Java and J2EE

Scheme and Syllabus

Subject Code: 06IS753 **I.A. Marks: 25**
Hours / Week: 04 **Exam Hours: 03**
Total Hours: 52 **Exam Marks: 100**

PART - A

UNIT - 1

INTRODUCTION TO JAVA: Java and Java applications; Java Development Kit (JDK); Java is interpreted, Byte Code, JVM; Object-oriented programming; Simple Java programs. Data types and other tokens: Boolean variables, int, long, char, operators, arrays, white spaces, literals, assigning values; Creating and destroying objects; Access specifiers. Operators and Expressions: Arithmetic Operators, Bitwise operators, Relational operators, The Assignment Operator, The? Operator; Operator Precedence; Logical expression; Type casting; Strings Control Statements: Selection statements, iteration statements, Jump Statements.

6 Hours

UNIT - 2

CLASSES, INHERITANCE, EXCEPTIONS, APPLETS: Classes: Classes in Java; Declaring a class; Class name; Super classes; Constructors; Creating instances of class; Inner classes. Inheritance: Simple, multiple, and multilevel inheritance; Overriding, overloading. Exception handling**:** Exception handling in Java. The Applet Class: Two types of Applets; Applet basics; Applet Architecture; An Applet skeleton; Simple Applet display methods; Requesting repainting; Using the Status Window; The HTML APPLET tag; Passing parameters to Applets; getDocumentbase() and getCodebase(); ApletContext and showDocument(); The AudioClip Interface; The AppletStub Interface; Output to the Console.

6 Hours

UNIT - 3

MULTI THREADED PROGRAMMING, EVENT HANDLING: Multi Threaded Programming: What are threads? How to make the classes threadable; Extending threads; Implementing runnable; Synchronization; Changing state of the thread; Bounded buffer problems, read-write problem, producer-consumer problems. Event Handling: Two event handling mechanisms; The delegation event model; Event classes; Sources of events; Event listener interfaces; Using the delegation event model; Adapter classes; Inner classes.

7 Hours

UNIT - 4

SWINGS: Swings: The origins of Swing; Two key Swing features; Components and Containers; The Swing Packages; A simple Swing Application; Create a Swing Applet; Jlabel and

ImageIcon; JTextField;The Swing Buttons; JTabbedpane; JScrollPane; JList; JComboBox; JTable.

7 Hours

PART - B

UNIT - 5

JAVA 2 ENTERPRISE EDITION OVERVIEW, DATABASE ACCESS: Overview of J2EE and J2SE. The Concept of JDBC; JDBC Driver Types; JDBC Packages; A Brief Overview of the JDBC process; Database Connection; Associating the JDBC/ODBC Bridge with the Database; Statement Objects; ResultSet; Transaction Processing; Metadata, Data types; Exceptions.

6 Hours

UNIT - 6

SERVLETS: Background; The Life Cycle of a Servlet; Using Tomcat for Servlet Development; A simple Servlet; The Servlet API; The Javax.servlet Package; Reading Servlet Parameter; The Javax.servlet.http package; Handling HTTP Requests and Responses; Using Cookies; Session Tracking.

7 Hours

UNIT - 7

JSP, RMI: Java Server Pages (JSP): JSP, JSP Tags, Tomcat, Request String, User Sessions, Cookies, Session Objects. Java Remote Method Invocation: Remote Method Invocation concept; Server side, Client side.

6 Hours

UNIT - 8

ENTERPRISE JAVA BEANS: Enterprise java Beans; Deployment Descriptors; Session Java Bean, Entity Java Bean; Message-Driven Bean; The JAR File.

7 Hours

TEXT BOOKS:

1. **Java - The Complete Reference** – Herbert Schildt, 7th Edition, Tata McGraw Hill, 2007.
2. **J2EE - The Complete Reference** – Jim Keogh, Tata McGraw Hill, 2007.

REFERENCE BOOKS:

1. **Introduction to JAVA Programming** – Y. Daniel Liang, 6th Edition, Pearson Education, 2007.
2. **The J2EE Tutorial** – Stephanie Bodoff et al, 2nd Edition, Pearson Education, 2004.

Table of Contents

UNIT-1

INTRODUCTION TO JAVA

Java and Java applications; Java Development Kit (JDK); Java is interpreted, Byte Code, JVM; Objectoriented programming; Simple Java programs.

Data types and other tokens:

Boolean variables, int, long, char, operators,

arrays, white spaces, literals, assigning values; Creating and destroying objects; Access specifiers.

Operators and Expressions: Arithmetic Operators,

Bitwise operators, Relational operators, The Assignment Operator, The? Operator; Operator Precedence; Logical expression; Type casting; Strings Control Statements: Selection statements, iteration statements, Jump Statements.

1.1. Introduction to Java

- Java is an object-oriented programming language developed by Sun Microsystems, a company best known for its high-end Unix workstations.
- Java is modeled after C++
- Java language was designed to be small, simple, and portable across platforms and operating systems, both at the source and at the binary level (more about this later).
- Java also provides for portable programming with applets. Applets appear in a Web page much in the same way as images do, but unlike images, applets are dynamic and interactive.
- Applets can be used to create animations, figures, or areas that can respond to input from the reader, games, or other interactive effects on the same Web pages among the text and graphics.

1.1.2 Java Is Platform-Independent

Platform-independence **is a program's capability of moving easily from one computer system to another.**

- Platform independence is one of the most significant advantages that Java has over other programming languages, particularly for systems that need to work on many different platforms.
- Java is platform-independent at both the source and the binary level.

