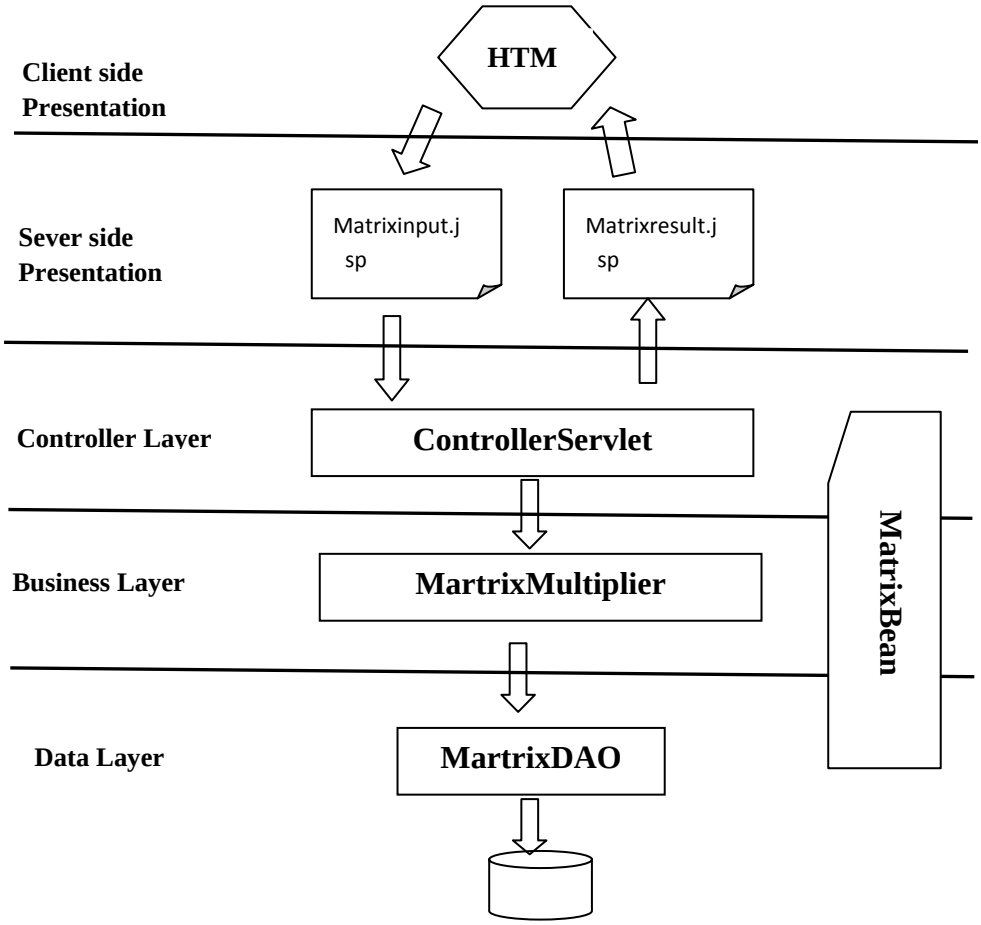
Matrix : Table

*	Field Name	Data Type
	id	AutoNumber
	mOrder	Text
	mValues	Text

Layer by Layer View

A picture's worth than thousand words. Therefore, before jumping on to code, let's put a glance over layers that will be used in this small case study. The classes that will be used on each layer and what functionality each class will perform will also be discussed.

First, look on the following picture that will describe the whole story.



The data layer has a class MatrixDAOthat is used to save the matrix result into database. As mentioned in the problem statement, that resultant matrix should be saved in the database. So, MatrixDAOis used to accomplish that.

MatrixDAOcalled by the MatrixMultiplier, a business layer class. The functionality list of MatrixMultiplierincludes:

-Converting the user input string (e.g. 2,3,4,1) into a proper object i.e. a matrix data structure.

-Helps in calculating product of two matrices.

Controller layer's class ControllerServletcalls the MatrixMultiplier. This layer calls the various business methods (like multiplication of two matrices) of business layer class and got the resultant matrix. Furthermore, ControllerServletsends the output to the matrixresult.jsp and receives the input from matrixinput.jsp.

The MatrixBean representing matrix data structure, as you can see in the figure is used across several layers. In fact, the object formed by MatrixMultiplier from a user input string is of MatrixBean type. It is used to transfer data from one layer to another.

First, look on the MatrixBean code given below:

MatrixBean

```
package bo;
import java.io.*;
public class MatrixBean implements Serializable{
// a 2D array representing matrix
public int matrix[ ][ ] ;
// constructor
public MatrixBean()
{
matrix = new int[2][2];
matrix[0][0] = 0;
matrix[0][1] = 0;
matrix[1][0] = 0;
matrix[1][1] = 0;
}
// setter that takes 4 int values and assigns these to array
public void setMatrix(int w, int x, int y, int z){
```

```
matrix[0][0] = w;
matrix[0][1] = x;
matrix[1][0] = y;
matrix[1][1] = z;
}
// getter returning a 2D array
public int[ ][ ] getMatrix(){
return matrix;
}
// used to convert 2D array into string
public String toString(){
return matrix[0][0] + "," + matrix[0][1] + "," + matrix[1][0] + "," +matrix[1][1] ;
}
} // end MatrixBean
```

matrixinput.jsp

This JSP is used to collect the input for two matrices in the form of string such as 2,3,5,8.The data will be submitted to ControllerServletfrom this page.

```
<html>
<body>
<h2>
Enter Two Matrices of order 2 * 2 to compute Product
</h2>
<h3>
<%--
“controller” is an alias/URL pattern of ControllerServlet
--%>
<form name="matrixInput" action="controller" > First Matrix:
<input type="text" name = "firstMatrix" /> E.g. 2,3,4,1
<br/>
Second Matrix:
```

```
<input type="text" name = "secondMatrix" />
<br/>
<input type = "submit" value = "Calculate Product" />
</form>
</h3>
</body>
</html>
```

ControllerServlet

This servlet acting as a controller receives the input from matrixinput.jsp. Furthermore,it will interact with the business layer class MatrixMultiplier to convert the string into a MatrixBean object, and to multiply two matrices.

```
package controller;
import bl.*;
import bo.* ;
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ControllerServlet extends HttpServlet {

// This method only calls processRequest()
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException{
processRequest(request, response);
}

// This method only calls processRequest()
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException{
processRequest(request, response);
}
```

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException{
// retrieving values from input fields of matrixinput.jsp
String sMatrix1 = request.getParameter("firstMatrix");
String sMatrix2 = request.getParameter("secondMatrix");
// Creating MatrixMultipler object
MatrixMultiplier mm = new MatrixMultiplier();
// Passing Strings to convertToObject() method of MatrixMultiplier
// convertToObject() is used to convert strings into MatrixBean
MatrixBean fMatrix = mm.convertToObject(sMatrix1);
MatrixBean sMatrix = mm.convertToObject(sMatrix2);

// passing MatrixBean's objects to multiply() method of
// MatrixMultiplier and receiving the product matrix in the form
// of MatrixBean
MatrixBean rMatrix = mm.multiply(fMatrix, sMatrix);

// saving results in database
mm.saveResult(rMatrix);
// storing the product of matrices into request, so that it can be
// retrieved on matrixresult.jsp
request.setAttribute("product", rMatrix);
// forwarding request to matrixresult.jsp
RequestDispatcher rd = request.getRequestDispatcher("matrixresult.jsp");
rd.forward(request, response);
} // end processRequest()
} // end ControllerServlet
```

MatrixMultiplier

The business layer class that's primary job is to calculate product of tow matrices given in the

form of MatrixBean. This class also has a method convertToObject that takes a String and returns back a MatrixBean object. MatrixMultiplier will also interact with the data layer class MatrixDAO to store results in the database.

```
package bl;
import bo.*;
import dal.*;
public class MatrixMultiplier {
//constructor
public MatrixMultiplier( ) {
}
// used to convert a String (like 2,3,4,5) into a MatrixBean object
public MatrixBean convertToObject(String sMatrix){
//splitting received string into tokens by passing “,” as
//delimeter
String tokens[] = sMatrix.split(",");
//creating MatrixBean object
MatrixBean matrixBO = new MatrixBean();

// converting tokens into integers
int w = Integer.parseInt(tokens[0]);
int x = Integer.parseInt(tokens[1]);
 int y = Integer.parseInt(tokens[2]);
int z = Integer.parseInt(tokens[3]);

// setting values into MatrixBean object by calling setter

matrixBO.setMatrix(w , x , y, z);
return matrixBO;
} // end convertToObject()
// used to multiply two matrices , receives two MatrixBean objects
// and returns the product in the form of MatrixBean as well
```

```
public MatrixBean multiply(MatrixBean fMatrix , MatrixBean sMatrix)
{
// creating MatrixBean object where product of the matrices will
// be
// stored
MatrixBean resultMatrix = new MatrixBean();

// retrieving two dimensional arrays from MatrixBeans object to
// perform multipication
int matrixA[ ][ ] = fMatrix.getMatrix();
int matrixB[ ][ ] = sMatrix.getMatrix();
int matrixC[ ][ ] = resultMatrix.getMatrix();
// code to multiply two matrices
for (int i=0; i<2; i++) {
for (int j=0; j<2; j++) {
for (int k=0; k<2; k++) {
matrixC[i][j] += (matrixA[i][k] * matrixB[k][j]);
}
}
// storing the product from 2d array to MatrixBean object by
// calling setter
resultMatrix.setMatrix( matrixC[0][0], matrixC[0][1], matrixC[1][0], matrixC[1][1] );
return resultMatrix;
} // end multiply()
// save results (MatrixBean containg product of two matrices)
//into
// database using DAO
public void saveResult( MatrixBean resultMatrix )
{
MatrixDAo dao=null;
try{
```

```
dao = newMatrixDAO();
} catch(ClassNotFoundException e){
} catch(SQLException e){
} dao.saveMatrix(resultMatrix);
}
} // end MatrixMulitplier
```

MatrixDAO

As class name depicts, it is used to store product results into database. Let's look on the code to see how it is accomplished.