1.1.2 Java Development Kit (JDK)- Byte code

- *Bytecodes* are a set of instructions that look a lot like machine code, but are not specific to any one processor
- Platform-independence doesn't stop at the source level, however. Java binary files are also platform-independent and can run on multiple platforms without the need to recompile the source. Java binary files are actually in a form called bytecodes.

1.1.3 Object-Oriented Programming

- Many of Java's object-oriented concepts are inherited from C++, the language on which it is based, but it borrows many concepts from other object-oriented languages as well.

- Java includes a set of class libraries that provide basic data types, system input and output capabilities, and other utility functions.

- These basic classes are part of the Java development kit, which also has classes to support networking, common Internet protocols, and user interface toolkit functions.
- Because these class libraries are written in Java, they are portable across platforms as all Java applications are.

1.1.4 Creating a simple Java Program

Hello World example :

```
class HelloWorld {
public static void main (String args[]) {
System.out.println("Hello World! ");
}
}
```

This program has two main parts:

- All the program is enclosed in a class definition—here, a class called Hello World.

- The body of the program (here, just the one line) is contained in a method (function) called main(). In Java applications, as in a C or C++ program, main() is the first method (function) that is run when the program is executed.

1.1.5 Compiling the above program :

- In Sun's JDK, the Java compiler is called javac.

```
javac HelloWorld.java
```

- When the program compiles without errors, a file called HelloWorld.class is created, in the same directory as the source file. This is the Java bytecode file.
- Then run that bytecode file using the Java interpreter. In the JDK, the Java interpreter is called simply java.

```
java HelloWorld
```

If the program was typed and compiled correctly, the output will be :

"Hello World!"

1.2 Variables and Data Types

- Variables are locations in memory in which values can be stored. They have a name,

a type, and a value.

- Java has three kinds of variables: instance variables, class variables, and local variables.
- Instance variables, are used to define attributes or the state for a particular object. Class variables are similar to instance variables, except their values apply to all that class's instances (and to the class itself) rather than having different values for each object.
- Local variables are declared and used inside method definitions,

for example, for index counters in loops, as temporary variables, or to hold values that you need only inside the method definition itself

Variable declarations consist of a type and a variable name:

Examples :

```
int myAge;

String myName;
boolean isTired;
```

1.2.1 Integer types.

Type Size Range

byte 8 bits —128 to 127

short 16 bits —32,768 to 32,767

int 32 bits —2,147,483,648 to 2,147,483,647

—9223372036854775808 to 9223372036854775807 long 64 bits

1.2.2 Floating-point

This is used for numbers with a decimal part. Java floating-point numbers are compliant with IEEE 754 (an international standard for defining floating-point numbers and arithmetic).

There are two floating-point types: float (32 bits, single-precision) and double (64 bits, double-precision).

1.2.3 Char

The char type is used for individual characters. Because Java uses the Unicode character set, the char type has 16 bits of precision, unsigned.

1.2.4 Boolean

The boolean type can have one of two values, true or false. Note that unlike in other C-like languages, boolean is not a number, nor can it be treated as one. All tests of Boolean variables should test for true or false.

1.2.5 Literals

Literals are used to indicate simple values in your Java programs.

Number Literals

- There are several integer literals. 4, for example, is a decimal integer literal of type int
- A decimal integer literal larger than an int is automatically of type long.
- Floating-point literals usually have two parts: the integer part and the decimal part—for example, 5.677777.

Boolean Literals

Boolean literals consist of the keywords true and false. These keywords can be used anywhere needed a test or as the only possible values for boolean variables.

1.2.6 Character Literals

Character literals are expressed by a single character surrounded by single quotes: 'a', '#', '3', and so on. Characters are stored as 16-bit Unicode characters.

1.3Expressions and Operators

- Expressions are the simplest form of statement in Java that actually accomplishes something. *Expressions* are statements that return a value.
- *Operators* are special symbols that are commonly used in expressions.

Arithmetic and tests for equality and magnitude are common examples of expressions. Because they return a valuethe value can be assigned to a variable or test that value in other Java statements.

Operators in Java include arithmetic, various forms of assignment, increment and decrement, and logical operations.

1.3.1 Arithmetic

Java has five operators for basic arithmetic

Arithmetic operators.

Operator	*Meaning*	*Example*
+	Addition	3 + 4
—	Subtraction	5 —
*	Multiplication	5 * 5
/	Division	14 / 7
%	Modulus	20 % 7

Example program :

```
class ArithmeticTest {

public static void main (String args[]) {
short x = 6;
      int y = 4;
      float a = 12.5f;
      float b = 7f;
      System.out.println("x + y = " +                    (x + y));
      System.out.println("x - y = " +                    (x - y));
      System.out.println("x / y = " +                    (x / y));
      System.out.println("a is " + a + ", b is " + b);
      System.out.println("a / b = " + (a / b));
} }
```

Assignment operators.

Expression	*Meaning*
x += y	x = x + y
x —= y	x = x — y
x *= y	x = x * y
x = x / y	x /= y

Incrementing and Decrementing

x++ increments the value of x by 1 just as if you had used the expression x = x + 1. Similarly x-- decrements the value of x by 1.

Exercise : write the difference between :

y = x++;

y = ++x;

Comparison operators.