```
package dal;
import java.util.*;
import java.sql.*;
import bo.*;
public class MatrixDAO{
private Connection con;
// constructor
public MatrixDAO() throws ClassNotFoundException , SQLException{
establishConnection();
}
// method used to establish connection with db
private void establishConnection() throws ClassNotFoundException,SQLException{
// establishing conection
class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
String conUrl = "jdbc:odbc:MatrixDSN";
con = DriverManager.getConnection(conUrl);
}
// used to store MatrixBean into database after converting it to
// a String
public void saveMatrix(MatrixBean matrix){
```

```
try{
String sql = "INSERT INTO Matrix(mOrder, mValues) VALUES (?,?)";
PreparedStatement pStmt = con.prepareStatement(sql);
// converting MatrixBean into String by calling toString()
String sMatrix = matrix.toString();
// setting order of matrix
pStmt.setString( 1 , "2*2" );

// setting matrix values in the form of string
pStmt.setString( 2 , sMatrix );
pStmt.executeUpdate();
}catch(SQLException sqlex){
System.out.println(sqlex);
}
} // end saveMatrix
// overriding finalize method to release acquired resources
public void finalize( ) {
try{
if(con != null){
con.close();
}
}catch(SQLException sex){ System.out.println(sex);
}
} // end finalize
} // end MatrixDAO class
```

matrixresult.jsp

Used to display resultant product of two matrices. The code is given below:

```
<%-- importing “bo” package that contains MatrixBean --%>
<%@ page import="bo.*"%>
```

```
<html>
<body>
<h1>The resultant Matrix is </h1>
<%--
retrieving MatrixBean object from request, that was set on
ControllerServlet
--%>
<%
MatrixBean productMatrix = (MatrixBean)request.getAttribute("product");
// retrieving values in 2d array so that it can be displayed
int matrix[][] = productMatrix.getMatrix() ;
%>
<%-- displaying MatrixBean's object values --%>
<TABLE>
<TR>
<TD> <%= matrix[0][0] %> </TD>
<TD> <%= matrix[0][1] %> </TD>
</TR>
<TR>
<TD> <%= matrix[1][0] %> </TD>
<TD> <%= matrix[1][1] %> </TD>
</TR>
</TABLE>
</body>
</html>
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
<servlet>
<servlet-name> ControllerServlet </servlet-name>
```

```
<servlet-class> controller.ControllerServlet </servlet-class>
</servlet>
<servlet-mapping>
<servlet-name> ControllerServlet </servlet-name>
<url-pattern> /controller </url-pattern>
</servlet-mapping>
</web-app>
```

41.5 References:

- Java A Lab Course by Umair Javed.
- Java Passion by Sang Shin

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 42: Expression Language

Sun Microsystems introduced the Servlet API, in the later half of 1997, positioning it as a powerful alternative for CGI developers who were looking around for an elegant solution that was more efficient and portable than CGI (Common Gateway Interface)programming. However, it soon became clear that the Servlet API had its own drawbacks, with developers finding the solution difficult to implement, from the perspective of code maintainability and extensibility. It is in some ways, this drawback that prompted the community to explore a solution that would allow embedding Java Code in HTML - JavaServer Pages (JSP) emerged as a result of this exploration.

Java as the scripting language in JSP scares many people particularly web page designers which have enough knowledge to work with HTML and some scripting language, faced lot of difficulties in writing some simple lines of java code. Can we simplify this problem to ease the life of web designer? Yes, by using *Expression Language (EL).*

JavaServer Pages Standard Tag Library (JSTL) 1.0 introduced the concept of the EL but it was constrained to only the JSTL tags. With JSP 2.0 you can use the EL with template text.

Note: - JSTL will be discussed in the following Handout.

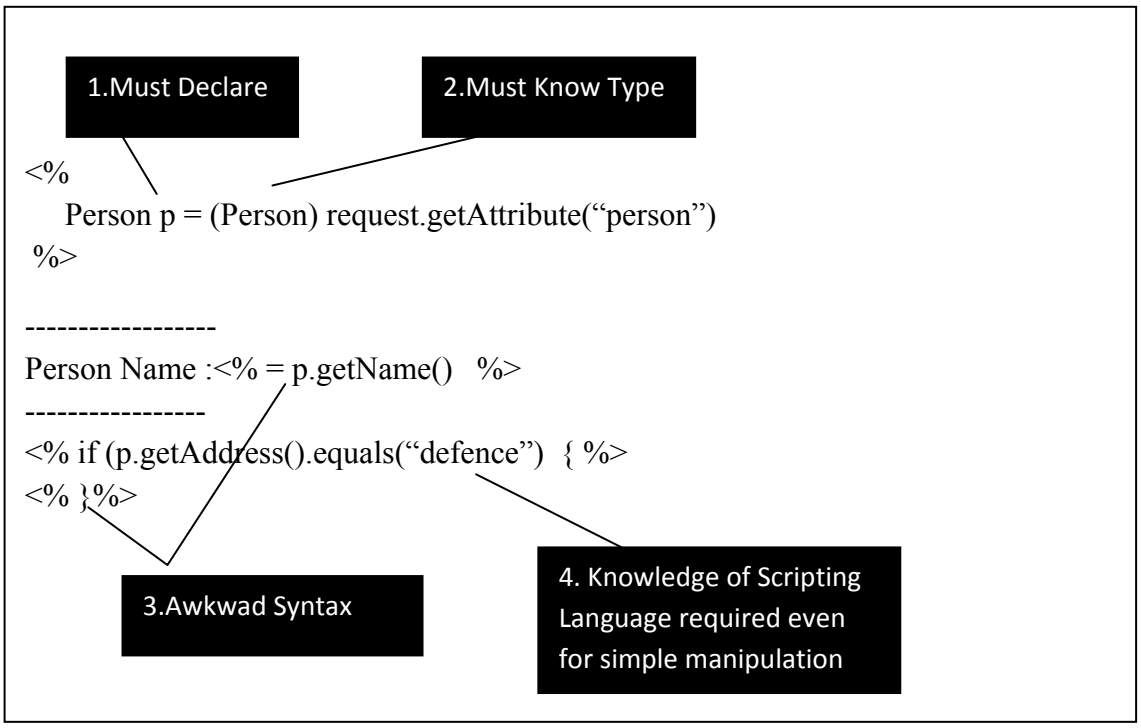
42.1 Overview

The Expression Language, not a programming or scripting language, provides a way to simplify expressions in JSP. It is a simple language that is geared towards looking up objects, their properties and performing simple operations on them. It is inspired form both the ECMAScript and the XPath expression language.

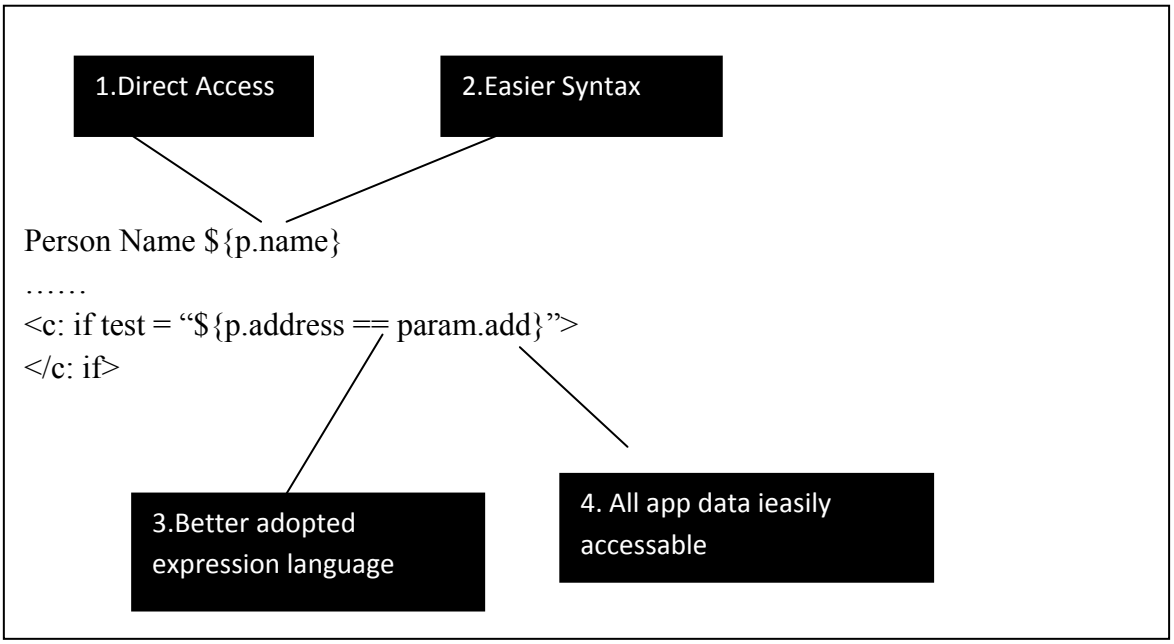
42.2 JSP Before and After EL

To add in motivational factor so that you start learning EL with renewed zeal and zest, a comparison is given below that illustrates how EL affects the JSPs.

The following figure depicts the situation of a JSP before EL. We have to declare a variable before using it, data type must be known in advance and most importantly have to use awkward syntax and many more. All these problems are highlighted in the following figure:



Contrary to the above figure, have a look on the subsequent figure that gives you a hint how useful EL can be?



42.3 Expression Language Nuggets

We'll discuss the following important pieces of EL. These are:

- Syntax of EL
- Expressions & identifiers
- Arithmetic, logical & relational operators
- Automatic type conversion
- Access to beans, arrays, lists & maps
- Access to set of implicit objects

42.3.1EL Syntax

The format of writing any EL expression is:

$ { validExpression }

The valid expressions can consist on these individuals or combination of these given below:

- Literals
- Operators
- Variables (object references)
- Implicit call to function using property name

- **EL Literals**

The list of literals that can be used as an EL expression and their possible values are given in the tabular format below:

Literals	Literal Values
Boolean	true or false
Integer	Similar to Java e.g. 243, -9642
Floating Point	Similar to Java e.g. 54.67, 1.83
String	Any string delimited by single or double quote e.g. "hello" , 'hello'
Null	Null

Examples of using EL literals are:

${ false } <%-- evaluates to false --%>

${ 8*3 } <%-- evaluates to 24 --%>

- **EL Operators**

The lists of operators that can be used in EL expression are given below:

Type	Operator
Arithmetic	+ - * / (div) % (mod)
Grouping	()
Logical	&&(and) ||(or) !(not)
Relational	== (eq) != (ne) < (lt) > (gt) <= (le) >= (ge)
Empty	The empty operator is a prefix operation used to determine if a value is null or empty. It returns a Boolean value.
Conditional	?:

Let us look at some examples that use operators as valid expression:

- ${ (6*5) + 5 } <%-- evaluate to 35 --%>
- ${ (x >= min) && (x <= max) }
- ${ empty name }
    - Returns *true* if name is
        - Empty string (“”),
        - Null etc.
- **EL Identifiers**

Identifiers in the expression language represent the names of objects stored in one of the JSP scopes: *page*, *request*, *session*, or *application*. These types of objects are referred to *scoped variables* throughout this handout.

EL has 11 reserved identifiers, corresponding to 11 *implicit objects*. All other identifiers assumed to refer to scoped variables.

- **EL implicit Objects**

The Expression Language defines a set of implicit objects given below in tabular format:

Category	Implicit Object	Operator
JSP	pageContext	The context for the JSP page, used to access the JSP implicit objects such as request, response, session,
Scopes	pageScope	A *Map* associating names & values of *page* scoped attributes
	requestScope	A *Map* associating names & values of *request* scoped attributes
	sessionScope	A *Map* associating names & values of *session* scoped attributes
	applicationScope	A *Map* associating names & values of *application* scoped attributes
Request Parameters	param	Maps a request parameter name to a single *String* parameter value.
	paramValues	Maps a request parameter name to an array of values
Request Headers	header	Maps a request header name to a single header value.
	headerValues	Maps a request header name to an array of value.
Cookies	cookie	A *Map* storing the *cookies* accompanying the request by
Initialization Parameters	initParam	A *Map* storing the *context initialization parameters* of the web application by name

Examples of using implicit objects are:

- ${ pageContext.response }
  - Evaluates to responseimplicit object of JSP
- ${ param.name }
  - This expression is equivalent to calling request.getParameter("name");
- ${ cookie.name.value }
  - Returns the value of the first cookie with the given name
  - Equivalent to