Operator	*Meaning*	*Example*
==	Equal	x == 3
!=	Not equal	x
<	Less than	x <
>	Greater than	x >
<=	Less than or equal	x

>= Greater than or equal to x >= 3 **Logical**

Operators

- Expressions that result in boolean values (for example, the comparison operators) can be combined by using logical operators that represent the logical combinations
- AND, OR, XOR, and logical NOT.
- For AND combinations, use either the & or &&. The expression will be true only if both expressions are also true
- For OR expressions, use either | or ||. OR expressions result in true if either or both of the operands is also true
- In addition, there is the XOR operator ^, which returns true only if its operands are different (one true and one false, or vice versa) and false otherwise (even if both are true).
- In general, only the && and || are commonly used as actual logical combinations. &, |, and ^ are more commonly used for bitwise logical operations.
- For NOT, use the ! operator with a single expression argument. The value of the NOT expression is the negation of the expression; if x is true, !x is false.

Bitwise Operators

These are used to perform operations on individual bits in integers.

Operator	*Meaning*
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR

<< Left shift

>> Right shift

>>> Zero fill right shift

~ Bitwise complement

<<= Left shift assignment (x = x << y)

>>= Right shift assignment (x = x >> y)

>>>= Zero fill right shift assignment (x = x >>> y) x&=y
AND assignment (x = x & y)

x|=y OR assignment (x + x | y)

x^=y XOR assignment (x = x ^ y)

Operator Precedence

Operator precedence determines the order in which expressions are evaluated. This, in some cases, can determine the overall value of the expression. For example, take the following expression:

y = 6 + 4 / 2

Depending on whether the 6 + 4 expression or the 4 / 2 expression is evaluated first, the value of y can end up being 5 or 8. In general, increment and decrement are evaluated before arithmetic, arithmetic expressions are evaluated before comparisons, and comparisons are evaluated before logical expressions. Assignment expressions are evaluated last.

1.4 Arrays

Arrays in Java are actual objects that can be passed around and treated just like other objects.

Arrays are a way to store a list of items. Each slot of the array holds an individual element, and you can place elements into or change the contents or those slots as you need to.

Three steps to create an array:

1. Declare a variable to hold the array.

2. Create a new array object and assign it to the array variable.

3. Store things in that array.

E.g.

```
String[ ] names;
names = new String[10];
names [1] =     “n1”;
names[2] =     ‘n2’;
```

. . .

1.4.1 Multidimensional Arrays

Java does not support multidimensional arrays. However, you can declare and create an array of arrays (and those arrays can contain arrays, and so on, for however many dimensions you need), and access them as you would C-style multidimensional arrays:

```
int coords[] [] = new int[12] [12];

coords[0] [0] = 1; coords[0] [1] = 2;
```

1.5 Control Statement

1.5.1 if Conditionals

- The if conditional, which enables you to execute different bits of code based on a simple test in Java, is nearly identical to if statements in C.
- if conditionals contain the keyword if, followed by a boolean test, followed by a statement (often a block statement) to execute if the test is true:
- if (x < y)

```
System.out.println("x is smaller than y");
```

An optional else keyword provides the statement to execute if the test is false:

```
if (x < y)

System.out.println("x is smaller than y"); else
System.out.println("y is bigger");
```

1.5.2 The Conditional Operator

An alternative to using the if and else keywords in a conditional statement is to use the conditional operator, sometimes called the ternary operator.

The *conditional operator* is a *ternary operator* because it has three terms.

Syntax : test ? trueresult : falseresult

The *test* is an expression that returns true or false, just like the test in the if statement. If the test is true, the conditional operator returns the value of *trueresult;* if it's false, it returns the value of *falseresult.* For example, the following conditional tests the values of x and y, returns the smaller of the two, and assigns that value to the variable smaller:

int smaller = x < y ? x : y;

The conditional operator has a very low precedence; that is, it's usually evaluated only after all its subexpressions are evaluated. The only operators lower in precedence are the assignment operators..

1.5.3 switch Conditionals

This is the switch or case statement; in Java it's switch and behaves as it does in C:

switch (test) { case
valueOne:

resultOne;

break;

case valueTwo:

resultTwo;

break;

case valueThree:

resultThree;

break; ...

default: defaultresult;

}

In the switch statement, the test (a primitive type of byte, char, short, or int) is compared with each of the case values in turn. If a match is found, the statement, or statements after the test is executed. If no match is found, the default statement is executed. The default is optional, so if there isn't a match in any of the cases and default doesn't exist, the switch statement completes without doing anything.

1.5.4 for Loops

The for loop, as in C, repeats a statement or block of statements some number of times until a condition is matched. for loops are frequently used for simple iteration in which you repeat a block of statements a certain number of times and then stop, but you can use for loops for just about any kind of loop.

The for loop in Java looks roughly like this:

for (initialization; test; increment) { statements;

}

The start of the for loop has three parts:

- Initialization is an expression that initializes the start of the loop. If you have a loop index, this expression might declare and initialize it, for example, int i = 0. Variables that you declare in this part of the for loop are local to the loop itself; they cease existing after the loop is finished executing. Test is the test that occurs after each pass of the loop. The test must be a boolean expression or function that returns a boolean value, for example, i < 10. If the test is true, the loop executes. Once the test is false, the loop stops executing
- Increment is any expression or function call. Commonly, the increment is used to change the value of the loop index to bring the state of the loop closer to returning false and completing.

The statement part of the for loop is the statements that are executed each time the loop iterates. Just as with if, you can include either a single statement here or a block; the previous example used a block because that is more common. Here's an example of a for loop that initializes all the values of a String array to null strings:

```
String strArray[] = new String[10]; int
i; // loop index

for (i = 0; i < strArray.length; i++)
strArray[i] = "";
```

1.5.5 while and do Loops

Finally, there are while and do loops. while and do loops, like for loops, enable a block of Java code to be executed repeatedly until a specific condition is met. Whether you use a for loop, a while, or a do is mostly a matter of your pro gramming style. while and do loop, are exactly the same as in C and C++ except their test condition must be a boolean.

1.5.6 while Loops

The while loop is used to repeat a statement or block of statements as long as a particular condition is true. while loops look like this:

```
while (condition) {
bodyOfLoop; }
```

The *condition* is a boolean expression. If it returns true, the while loop executes the statements in bodyOfLoop and then tests the condition again, repeating until the condition is false:

```
int count = 0;

while ( count < array 1 .length && array 1 [count] !=0) {
array2[count] = (float) array1[count++];

}
```

1.5.7 do...while Loops

The do loop is just like a while loop, except that do executes a given statement or block until the condition is false. The main difference is that while loops test the condition

before looping, making it possible that the body of the loop will never execute if the condition is false the first time it's tested. do loops run the body of the loop at least once before testing the condition. do loops look like this:

```
do {

bodyOfLoop;

} while (condition);
```

Here, the bodyOfLoop part is the statements that are executed with each

```
int x = 1;

do {

System.out.println("Looping, round " + x); x++;

} while (x <= 10);
```

Here's the output of these statements:

Looping, round 1 Looping, round 2 Looping, round 3 Looping, round 4 Looping, round 5 Looping, round 6

Looping, round Looping, round Looping, round Looping, round

UNIT-2: CLASSES, INHERITANCE, EXCEPTIONS, APPLETS

- **Classes:**

Classes in Java; Declaring a class; Class name; Super classes; Constructors; Creating instances of class; Inner classes. Inheritance: Simple, multiple, and multilevel inheritance; Overriding, overloading.

- **Exception handling:** Exception handling in Java.

- **The Applet Class:**

Two types of Applets; Applet

basics; Applet Architecture; An Applet skeleton; Simple Applet display methods; Requesting repainting; Using the Status Window; The HTML APPLET tag; Passing parameters to Applets; getDocumentbase() and getCodebase(); ApletContext and showDocument(); The AudioClip Interface; The AppletStub Interface; Output to the Console.

2.1. Defining Classes, Class Name

- To define a class, use the class keyword and the name of the class:

  class MyClassName {

  ... }

- If this class is a subclass of another class, use extends to indicate the superclass of this

  class:

  class myClassName extends mySuperClassName {

  ...

  }

- If this class implements a specific interface, use implements to refer to that interface: class MyRunnableClassName implements Runnable {

...

}

Super Classes

- Each class has a superclass (the class above it in the hierarchy), and each class can have one or more subclasses (classes below that class in the hierarchy). Classes further down in the hierarchy are said to inherit from classes further up in the hierarchy.
- Subclasses inherit all the methods and variables from their superclasses—that is, in any particular class, if the superclass defines behavior that your class needs, you don't have to redefine it or copy that code from some other class. Your class automatically gets that behavior from its superclass, that superclass gets behavior from its superclass, and so on all the way up the hierarchy.
- **At the top of the Java class hierarchy is the class Object;** all classes inherit from this one superclass. Object is the most general class in the hierarchy; it defines behavior inherited by all the classes in the Java class hierarchy. Each class farther down in the hierarchy adds more information and becomes more tailored to a specific purpose.

E.g.

public class HelloAgainApplet extends java.applet.Applet { }

2.2 Constructors, Creating instances of a class

The following example demonstrates the creation of classes, creating objects and the usage of constructors

class Motorcycle { }

create some instance variables for this class

String make; String color; boolean engineState;

Add some behavior (methods) to the class.

void startEngine() {

if (engineState == true)

System.out.println("The engine is already on."); else {

engine State = true;

System.out.println("The engine is now on."); }

}

The program looks like this now :

```
class Motorcycle {                engine is already
                                  on.");
String make;
                                  engine is now
String color;                     on.");

boolean engineState;

void startEngine() {

if (engineS tate == true)
System.out.println("The
}
```

The showAtts method prints the current values of the instance variables in an instance of your Motorcycle class. Here's what it looks like:

void showAtts() {

System. out .println ("This motorcycle is a " + color + " " + make);

if (engineState == true)

System.out.println("The engine is on."); else System.out.println("The engine is off.");

The showAtts method prints two lines to the screen: the make and color of the motorcycle object, and whether or not the engine is on or off.

2.2.1 Add the main method

public static void main (String args[]) { Motorcycle m = new Motorcycle(); m.make = "Yamaha RZ350";

m.color = "yellow";

System.out.println("Calling showAtts..."); m.showAtts(); System.out.println("———

— "); System.out.println("Starting

engine..."); m.startEngine();

System.out.println("————————");

System.out.println("Calling showAtts..."); m.showAtts(); System.out.println("———

— "); System.out.println("Starting

engine..."); m.startEngine();

}

With the main() method, the Motorcycle class is now an application, and you can compile it again and this time it'll run. Here's how the output should look:

Calling showAtts...

This motorcycle is a yellow Yamaha RZ350 The engine is off.

Starting engine... The engine is now on.

Calling showAtts...

This motorcycle is a yellow Yamaha RZ350 The engine is on.

Starting engine...

The engine is already on.

2.3. Inheritance

Inheritance is a powerful mechanism that means when you write a class you only have to specify how that class is different from some other class; inheritance will give you automatic access to the information contained in that other class.