```
if (cookie.getName().equals("name"){
 String val = cookie.getValue();
   }
```

Example Code: Summation of Two Numbers using EL

This simple example demonstrates you the capabilities of EL. index.jspis used to collect input for two numbers and their sum is displayed on result.jspusing EL.
Let's first see the code of index.jsp

index.jsp

```
<html>
<body>
Enter two numbers to see their sum
<form action="result.jsp" >
First Number :
<input type="text" name="num1" />
<br>
Second Number:
<input type="text" name="num2" />

<input type="submit" value="Calculate Sum" />

</form>
</body>
</html>
```

result.jsp

```
<html>
<body>
<%--  The code to sum two numbers if we used scriptlet
<%
String no1 = request .getParameter("num1"); String no2 = request .getParameter("num2");

int num1 = Integer.parseInt(no1);
int num2 = Integer.parseInt(no2);
```

```
%>
Result is: <%= num1 + num2 %>
--%>


<%-- implicit Object param is used to access request parameters
By Using EL summing two numbers
--%>
Result is: ${param.num1 + param.num2}
</body>
</html>
```

42.3.2 EL Identifiers (cont.)

We had started our discussion on EL identifiers. Let's find out how these identifiers (variables) can be stored/retrieved in/from different scopes.

- **Storing Scoped Variables**

By using java code, either in pure servlet or in a scriptlet of JSP, we can store variables in a particular scope. For example,

- **Storing a variable in session scope using Java code**

Assume that we have PersonInfo class and we want to store its object p in *session* scope then we can write the following lines of code to accomplish that:

```
HttpSession ses = request.getSession(true); PersonInfo p = new
PersonInfo(); p.setName("ali");
ses.setAttribute("person" , p);
```

- **Storing a variable in request scope using Java code**

For the following lines of code, assume that request is of HttpServletRequest type. To store PersonInfo object p in request scope, we'll write:

```
PersonInfo p = new PersonInfo(); p.setName("ali");
request.setAttribute("person" , p);
```

You must be thinking of some another method (with which you are already familiar) to store a variable in a scope, certainly by using JSP action tags, we learned how to store a variable in any particular scope.

- **Storing a variable in request scope using JSP action tag**

  If we want to store p of type PersonInfo in request scope by using JSP action tags, then we'll write:

  ```
  <jsp:useBean id="p" class="PersonInfo"
  scope="request"/>
  ```

Later, you can change the properties of object p by using action tag as well.For example

```
<jsp:setProperty name="p" property="name" value="ali"
/>
```

- **Retrieving Scoped variables**

You are already very much familiar of retrieving any stored scoped variable by using java code and JSP action tags. Here, we'll discuss how EL retrieves scoped variables. As already mentioned, identifiers in the valid expression represent the names of objects stored in one of the JSP scopes: *page*, *request*, *session* and *application*.

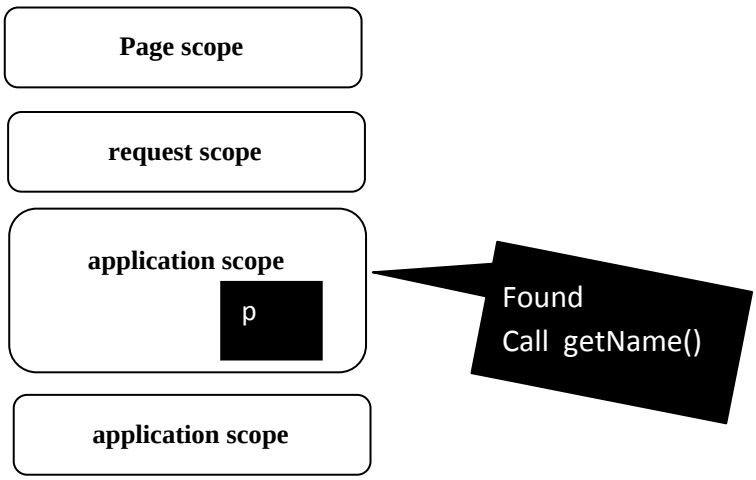
When the expression language encounters an identifier, it searches for a scoped variable with that name first in *page* scope,then in *request* scope,then in *session* scope,and finally in *application* scope

Note: - If no such object is located in four scopes, *null* is returned.

For example, if we've stored PersonInfo object p in *session* scope by mean of any mechanism discussed previously and have written the following EL expression to access the *name* property of p

${p.name}

Then EL searches for *p* first in page scope, then in request scope, then in session scope where it found p. After that it calls p.getName()method. This is also shown in pictorial form below:



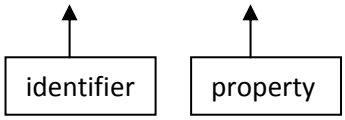
42.3.3 EL Accessors

The dot (**.**) and bracket (**[ ]**) operator let you access identifies and their properties. The dot operator typically used for accessing the *properties of an object* and the bracket operator is generally used to retrieve *elements of arrays and collections*.

- **Dot (.) operator**

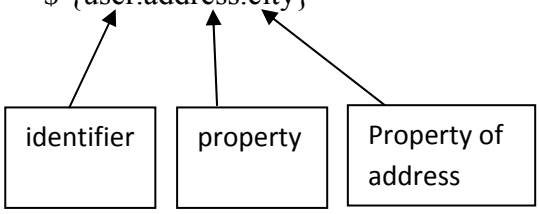
Assume that JavaBean PersonInfohas *name* property and its object personis stored in some scope. Then to access the *name* property of personobject, we'll write the following expression using EL:

$ {person.name}



The EL accesses the object's properties using the *JavaBeans conventions* therefore getName() must be defined in PersonInfo. Moreover, if property being accessed itself an *object*, the dot operator can be applied recursively. For example

$ {user.address.city}



- **Bracket ([ ]) operator**

This operator can be applied to arrays & collections implementing *List* interface e.g. ArrayListetc.

- *Index* of the element appears inside brackets
- For example, ${ personList[2] }returns the *3rd element* stored in it Moreover, this operator can also be applied to collections implementing *Map* interface e.g. HashMap etc.
- *Key* is specified inside brackets
- For example, ${ myMap["id"] }returns the value associated with the

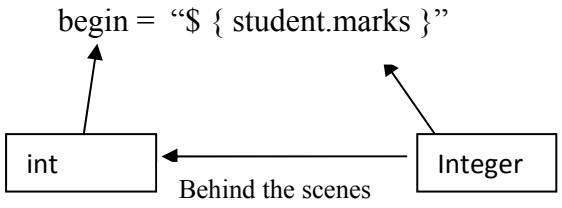
42.3.4EL – Robust Features

Some powerful characteristics of Expression Language are:

- Multiple expressions can be combined and intermixed with static text. For example

  $ { “Hello” ${user.firstName} ${user.lastName} }

- EL also supports automatic type conversion; as a result primitive can implicitly wrap and unwrap into/from their corresponding java classes. For example



- Most importantly, if object/identifier is *null*, no NullPointerException would be thrown. For example. If the expression written is:

  ${person.name}

Assume that personis *null*, then no exception would be thrown and the result would also be null.

42.3.5Using Expression Language

Expression Language can be used in following situations

- As attribute values in standard & custom actions. E.g.

  <jsp:setProperty id = “person” value = **${….}** />

- In template text – the value of the expression is inserted into the current output. E.g.

  <h3> **$ { …… }** </h3>

- With JSTL (discussed in the next handout)

Example Code: AddressBook using EL

So far, we have shown you implementation of *AddressBook* example in number of different ways. This time EL will be incorporated in this example. AddressBook code example consists on searchperson.jsp, showperson.jsp,ControllerServlet, PersonInfoand PersonDAOclasses. Let's look on the code of each of these components:

PersonInfo.java

The JavaBean used to represent one person record.