With inheritance, all classes—those you write, those from other class libraries that you use, and those from the standard utility classes as well—are arranged in a strict hierarchy

2.3.1 Single and Multiple Inheritance

Single inheritance means that each Java class can have only one superclass (although any given superclass can have multiple subclasses).

In other object-oriented programming languages, such as C++, classes can have more than one superclass, and they inherit combined variables and methods from all those classes. This is called multiple inheritance.

Multiple inheritance can provide enormous power in terms of being able to create classes that factor just about all imaginable behavior, but it can also significantly complicate class definitions and the code to produce them. **Java makes inheritance simpler by being only singly inherited.**

2.3.2 Overriding Methods

- When a method is called on an object, Java looks for that method definition in the class of that object, and if it doesn't find one, it passes the method call up the class hierarchy until a method definition is found.
- Method inheritance enables you to define and use methods repeatedly in subclasses without having to duplicate the code itself.
- However, there may be times when you want an object to respond to the same methods but have different behavior when that method is called. In this case, you can override that method. Overriding a method involves defining a method in a subclass that has the same signature as a method in a superclass. Then, when that method is called, the method in the subclass is found and executed instead of the one in the superclass.

2.3.3 Creating Methods that Override Existing Methods

To override a method, all you have to do is create a method in your subclass that has the same signature (name, return type, and parameter list) as a method defined by one of your class's superclasses. Because Java executes the first method definition it finds that matches the signature, this effectively "hides" the original method definition. Here's a simple example

The PrintClass class.

```
class PrintClass {

int x = 0; int y = 1;

void printMe() {

System.out.println("X is " + x + ", Y is " + y);

System.out.println("I am an instance of the class " +

this.getClass().getName());
```

Create a class called PrintSubClass that is a subclass of (extends) PrintClass.

class PrintSubClass extends PrintClass { int z = 3;

public static void main(String args[]) { PrintSubClass obj = new PrintSubClass(); obj .printMe();

}

}

Here's the output from PrintSubClass:

X is 0, Y is 1

I am an instance of the class PrintSubClass

In the main() method of PrintSubClass, you create a PrintSubClass object and call the printMe() method. Note that PrintSubClass doesn't define this method, so Java looks for it in each of PrintSubClass's superclasses—and finds it, in this case, in PrintClass. because printMe() is still defined in PrintClass, it doesn't print the z instance variable.

To call the original method from inside a method definition, use the super keyword to pass the method call up the hierarchy:

void myMethod (String a, String b) { // do stuff here

super.myMethod(a, b);

// maybe do more stuff here }

The super keyword, somewhat like the this keyword, is a placeholder for this class's superclass. You can use it anywhere you can use this, but to refer to the superclass rather than to the current class.

2.4 Exception handling

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

2.4.1 The Three Kinds of Exceptions

- Checked exceptions *are subject* to the Catch or Specify Requirement. All exceptions are checked exceptions, except for those indicated by Error, RuntimeException, and their subclasses.

- Errors *are not subject* to the Catch or Specify Requirement. Errors are those exceptions indicated by Error and its subclasses.

- Runtime exceptions *are not subject* to the Catch or Specify Requirement. Runtime exceptions are those indicated by Runtime Except ion and its subclasses.

Valid Java programming language code must honor the *Catch or Specify Requirement.* This means that code that might throw certain exceptions must be enclosed by either of the following:

• A try statement that catches the exception. The try must provide a handler for the exception, as described in Catching and Handling Exceptions.

• A method that specifies that it can throw the exception. The method must provide a throws clause that lists the exception, as described in Specifying the Exceptions Thrown by a Method.

Code that fails to honor the Catch or Specify Requirement will not compile.

This example describes how to use the three exception handler components — the try, cat ch, and finally blocks

2.4.2 try block

• The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block. In general, a try block looks like the following.

```
try {
code
}
catch and finally blocks . . .
```

Example :

```
private Vector vector;
private static final int SIZE = 10;
PrintWriter out = null;
try {
System.out.println("Entered try statement");
out = new PrintWriter(new FileWriter("OutFile.txt"));
for (int i = 0; i < SIZE; i++) {
out.println("Value at: " + i + " = " + vector.elementAt(i));
}
}
```

The catch Blocks

You associate exception handlers with a try block by providing one or more catch blocks directly after the try block. No code can be between the end of the try block and the beginning of the first catch block.

try {

} catch *(ExceptionType name) {*

} catch *(ExceptionType name) {*

}

Each catch block is an exception handler and handles the type of exception indicated by its argument

finally block

The runtime system always executes the statements within the finally block regardless of what happens within the try block. So it's the perfect place to perform cleanup.

The following finally block for the write Li st method cleans up and then closes the PrintWriter.

finally {

if (out != null) {

System.out.println("Closing PrintWriter"); out . close();

} else {

System.out.println("PrintWriter not open"); }

}

2.5 The Applet Class

Applet Basics

- An applet is a special kind of Java program that a browser enabled with Java technology can download from the internet and run.
- An applet is typically embedded inside a web-page and runs in the context of the browser.
- An applet must be a subclass of the java.applet.Applet class, which provides the standard interface between the applet and the browser environment.
- Simple example :

public class HelloWorld extends java.applet.Applet {

public void paint(java.awt.Graphics g) {

g.drawString("Hello World!",50,25); System.out.println("Hello World!");

}

}

An applet can be included in an HTML page, much in the same way an image is included in a page.

When Java technology enabled Browser is used to view a page that contains an applet, the applet's code is transferred to your system and executed by the browser's Java Virtual Machine (JVM)

Two Types of Applets

1 .Local applet - operate in single machine browser which is not connected in network, 2.Remote applet - remote applet operate over internet via network.

Applet Architecture

Event driven :

An applet waits until an event occurs.

The *AWT* notifies the applet about an event by calling event handler that has been provided by the applet.

The applet takes appropriate action and then quickly return control to *AWT* All *Swing* components descend from the AWT Container class

User initiates interaction with an Applet *(and not the other way around)* **An Applet Skeleton**

import java.awt.*;

import javax.swing.*;

/*

<applet code="AppletSkel" width=300 height=100>

</applet>

*/

public class AppletSkel extends JApplet { // Called first.

public void init() {

// initialization

}

/* Called second, after init(). Also called whenever the applet is restarted. */

public void start() {

// start or resume execution

}

// Called when the applet is stopped. public void stop () {

// suspends execution

}

/* Called when applet is terminated. This is the last

method executed. */

public void destroy() {

// perform shutdown activities }

// Called when an applet's window must be restored. public void paint(Graphics g) {

// redisplay contents of window

}

}

5.1 Simple Applet Display Methods

void drawstring(String message, int x, int y) - void setBackground(Color newColor) void setForeground(Color newColor) **Example :**

public class SimpleApplet extends Applet { public void paint (Graphics g) { g.drawString("First Applet", 50, 50); }

}

Requesting Repainting

repaint( ) function is called when you have changed something and want your changes to show up on the screen

repaint( ) is a *request*--it might not happen

When you call repaint( ), Java schedules a call to update(Graphics g)

Here's what update does:

public void update(Graphics g) {

// Fills applet with background color, then

paint(g);

Using The Status Window

Syntax : public void **showStatus**(String status)

Parameters:

status - a string to display in the status window.

Requests that the argument string be displayed in the "status window".

Many browsers and applet viewers provide such a window, where the application can inform users of its current state.

Example :

```
import java.applet. *; import java.awt.*;

public class NetExample extends Applet

{

private AppletContext browser = null;

private Button showStatus = new Button("Show Status"); public void init()

{

Panel panel = new Panel();

panel.setLayout(new GridLayout(1 ,2));

panel.add(showStatus);

setLayout(new BorderLayout()); add("South", panel);

browser = getAppletContext(); }

public boolean action(Event e, Object o)

{

if (e.target == showStatus)

browser.showStatus("Here is something for your status line ..."); return true;

}

}
```

2.5 The HTML Applet Tag

- The APPLET tag is used to start an applet from both an HTML document and from an applet viewer.
- An applet viewer will execute each APPLET tag that it finds in a separate window,while web browsers like Netscape Navigator, Internet Explorer, and HotJava will allow many applets on a single page.
- The syntax for the standard APPLET tag is shown here. Bracketed items are optional.

< APPLET

[CODEBASE = *codebaseURL]*

CODE = *appletFile*

[ALT = *alte rnate Text]*

[NAME = *appletInstanceName]*

WIDTH = *pixels* HEIGHT = *pixels*

[ALIGN = *alignment]*

[VSPACE = *pixels]* [HSPACE = *pixels]*

>

[< PARAM NAME = *AttributeName* VALUE = *Attribute Value*>] [< PARAM NAME = *AttributeName2* VALUE = *Attribute Value*>] . . .

[HTML Displayed in the absence of Java]

</APPLET>

- **CODEBASE** is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag). The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read.
- **CODE** is a required attribute that gives the name of the file containing your applet's compiled **.class** file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.
- **ALT** is an optional attribute used to specify a short text message that should be displayed if the browser understands the APPLET tag but can't currently run Java applets. This is distinct from the alternate HTML you provide for browsers that don't support applets.
- **WIDTH AND HEIGHT** are required attributes that give the size (in pixels) of the applet display area.

- **ALIGN** is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values:
- LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

VSPACE AND HSPACE These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes.

PARAM NAME AND VALUE The PARAM tag allows you to specify appletspecific arguments in an HTML page. Applets access their attributes with the **getParameter( )** method.

Passing Parameters to Applets

Parameters are passed to applets in NAME=VALUE pairs in <PARAM> tags between the opening and closing APPLET tags.

• Inside the applet, you read the values passed through the PARAM tags with the getParameter() method of the java.applet.Applet class.

The applet parameter "Message" is the string to be drawn.

```
import java.applet. *; import
java.awt.*;

public class DrawStringApplet extends Applet { private
 String defaultMessage = "Hello!"; public void
 paint(Graphics g) {

String inputFromPage = this .getParameter("Mes sage");

   if (inputFromPage == null) inputFromPage = defaultMessage;
   g.drawString(inputFromPage, 50, 25);

  }
}
```

HTML file that references the above applet.

```
<HTML> <HEAD>

<TITLE> Draw String </TITLE>

</HEAD>
```

<BODY>

This is the applet:<P>

<APPLET code="DrawStringApplet" width="3 00" height="50">

<PARAM name="Message" value="Howdy, there!"> This page will be very boring if your

browser doesn't understand Java.

</APPLET>

</BODY> </HTML>

getDocumentBase() and getCodeBase()

Syntax : public URL **getDocumentBase()**

Returns:

the URL of the document that contains this applet.