```
package vu;
import java.io.*;
public class PersonInfo implements Serializable{
private String name;
private String address;
private int phoneNum;
// no argument constructor
public PersonInfo() {
name = "";
address = "";
phoneNum = 0;
}
// setters
public void setName(String n){
name = n;
}
public void setAddress(String a){
address = a;
}
public void setPhoneNum(int pNo){
phoneNum = pNo;
}
```

```
// getters
public String getName( ){
return name;
}
public String getAddress( ){
return address;
}
public int getPhoneNum( ){
return phoneNum;
}
}
```

PersonDAO.java

It is used to retrieve/search person records from database.

```
package vu;
import java.util.*;
import java.sql.*;
public class PersonDAO{
private Connection con;
// constructor
public PersonDAO() throws ClassNotFoundException , SQLException {
establishConnection();
}//used to establish connection with database
private void establishConnection()
throws ClassNotFoundException , SQLException{
// establishing connection
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
String conUrl = "jdbc:odbc:PersonDSN";
con = DriverManager.getConnection(conUrl);
}
```

```
// used to search person records against name
 public ArrayList retrievePersonList(String pName) throws SQLException{
ArrayList personList = new ArrayList();
String sql = " SELECT * FROM Person WHERE name = ?";
PreparedStatement pStmt = con.prepareStatement(sql);
pStmt.setString( 1, pName);
System.out.println("retrieve person list");
ResultSet rs = pStmt.executeQuery();
String name;
String add;
int pNo;
while ( rs.next() ) {
name = rs.getString("name");
add = rs.getString("address");
pNo = rs.getInt("phoneNumber");
// creating a PersonInfo object
PersonInfo personBean = new PersonInfo();
personBean.setName(name);
personBean.setAddress(add);
personBean.setPhoneNum(pNo);
// adding a bean to arraylist
personList.add(personBean);
} // end while
return personList;
} // end retrievePersonList
//overriding finalize method to release resources
public void finalize( ) {
try{
if(con != null){
con.close();
```

```
}
}catch(SQLException sex){
System.out.println(sex);
}
} // end finalize
} // end class
```

searchperson.jsp

This JSP is used to gather person's name from the user and submits this data to the ControllerServlet.

```
<html>
<body>
<center>
<h2> Address Book </h2>
<h3> Search Person</h3>
<FORM name ="search" action="controllerservlet" />
<TABLE BORDER="1" >
<TR>
<TD> <h4 >Name</h4> </TD>
<TD> <input type="text" name="name" />     </TD>
</TR>
<html>
<body>
<center>
<h2> Address Book </h2>
<h3> Search Person</h3>
<FORM name ="search" action="controllerservlet" />
<TABLE BORDER="1" >
<TR>
<TD> <h4 >Name</h4> </TD>
<TD> <input type="text" name="name" />     </TD>
</TR>
```

ControllerServlet.java

The Controller Servlet receives request from *searchperson.jsp* and after fetching search results from database, forwards the request to *showperson.jsp*.

package controller;

```
import vu.*;
import java.util.*;
import java.io.*;
import java.net.*;
 import javax.servlet.*;
import javax.servlet.http.*;

public class ControllerServlet extends HttpServlet {
// This method only calls processRequest()
protected void doGet(HttpServletRequest request, HttpServletResponse response throws
ServletException, IOException e){
processRequest(request, response);
}
// This method only calls processRequest()
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
processRequest(request, response);
}
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
// defined below
searchPerson(request, response);
} // end processRequest()
protected void searchPerson(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
try {
```

```
// creating PersonDAO object
PersonDAO pDAO = new PersonDAO();
// retrieving request parameter “name” entered on showperson.jsp
String pName = request.getParameter("name");
// calling DAO method to retrieve personlist from database
// against the name entered by the user
ArrayList personList = pDAO.retrievePersonList(pName);
// storing personlist in request scope, later it is retrieved
// back on showperson.jsp
 request.setAttribute("plist", personList);

// forwarding request to showperson, so it renders personlist
RequestDispatcher rd = request.getRequestDispatcher("showperson.jsp");
rd.forward(request, response);
}catch (Exception ex) {
System.out.println("Exception is" + ex);
}
} // end searchPerson
} // end ControllerServlet
```

showperson.jsp

This page is used to display the search results. To do so, it reclaims the stored ArrayList (*personList*) from the *request* scope. Furthermore, this page also uses the Expression Language to display records.

```
<%-- importing required packages--%>
<%@page import="java.util.*" %>
<%@page import="vu.*" %>
<html>
<body>
<center>
```

```
<h2> Address Book </h2>
<h3> Following results meet your search criteria</h3>
<TABLE BORDER="1" >
<TR>
<TH>Name</TH>
<TH>Address</TH>
<TH>PhoneNum</TH>
</TR>
<%-- start of scriptlet --%>
<%
// retrieving ArrayList from request scope
ArrayList personList =(ArrayList)request.getAttribute("plist");
PersonInfo person = null;
for(int i=0; i<personList.size(); i++) {
person = (PersonInfo)personList.get(i);
// storing PersonInfo object in request scope
/* As mentioned, an object must be stored in
some scope to work with Expression Language*/
request.setAttribute("p", person);
%>
<%-- end of scriptlet --%>
<TR>
<%-- accessing properties of stored PersonInfo
object with name “p” using EL --%>
<TD> ${ p.name } </TD>
<TD> ${ p.address} </TD>
<TD> ${ p.phoneNum} </TD>
<%-- The following expressions are now replaced by EL statements written above--%>
<%-- <%= person.getName()%> --%>
<%-- <%= person.getAddress()%> --%>
```

```
<%-- <%= person.getPhoneNum()%> --%>
</TR>
<%
} // end for
%>
</TABLE >
</center>
</body>
</html>
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
<servlet>
<servlet-name> ControllerServlet </servlet-name>
<servlet-class> controller.ControllerServlet </servlet-class>
</servlet>
<servlet-mapping>
<servlet-name> ControllerServlet </servlet-name>
<url-pattern> /controllerservlet </url-pattern>
</servlet-mapping>
</web-app>
```

42.4 References:

- Java A Lab Course by Umair Javed.
- Expression Language Tutorial by Sun
  http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JSPIntro7.html
- The JSTL Expression Language by David M. Geary
  http://www.informit.com/articles/article.asp?p=30946&rl=1

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 43: JavaServer Pages Standard Tag Library (JSTL)

43.1 Introduction

The JSP Standard Tag Library (JSTL) is a collection of custom tag libraries that implement general-purpose functionality common to Web applications, including iteration and conditionalization, data management formatting, manipulation of XML, and database access. Like JSP, JSTL is also a specification not an implementation. The development theme of JSTL is "scriptlet free JSP".

These tag libraries provide a wide range of custom action functionality that most JSP authors have found themselves in need of in the past. Having a defined specification for how the functionality is implemented means that a page author can learn these custom actions once and then use and reuse them on all future products on all application containers that support the specification. Using the JSTL will not only make your JSPs more readable and maintainable, but will allow you to concentrate on good design and implementation practices in your pages.

43.2 JSTL & EL

JSTL includes supports for Expression Language thus EL can be used to specify dynamic attribute values for JSTL actions without using full-blown programming language. Prior to JSP 2.0, EL can only be used in attributes of JSTL tags but EL now becomes a standard part of JSP 2.0. This allows the use of EL anywhere in the document.

43.3 Functional Overview

As mentioned, JSTL encapsulates common functionality that a typical JSP author would encounter. This set of common functionality has come about through the input of the various members of the expert group. Since this expert group has a good cross-section of JSP authors and users, the actions provided in the JSTL should suit a wide audience.While the JSTL is commonly referred to as a single tag library, it is actually composed of four separate tag libraries:

Core−>contains tags for conditions, control flow and to access variables etc.

XML manipulation−>contains tags for XML parsing and processing

SQL−>contains tags for accessing and working with database.

Internationalization and formatting−>contains tags to support locale messages, text, numbers and date formation

43.4 Twin Tag Libraries

JSTL comes in two flavors to support various skill set personal

- **Expression Language (EL) version**
  - o Dynamic attribute values of JSTL tags are specified using JSTL expression language (i.e. **${ expression }**)
  - o The EL based JSTL tag libraries along with URIs and preferred prefixes are given below in tabular format

Library	URI	Prefix
Core	http://java.sun.com/jsp/jstl/core	c
SQL	http://java.sun.com/jsp/jstl/sql	sql
Internationalization/	http://java.sun.com/jsp/jstl/fmt	fmt
XML	http://java.sun.com/jsp/jstl/xml	x

- **Request Time (RT) version**
  - o Dynamic attribute values of JSTL tags are specified using JSP expression (i.e. **<%= expression %>**)
  - o The RT based JSTL tag libraries along with URIs and preferred prefixes are given below in tabular format

Library	URI	Prefix
Core	http://java.sun.com/jsp/jstl/core**_rt**	c**_rt**
SQL	http://java.sun.com/jsp/jstl/sql**_rt**	sql**_rt**
Internationalization/	http://java.sun.com/jsp/jstl/fmt**_rt**	fmt**_rt**
XML	http://java.sun.com/jsp/jstl/xml**_rt**	x**_rt**

43.5 Using JSTL

As we discussed earlier, JSTL includes four standard tag libraries. As is true with any JSP custom tag library, a taglibdirective must be included in any page that you want to be able to use this library's tags.

For example, to use *EL based core* tag library, the taglibdirective appears as:

<%@taglib prefix="c" uri=http://java.sun.com/jsp/jstl/core %>

And to use *RT based core* tag library, the taglibdirective appears as:
<%@taglib prefix="c**_rt**" uri=http://java.sun.com/jsp/jstl/core**_rt**
%>

43.6 Working with Core Actions (tags)

The set of tags that are available in the Core tag library come into play for probably most anything you will be doing in your JSPs such as:

- Manipulation of scoped variables
- Output
- Conditional logic
- loops
- URL manipulation
- and Handling errors.

Let's walk through some important core actions:

c:se
t

Provides a tag based mechanism for creating and setting scope based variables. Its syntax is as follows:

<c:set var="name" scope = "scope" value = "expression" />

Where the varattribute specifies the name of the scoped variable, the scopeattribute indicates which scope (page | request | session | application) the variable resides in, and the valueattribute specifies the value to be bound to the variable. If the specified variable already exists, it will simply be assigned the indicated value. If not, a new scoped variable is created and initialized to that value.

The scopeattribute is optional and default to page.

Three examples of using c:setare given below. In the first example, a pagescoped variable *"timezone"* is set to a value*"Asia / Karachi"*.
<c:set var="timezone" value="Asia/Karachi" **/>**

In the second example, a request scoped variable *"email"* email is set to a value
"me@gmail.co m"

<c:set var="email" scope="request" value="me@gmail.com" **/>**

In the third example, a page scoped variable *"email"* is set to value of request parameter*"email"*
by using paramimplicit object. If email parameter is defined in JSP page as:

<input type="text" value = "email" />

Then c:settag would be used as:

<c:set var="email" scope="request" value="param.email" **/>**

Using c:set with JavaBeans & Map

c:settag can also be used to change the property of a bean or the value against some key. For this purpose, the syntax of the c:settag would look like this:

<c:set target="bean/map" property="property/key" value="value" />

If *target* is a bean, sets the value of the property specified. This process is equivalent to <jsp:setProperty … />JSP action tag.

If *target* is a Map, sets the value of the key specified

And of course, these beans and maps must be stored in some scope prior to any attempt is made to change their properties.

For example, consider the following snippet of code that stores *PersonInfo's* object *person* into request scope using <jsp:useBean … />tag. Then using c:settag,
person's *name* property is set to *"ali"*.

<jsp:useBean id="person" class="vu.PersonInfo" scope="request" **/>**

<c:set target="person" property ="name" value = "ali" **/>**

c:out

A developer will often want to simply display the value of an expression, rather than store it.

This can be done by using c:outcore tag, the syntax of which appears below:

<c:out value = "expression" default = "expression" />

This tag evaluates the expression specified by its valueattribute, and then prints the result. If the optional defaultattribute is specified, the c:outaction will print its

(default) value if the valueattribute's expression evaluates either to *null* or an *empty String*.
This tag is equivalent to JSP expression i.e. <%=expression %>.

Consider the following examples in which the usage of c:outtag has shown. In the first example, string “Hello” would be displayed

<c:out value = “Hello” />

In the second example, if request parameter num evaluates to null or an empty string then default value “0” would be displayed.

<c:out value = “${param.num}” default = “0” />

The above fragment of code is equivalent to following scriptlet:

```
<%
String no = request.getParameter(“num”);

if (no == null || no.equals(“”)) {

System.out.println(0);
}else{ Out.println(no);
}
%>
```

If we want to display the property of a bean like *name*, we’ll write

<c:out value= “${person.name}” default = “Not Set” />

c:remove

As its name suggests, the c:removeaction is used to delete a scoped variable, and takes two attributes. The varattribute names the variable to be removed, and the optional scopeattribute indicates the scope from which it should be removed and defaults to page.

For example, to remove a variable named *square* from page scope, we’ll write:

<c:remove var = “square” />

And if variable *email* is required to be removed from requestscope, then c:removetag will look like:

<c:remove var = “email” scope = “request” />

c:forEach

In the context of Web applications, iteration is primarily used to fetch and display collections of data, typically in the form of a list or sequence of rows in a table. The primary JSTL action for implementing iterative content is the c:forEachcore tag. This tag supports two different styles of iteration:

Iteration over an integer range (like Java language's for statement)

Iteration over a collection (like Java language's Iterator and Enumeration classes).

Iteration over an Integer range

To iterate over a range of integers, the syntax of the c:forEachtag will look like:

<c:forEach var="name" begin="expression" end="expression"
step="expression" **>**
Body Content
</c:forEach>

The beginand endattributes should be either constant *integer values* or expressions evaluating to integer values. They specify the initial value of the index for the iteration and the index value at which iteration should cease, respectively. When iterating over a range of integers using c:forEach, these two attributes are required and all others are optional.

The stepattribute specifies the amount to be added to the index after each iteration. Thus the index of the iteration starts at the value of the beginattribute, is incremented by the value of the stepattribute, and halts iteration when it exceeds the value of the endattribute. Note that if the stepattribute is omitted, the step size defaults to 1.

If the varattribute is specified, then a scoped variable with the indicated name will be created and assigned the current value of the index for each pass through the iteration. This scoped variable has nested visibility that is it can only be accessed within the body of the c:forEach tag.

For example to generate squares corresponding to range of integer values, the c:forEachtag will be used as:

<c:forEach var="x" begin="0" end="10" step="2" **>**
<c:out value="${x * x}" />
</c:forEach>

By executing the above code, following output would appear:
4 16 36 64 100

Iteration over a Collection

When iterating over the members of a collection and arrays etc, one additional attribute of the c:forEachtag is used: the itemsattribute. Now the c:forEachtag will look similar to this:

<c:forEach var="name" **items="expression" >**
Body Content
</c:forEach>

When you use this form of the c:forEachtag, the itemsattribute is the only required attribute. The value of the itemsattribute should be the collection/array over whose members the iteration is to occur, and is typically specified using an EL expression. If a variable name is also specified using varattribute, then the named variable will be bound to successive elements of the collection for each iteration pass.

For example, to iterate over a String array (messages) using java code, we used to write in JSP:

```
<
%
for(int i=0; i<messages.length; i++) { String msg =
messages[i];
%
>

<%= msg %>

<
%
} // end for
%
>
```

This can be done using c:forEachtag in much simpler way as shown below:

<c:forEach var="msg" **items="${messages}" >**

<c:out value= "${msg}" />

</c:forEach>

Similarly, to iterate over a *persons* ArrayListthat contains *PersonInfo* objects, w used to write in JSP:

```
<%
ArrayList persons = (ArrayList)request.getAttribute("pList");
for(int i=0; i<persons.size(); i++) { PersonInfo p ==
(PersonInfo)persons.get(i); String name = p.getName();
%>
<%= name %>
<%
} // end for
%>
```

Indeed, the above task can be achieved in much simpler way using c:forEachtag as shown below:

<c:forEach var="p" **items="${persons}" >**

<c:out value= "${p.name}" />

</c:forEach>

The c:forEachtag processes each element of this list(*persons*) in turn, assigning it to a scoped variable named *p*. Note that typecast is also not required.

Furthermore, you can use the begin, end, and stepattributes to restrict which elements of the

c:if

Like ordinary Java's if, used to conditionally process the body content. It simply evaluates a single test expression and then processes its body content only if that expression evaluates to true. If not, the tag's body content is ignored. The syntax for writing c:iftag is:

<c:if test= "expression" **>**
Body Content
</c:if>

For example, to display a message "a equals b" if two strings a & b are equal, the c:if tag is

used as:

<c:if test= "${a == b}" **>**
<h2> A equals B </h2>

</c:if>

c:choose

c:choosethe second conditionalization tag, used in cases in which mutually exclusively test

are required to determine what content should be displayed. The syntax is shown below:

<c:choose>
<c:when test= "expression" **>**
Body content
</c:when>
……
……
<c:otherwise >
Body content
</c:otherwise>
</c:choose>

Each condition to be tested is represented by a corresponding <c:when>tag, of which there must be at least one. Only the body content of the first <c:when>tag whose test evaluates to *true* will be processed. If none of the <c:when>tests return true, then the body content of the <c:otherwise>tag will be processed.

Note, though, that the <c:otherwise>tag is optional; a <c:choose>tag can have at most one nested <c:otherwise>tag. If all <c:when>tests are false and no <c:otherwise> action is present, then no <c:choose>body content will be processed.

The example code given below illustrates the usage of c:choosetag in which two strings a & b are compared and appropriates messages are displayed:

<c:choose>