- Gets the URL of the document in which this applet is embedded.
- For example, suppose an applet is contained within the document:

  http://java.sun.com/products/jdk/1.2/index.html
- The document base is:

  http://java.sun.com/products/jdk/1.2/index.html

  Syntax : public URL **getCodeBase()**

  Returns:

  the base URL of the directory which contains this applet.
- Gets the base URL. This is the URL of the directory which contains this applet.
- Example segments:

URL codeBase = getCodeBase();

Image myImage = getImage(codeBase, "images/myimage.gif"); **Applet Context and showDocument()**

AppletContext is an interface that provides the means to control the browser environment in which the applet is running.

The AudioClip Interface

- The AudioC lip interface is a simple abstraction for playing a sound clip.
- Multiple Audi oC lip items can be playing at the same time, and the resulting sound is mixed together to produce a composite.

- It has the following methods :

play

public abstract void play()

- **loop** Starts playing this audio clip. Each time this method is called, the clip is restarted from the beginning.
- **stop** public abstract void loop()

public abstract void stop()

Stops playing this audio clip.

The AppletStub Interface

The AppletStub interface provides a way to get information from the run-time browser environment.

The Applet class provides methods with similar names that call these methods. Methods

public abstract boolean isActive ()

The isActive() method returns the current state of the applet. While an applet is initializing, it is not active, and calls to isActive() return false. The system marks the applet active just prior to calling start(); after this point, calls to isActive() return true.

- *public abstract URL getDocumentBase ()*

The getDocumentBase() method returns the complete URL of the HTML file that loaded the applet. This method can be used with the getImage() or getAudioClip() methods to load an image or audio file relative to the HTML file.

- *public abstract URL getCodeBase ()*

The getCodeBase() method returns the complete URL of the *.class* file that contains the applet. This method can be used with the getImage() method or the getAudioClip() method to load an image or audio file relative to the *.class* file.

- *public abstract String getParameter (String name)*

The getParameter() method allows you to get parameters from <PARAM> tags within the <APPLET> tag of the HTML file that loaded the applet. The name parameter of getParameter() must match the name string of the <PARAM> tag; name is case insensitive. The return value of getParameter() is the value associated with name; it is always a String regardless of the type of data in the tag. If name is not found within the <PARAM> tags of the <APPLET>, getParameter() returns null.

- *public abstract AppletContext getAppletContext ()*

The getAppletContext() method returns the current AppletContext of the applet. This is part of the stub that is set by the system when setStub() is called.

- *public abstract void appletResize (int width, int height)*

The appletResize() method is called by the resize method of the Applet class. The method changes the size of the applet space to width x height. The browser must support changing the applet space; if it doesn't, the size remains unchanged

Output To the Console

The drawString method can be used to output strings to the console. The position of the text can also be specified.

The following prog shows this concept:

```
public class ConsolePrintApplet1 extends java.applet.Applet

{

public void init () {

// Put code between this line

double x = 5.0; double y = 3.0;

System.out.println( "x * y = "+ (x*y) );

System.out.println( "x / y = "+ (x/y) );

                    //    // and this line.

}

// Paint message in the applet window. Public
```

UNIT-3 : MULTI THREADED PROGRAMMING, EVENT HANDLING

Multi Threaded Programming:

What are threads? How to make the classes threadable; Extending threads; Implementing runnable; Synchronization; Changing state of the thread; Bounded buffer problems, read-write problem, producer-consumer problems.

Event Handling:

Two event handling mechanisms; The delegation event model; Event classes; Sources of events; Event listener interfaces; Using the delegation event model; Adapter classes; Inner classes.

3.1 What are Threads?

A thread is a single path of execution of code in a program.

- A Multithreaded program contains two or more parts that can run concurrently.
- Each part of such a program is called a Thread.
- Each thread defines a separate path of execution. Multithreading is a specialized form of Multitasking.

3.1 How to make the classes threadable

A class can be made threadable in one of the following ways

(1) implement the Runnable Interface and apply its run() method.

(2) extend the Thread class itself.

1. Implementing Runnable Interface: The easiest way to create a thread is to create a class that implements the Runnable interface. To implement Runnable, a class need only implement a single method called run().

The Format of that function is public void run().

2. Extending Thread: The second way to create a thread is to create a new class that extends the Thread class and then to create an instance of this class. This class must override the run() method which is the entry point for the new thread.

1.2 Extending Threads

You can inherit the Thread class as another way to create a thread in your program. When you declare an instance of your class, you'll also have access to members of the Thread class. Whenever your class inherits the Thread class, you must override the run() method, which is an entry into the new thread. The following example shows how to inherit the Thread class and how to override the run() method. This example defines the MyThread class, which inherits the Thread class. The constructor of the MyThread class calls the constructor of the Thread class by using the super keyword and passes it the name of the new thread, which is My thread. It then calls the start() method to activate the new thread. The start() method calls the run() method of the MyThread class

```
class MyThread extends Thread {

MyThread() {

super("My thread");

start();

}
```

```
public void run() {
System.out.println("Child thread started");
System.out.println("Child thread terminated");
} }
class Demo {
public static void main (String args[]){ new
MyThread(); System.out.println("Main
thread started");
System.out.println("Main thread terminated");
} }
```

3.2 Implementing Runnable

The example in the next segment demonstrates the use of Runnable and its implementation.

Synchronization

1. Two or more threads accessing the same data simultaneously may lead to loss of data integrity. In order to avoid this java uses the concept of monitor. A monitor is an object used as a mutually exclusive lock.

2. At a time only one thread can access the Monitor. A second thread cannot enter the monitor until the first comes out. Till such time the other thread is said to be waiting.