```
<c:when test= “a == b” >
<h2> a equals b</h2>
</c:when>

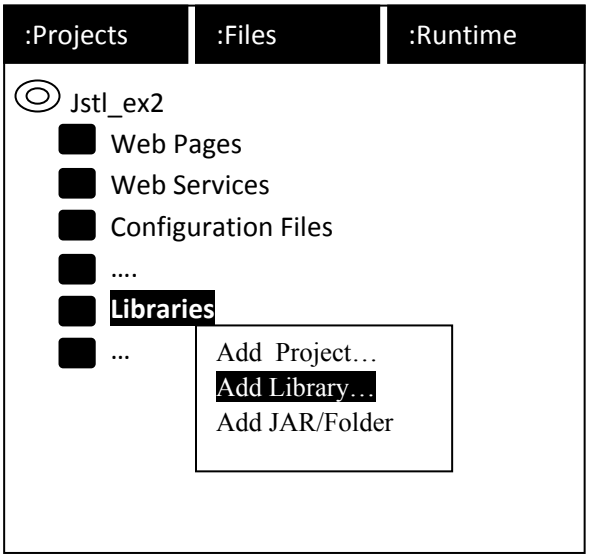
<c:when test= “a <= b” >
<h2> a is less than b</h2>
</c:when>

<c:otherwise >
<h2> Don’t know what a equals to </h2>
</c:otherwise>

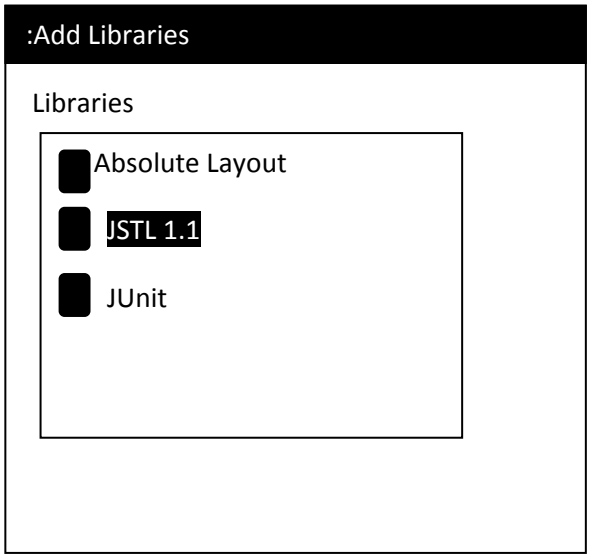
</c:choose>
```

43.7 NetBeans 4.1 and JSTL

If you are using *netBeans 4.1* IDE then you have to add JSTL library to your project manually. To do so, right click on the *libraries* folder, you can find it under project’s name and select the *Add Library* option. This is also shown in the following figure:



The Add Library dialog box opens in front of you. Select *JSTL 1.1* option and press *Add Library* button. Now you can refer to any JSTL library in your JSPs.



Note: Remember that the *JSTL 1.1* library is only added to current project. You have to repeat this step for each project in which you want to incorporate JSTL.

Example Code: AddressBook using JSTL core tags

This is the modified version of *AddressBook* that was built using *Expression Language* in the last handout. Only showperson.jspis modified to incorporate JSTL core tags along with Expression Language in place of scriptlets. The remaining participants searchperson.jsp, ControllerServlet, PersonInfoand PersonDAOleft unchanged. Let's look on the

code of each of these components:

PersonInfo.java

The JavaBean used to represent one person record.

```
package vu;
import java.io.*;
public class PersonInfo implements Serializable{
private String name; private String address; private int phoneNum;
// no argument constructor public PersonInfo() {
name = ""; address = ""; phoneNum = 0;
}
// setters
public void setName(String n){
```

```
name = n;
}
public void setAddress(String a){
address = a;
}
public void setPhoneNum(int pNo){
phoneNum = pNo;
}
// getters
public String getName( ){
return name;
}
public String getAddress( ){
return address;
}
public int getPhoneNum( ){
return phoneNum;
}}
```

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 44: Client Side Validation & JavaServer Faces (JSF)

In this handout, we'll talk about client side validation and also learn about growing in demand Java technology i.e. JSF. First start with client side validation

44.1 Client Side Validation

Forms validation on the client-side is essential -- it saves time and bandwidth, and gives you more options to point out to the user where they've gone wrong in filling out the form.
Furthermore, the browser doesn't have to make a round-trip to the server to perform routine client-side tasks. For example, you wouldn't want to send the browser to the server to validate that all of the required fields on a form were filled out.

Any scripting language can be used to achieve the said objective. However, *JavaScript* and *VBScript* are t wo popular options

44.1.1 Why is Client Side Validation Good?

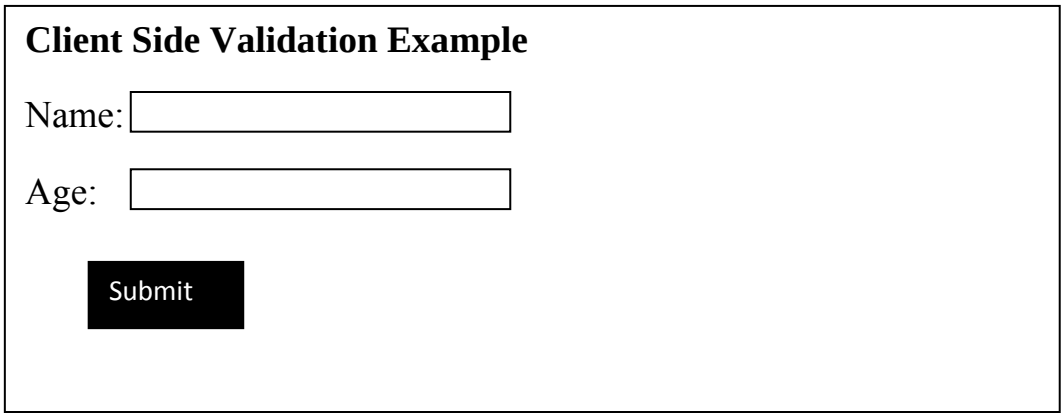
There are two good reasons to use client-side validation:

- It's a fast form of validation: if something's wrong, the alarm is triggered upon submission of the form.
- You can safely display only one error at a time and focus on the wrong field, to help ensure that the user correctly fills in all the details you need.

Code Example: Form Validation using JavaScript

For example on the following form, we want to make sure that text filed for name should not be left empty and age field does not contain any negative value. To accomplish this we'll use JavaScript. If user forgets to provide name and/or enters a negative value, a message would be displayed to the user that indicates what was went wrong? However, if user conforms to requirements, he/she would be taken to another page that displays a greeting message.

Note: In this example, JavaScript semantics isn't discussed over here as I am assuming that you might be familiar with some scripting language. Otherwise, www.w3schools.com is an excellent resource to learn about scripting languages



The code that is used to generate this page is given below:

```
<HTML>
<HEAD>
<!—- start of scripting code and mentioning type -->
<SCRIPT TYPE = "text/javascript">
/* defining a function that receives form's reference, defined inside the body and returns false if any
requirement violated
*/
function validateForm(thisform)
{
/* checking the value of the name field, if it is left empty
then displaying a message
*/
if (thisform.name.value == null || thisform.name.value == "")
{
alert("Username is required");
return false;
}
```

```
// if value of age is negative, displaying a message if (thisform.age.value < 0 )
{
alert("Age can't be negative");
return false;
} // end of function

</SCRIPT> <!—-end of script-- >
</HEAD>
<BODY>

<!—- validateForm method is called and specified as a value of onsubmit value, if this method returns
false, the user remains on the same page -->
<FORM method="post" onsubmit="return validateForm(this)" action = “greeting.jsp” >

<h2> Client Side Validation Example </h2>

<BR/>
Name: <INPUT type="text" name="name" size="30" />

<BR/> <BR/>
Age: <INPUT type="text" name="age" size="30" />
<BR/> <BR/>
<INPUT type="submit" value="Submit">

</FORM>

</BODY>
</HTML>
```

44.2 JavaServer Faces (JSF)

JSF technology simplifies building the user interface for web applications. It does this by providing a higher-level framework for working with your web applications. Some distinct features will be discussed provided by this technology. To begin with, have a look on some popular existing frameworks

44.2.1 Different existing frameworks

- **Struts**

A popular open source JSP-based Web application framework helps in defining a structured programming model (MVC), also validation framework and reduces tedious coding but…

  - Adds complexity and doesn't provide UI tags
  - Very Java programmer centric

- **Tapestry**

Another popular framework that is extensively used in the industry is Tapestry. It has almost similar sort of problems as with Struts.

44.2.2 JavaServer Faces

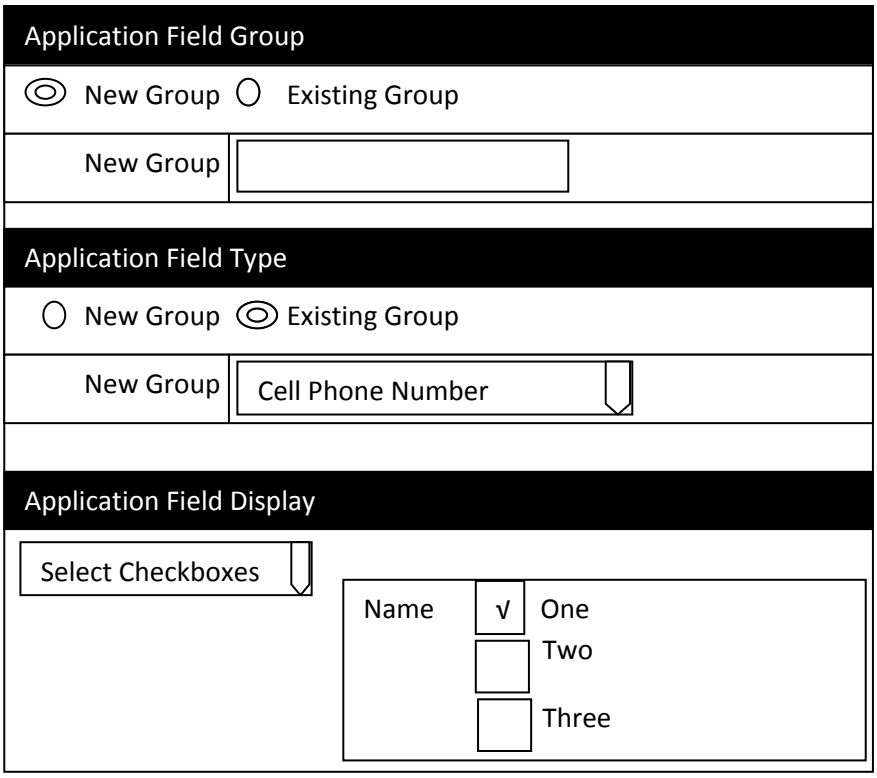
A framework which provides solutions for:

- Representing UI components
- Managing their state
- Handling events
- Input validation
- Data binding
- Automatic conversion
- Defining page navigation
- Supporting internationalization and accessibility.

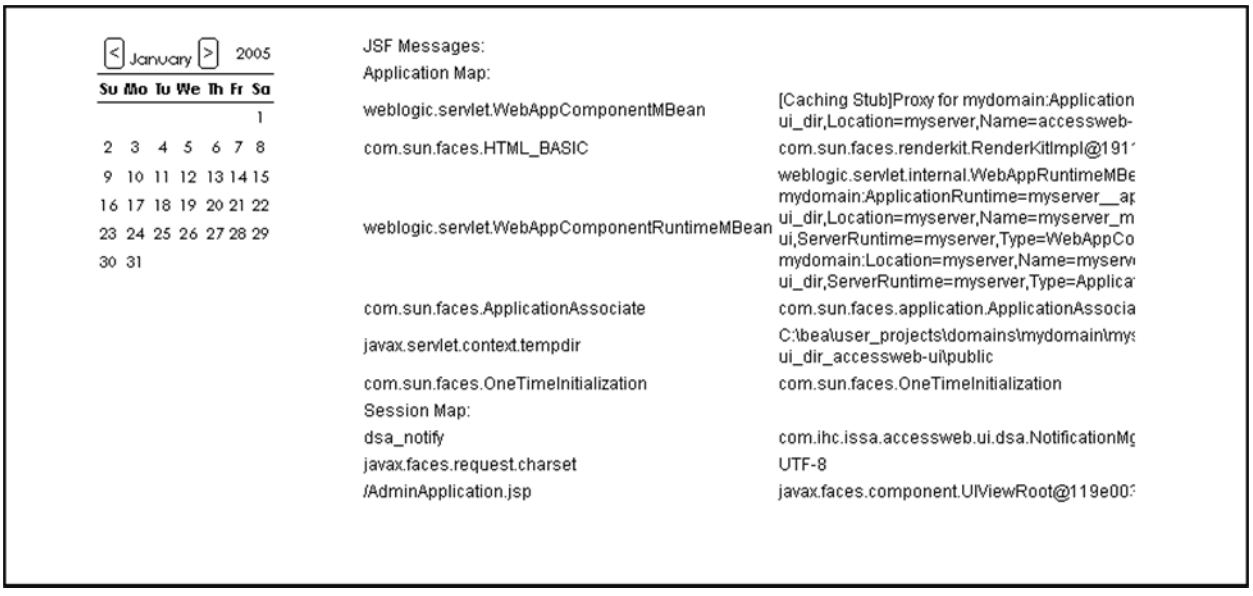
If you are familiar with Struts and Swing (the standard Java user interface framework for desktop applications), think of JavaServer Faces as a combination of those two frameworks. Like Swing, JSF provides a rich component model that eases event handling and component rendering; and like Struts, JSF provides Web application lifecycle management through a controller servlet.

44.2.3 JSF UI Components

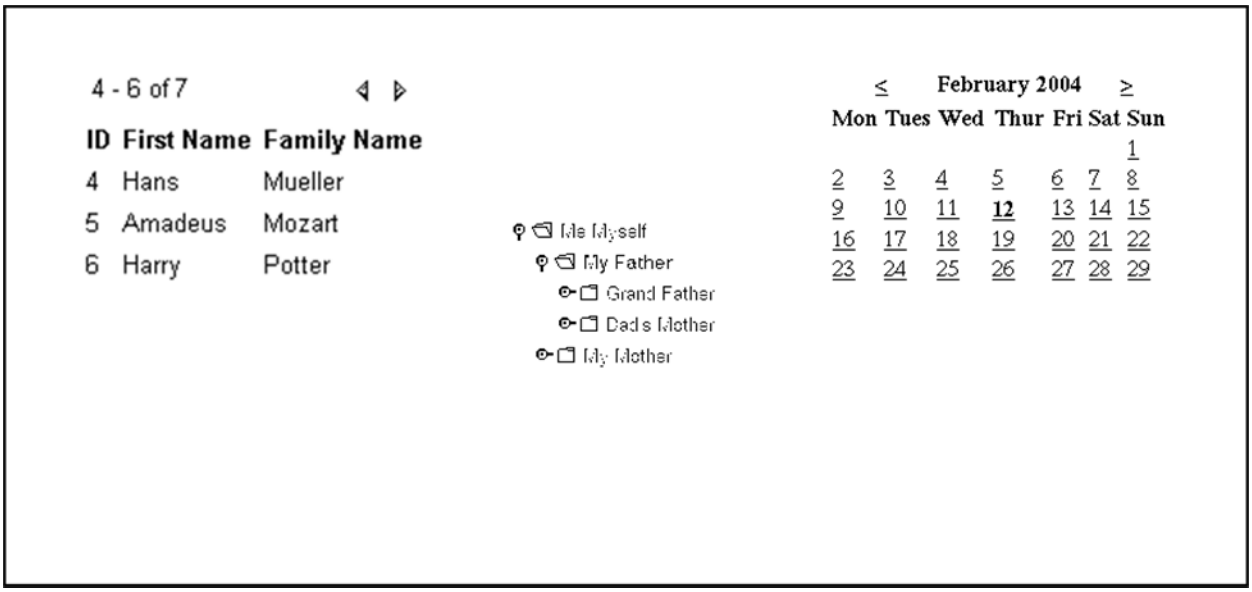
Some of the standard JavaServer Faces components are shown below:



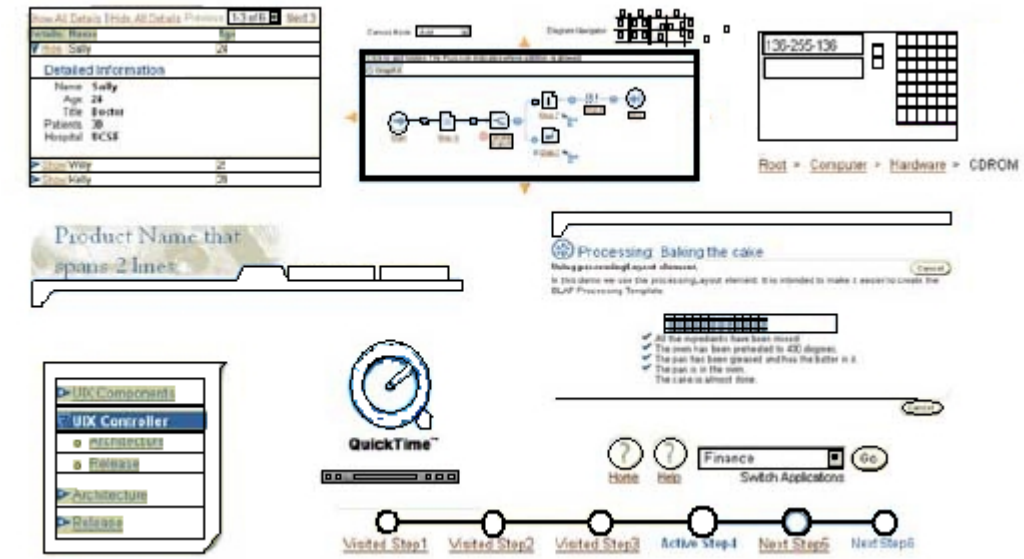
Some custom JavaServer Faces components are



And some open course JavaServer Faces components are also available like:



And some third-party Java Server Faces components are also available:



44.2.4 JSF Events Handling

A JSF application works by processing events triggered by the JSF components on the pages. These events are caused by user actions. For example, when the user clicks button, the button triggers an event. You, the JSF programmer, decide what the JSF application will do when a

particular event is fired. You do this by writing event listeners.In other words, a JSF application is event-driven.

For example, if you write a JSF code to create a button, you will write:

```
<h:commandButton value="Login“
actionListener=“#{customer.loginActionListener}”
action=“#{customer.login}” />
```

The value attribute specifies the text that appeared on the face of a button, the actionListenerattributes specifies to call the loginActionListenermethod written somewhere in a Customerclass if an event is triggered and on which to go next,is decided by the loginmethod of Customer class and given as a value of actionattribute.

The method specified in actionattribute should return a Stringvalue as the returned Stringvalue is used in page navigation.

Note: Many IDE provides visual support for JSF so you can drag and drop components instead of writing tedious coding for defining JSF components as shown above. Sun Studio Creator® is a free open source IDE that provides visual support for JSF and can be downloaded form Sun site.

The code examples are also built using this IDE.

```
class Customer {
    public void loginActionListener(ActionEvent e)
    {
    ……….……
    }
    public String login() {
       return “OK”;
    }}
```

Example Code: Hello User

The example code (“*hello user 1”*) is given along with the handout. It is strongly advised that you must see the lecture video in order to learn how this example is built.

User will provide a name in the text field and his/her name after appending “*hello”* to it,would be displayed on the same page.

44.2.5 JSF Validators

Validators make input validation simple and save developers hours of programming. JSF provides a set of validator classes for validating input values entered into input components.

needs.

Some built-in validators are:

- **DoubleRangeValidator**
  Any numeric type, between specified maximum and minimum values
- **LongRangeValidator**
  Any numeric type convertible to long, between specified maximum and minimum values
- **LengthValidator**
  Ensures that the length of a component's local value falls into a certain range (between minimum & maximum). The value must be of String type.

Example Code: *Hello User*

The example code (*"hello user 2"*) is given along with the handout. You can open it using Sun Studio Creator IDE. It is strongly advised that you must see the lecture video in order to learn how this example is built.

It is actually a modified version of the last example. This time, we'll make sure that user couldn't left blank the name field and must enter a name between ranges of 2 to 10 characters. If any condition fails, an appropriate message would be displayed.

44.2.6 JSF – Managed Bean-Intro

These are JavaBeans defined in the configuration file and are used to hold the data from JSF components. Managed beans represent the data model, and are passed between business logic and pages. Some other salient features are:

- Use the declarative model
- Entry point into the model and event handlers
- Can have beans with various states

Here is an example of a managed-bean element whose scope is *session*, meaning that an instance of this bean is created at the beginning of a user session.

```
<managed-bean>
        <managed-bean-name>myBean</managed-bean-name>
        <managed-bean-class>myPackage.MyBean</managed-bean-class>
        <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

44.2.7 JSF – Value Binding

Value binding expressions can be used inside of JSF components to:

- Automatically instantiate a JavaBean and place it in the request or session scope.
- Override the JavaBean's default values through its accessor methods.

- Quickly retrieve Map, List, and array contents from a JavaBean.
- Synchronize form contents with value objects across a number of requests.

The syntax of binding expressions is based on the JavaServer Pages (JSP) 2.0 Expression Language. In JSP, expressions are delimited with "${}", but in JSF they are delimited with "#{}".

44.2.8 JSF – Method Binding

Unlike a value binding, a method binding does not represent an accessor method. Instead, a method binding represents an activation method.

For example, binding an event handler to a method

```
<h:commandButton ……
actionListener="#{customer.loginActionListener}"
…… />
```

44.2.9 JSF Navigation

Page navigation determines the control flow of a Web application. JSF provides a default navigational handler and this behavior can be configured in configuration. However, you can do it visually in most tools like Sun Studio Creator

Note: We have quickly breezed through the JSF technology essentials due to shortage of time**.** You must explore it by yourself to excel on it. You can find the resources in the last handout to acquire further skills.

44.3 References:

- Java A Lab Course by Umair Javed
- Intrduction to JavaServer Faces by Sun
  http://java.sun.com
- JavaServer Faces Programming by Kumiawan

Note: Coding exercises in working condition for this lecture are also available on **"Downloads"** section of LMS.

Lecture 45: JavaServer Faces

In the last lecture, we have covered the basic nutshells of JSF. Having a belief on *"learning by doing"*, in this lecture another example is also given to show you the capabilities of JSF.

Example Code: Addition of Two Numbers

The example code (*"AddNumbers"*) is given along with the handout. It is strongly advised that you must see the lecture video in order to learn the making plus working of this example.

This example demonstrates the usage of value and method binding expressions, managed beans, and how to use page navigation technique using IDE etc.

45.1 Web Services

In the remaining handout, we'll take an overview of web services' potential, their types and working model. Resources are given at the end for those who are interested in learning new technologies.

45.1.1 Introduction

Web services are Web-based enterprise applications that use open, XML-based standards and transport protocols to exchange data with calling clients.

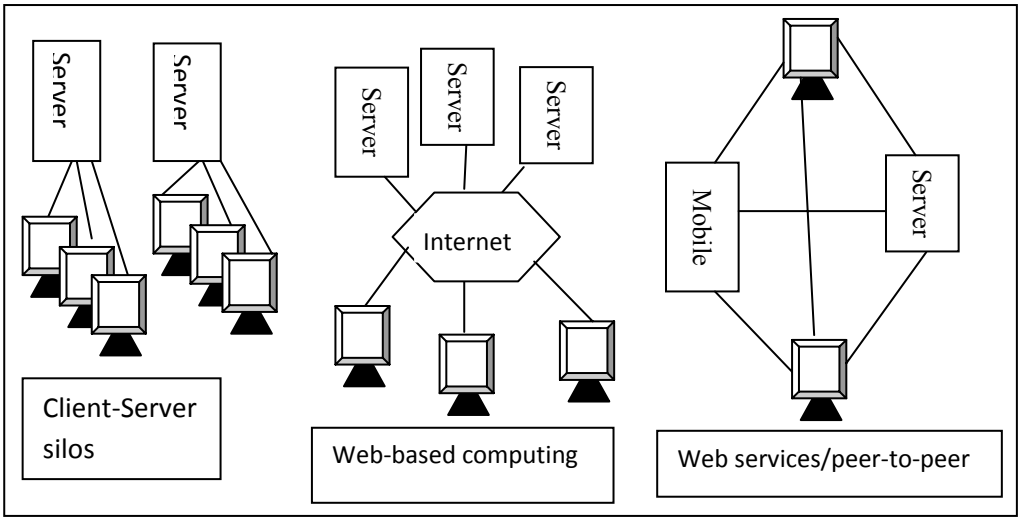
Web Service is becoming one of those overly overloaded buzzwords these days. Due to their increasing popularity, Java platform Enterprise Edition (J2EE) provides the APIs and tools you need to create and deploy interoperable web services and clients.

45.1.2 Web service, Definition by W3C

W3C recently has come up with a decent definition of web services. According to W3C,"A Web service is a software application identified by a URI, whose interfaces and binding are capable of being defined, described and discovered by XML artifacts and supports direct interactions with other software applications using XML based messages via internet-based protocols".

45.1.3 Distributed Computing Evolution

Let's think a little bit on how distributed computing technology has evolved.



In the beginning, things were built and deployed typically in the form of client and server model in which clients talk to a single server, for example, remote procedure calls (RPC).

The second phase can be called web-based computing in which many clients talk to many servers through the net. In this phase, communicating partners still have to go through some pre- arrangement in terms of what common object model they have to use or what common communication protocol they have to agree upon.

Finally, the web services model in which service users and service providers can be dynamically connected. And the pretty much every computing device and application participates as both service user and service provider.

45.1.4 Characteristics of Web services

Web services are XML-based throughout. Pretty much everything in the domain of Web services is defined in XML. For example, the format of the data being exchanged between service user and service provider is defined in XML or the description of web service is defined in XML.

Because the only contract that has to be agreed upon between service user and service provider is syntax and semantics of XML messages, as long as valid messages can be generated and understood, it does not matter what programming language is used. So a web service is said to be programming language independent.

Web services can be dynamically located and invoked. And typically they will be accessed and invoked over both internet and intranet.

45.1.5 Why Web services?

Interoperable
Connect across heterogeneous networks using ubiquitous web-based standards

Economical
Recycle components, no installation and tight integration of software

Automatic
No human intervention required even for highly complex transactions

Accessible
Legacy assets & internal apps are exposed and accessible on the web

Available
Services on any device, anywhere, anytime

Scalable
No limits on scope of applications and amount of heterogeneous applications

45.1.6 Types of Web service

Data providers
For example, a service providing stock quotes

Business-to-business process integration
For example, purchase orders

Enterprise application integration
Different applications work together simply by adding a webservice wrapper

Comparison between Web page & Web service

Just to give you a sense on the difference between a web page and a web service,consider the following table:

Web page	Web Service
Has a UI	No GUI
Interacts with user	Interacts with application
Works with web browser client	Works with any type of client

45.2 Web service Architectural Components

Following are the core building blocks of web service architecture:

- **Service Description-How do clients know how it works (which functions, parameters etc.)?**

At the minimum, you need a standard way of describing a web service that is universally

understood by all potential service users and service providers. This is important because without commonly agreed upon description of service, a service provider might have to produce individually tailored way of describing its service to all its potential service users.

Web Service Description Language (WSDL pronounced as viz-dal) is industry agreed upon XML language that can be used to describe web service. It provides XML format for describing web services in terms of methods, properties, data types and protocols.

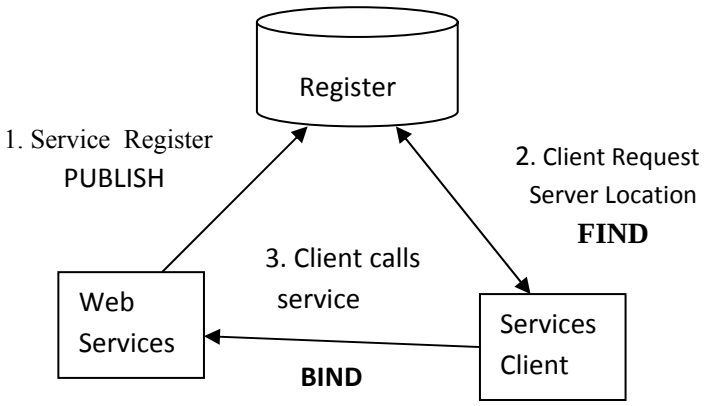
- **Service Registration (Publication) and Discovery**

There has to be registry by which a service can be published and discovered. Universal Description, Discovery & Integration (UDDI), a way to publish and find web services. A repository of web services on the internet where a machine or a human can find different web services. www.uddi.org

- **Service Invocation**

Then there has to be standard way of invoking a service. Finally, for business transactions in which secure and reliable message delivery is important, there has to be a standard electronic business framework.

The following figure represents simplified web service architecture and summarizes the working of web services:



45.3 References:

- Java A Lab Course by Umair Javed
- Web services overview by sang shin

Note: Coding exercises in working condition for this lecture are also available on **Downloads**" section of LMS.

45.4 Resources:

- An excellent resource for learning Java related technologies is:

http://www.apl.jhu.edu/~hall/java/



- http://java.sun.com
- http://www.javaworld.com
- http://www.theserverside.com
- http://www.jsfcentral.com
- http://www.jspolympus.com
- http://www.onjava.com