3. The keyword Synchronized is use in the code to enable synchronization and it can be used along with a method.

Changing the state of thread

There might be times when you need to temporarily stop a thread from processing and then resume processing, such as when you want to let another thread use the current resource. You can achieve this objective by defining your own suspend and resume methods, as shown in the following example. This example defines a MyThread class. The MyThread class defines three methods: the run() method, the suspendThread() method, and the resumeThread() method. In addition, the MyThread class declares the instance variable suspended, whose value is used to indicate whether or not the thread is suspended.

```
class MyThread implements Runnable {
String name;
Thread t;
```

```
boolean suspended;
MyThread() {
t = new Thread(this, "Thread");
suspended = false ; t.start();
}
public void run() {
try {
for (int i = 0; i < 10; i++) { System.out.println("Thread: " + i ); Thread.sleep(200);
synchronized (this) {
while (suspended) {
wait();
}
}
}
} catch (InterruptedException e ) { System.out.println("Thread: interrupted."); }
System.out.println("Thread exiting.");
}
void suspendThread() { suspended = true;
}
synchronized void resumeThread() {
suspended = false;
notify();
}
}
class Demo {
public static void main (String args [] ) { MyThread t1 = new MyThread();
try{
Thread.sleep(1000); t1.suspendThread(); System.out.println("Thread: Suspended");
Thread.sleep(1000);
```

```
t1.resumeThread(); System.out.println("Thread: Resume");

} catch ( InterruptedException e) {

}

try {

t1.t.join();

} catch ( InterruptedException e) { System.out.println (

"Main Thread: interrupted"); }

}

}
```

3.3 Bounded Buffer Problem

Problem Description :

In computer science the producer-consumer problem (also known as the **bounded-buffer**

problem) is a classical example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a

common, fixed-size buffer.

The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time the consumer is consuming the data (i.e. removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer. This is the code for solving the above stated:

```
class BufferItem {

public volatile double value = 0;  // multiple threads access public volatile boolean occupied = false; // so make these `volatile' }

class BoundedBuffer { // designed for a single producer thread and // a single consumer thread

private int numSlots = 0;

private BufferItem[] buffer = null;

private int putIn = 0, takeOut = 0;

// private int count = 0;

public BoundedBuffer(int numSlots) {
```

```
if (numSlots <= 0) throw new IllegalArgumentException("numSlots<=0"); this.numSlots = numSlots;

buffer = new BufferItem[numSlots];

for (int i = 0; i < numSlots; i++) buffer[i] = new BufferItem();

putIn = (putIn + 1) % numSlots;

// count++;// race condition!!! }

public double fetch() {

double value;

while (!buffer[takeOut].occupied) // busy wait Thread.currentThread().yield();

value = buffer[takeOut].value;     // C

buffer [takeOut] .occupied = false;          // D takeOut = (takeOut + 1) % numSlots;

//            count--;  // race condition!!! return value;

}

}
```

Read-Write problem

- Read/Write locks provides a synchronization mechanism that allow threads in an application to more accurately reflect the type of access to a shared resource that they require.
- Many threads can acquire the same read/write lock if they acquire a shared read lock on the read/write lock object.
- Only one thread can acquire an exclusive write lock on a read/write lock object.
- When an exclusive write lock is held, no other threads are allowed to hold any lock.

```
public class Read WriteLock{

private int readers = 0;

private int writers  = 0; private int writeRequests = 0;

public synchronized void lockRead() throws InterruptedException{ while(writers > 0 || writeRequests > 0){

wait();

}

readers++;
```

```
}
public synchronized void unlockRead(){ readers--;
notifyAll();
}
public synchronized void lockWrite() throws InterruptedException{
writeRequests++;
while(readers > 0 || writers > 0){ wait();
}
writeRequests--;
writers++;
}
public synchronized void unlockWrite() throws InterruptedException{ writers--;
notifyAll();
}
}
```

Producer-Consumer problems

- In producer/consumer synchronizations, producer processes make items available to consumer processes.
- Examples are a message sender and a message receiver, or two machines working on items in sequence.
- The synchronization here must ensure that the consumer process does not consume more items than have been produced. If necessary, the consumer process is blocked (must wait) if no item is available to be consumed.
- Producer and consumer processes are coupled by a buffer to allow asynchronous production and consumption.
- The buffer can be bounded (have a capacity limit) or unbounded (be able to store an unlimited number of items).

3.4 Event Handling

In Java, events represent all activity that goes on between the user and the application.

Two event handling mechanisms :

Delegation event model : It defines standard and consistent mechanisms to generate and process events. Here the source generates an event and sends it to on or more listeners. The listener simply waits until it receives an event. Once it is obtained, It processes this event and returns. Listeners should register themselves with a source in order to receive an even notification. Notifications are sent only to listeners that want to receive them.

Events

In the delegation model, an *event* is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are : pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed.

Event Classes

The classes that represent events are at the core of Java's event handling mechanism. **EventObject :** It is at the root of the Java event class hierarchy in **java.util.** It is the superclass for all events. Its one constructor is shown here: EventObject(Object *src)* Here, *src* is the object that generates this event. EventObject contains two methods: getSource( ) and toString( ). The getSource( ) method returns the source of the event.

EventObject is a superclass of all events.

The ActionEvent Class :

An **ActionEvent** is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. The **ActionEvent** class defines four integer constants that can be used to identify any modifiers associated with an action event: **ALT_MASK, CTRL_MASK, META_MASK,** and **SHIFT_MASK.**

ActionEvent has these three constructors: ActionEvent(Object *src,*

int *type,* String *cmd)* ActionEvent(Object *src,* int *type,*

String *cmd,* int *modifiers)*

ActionEvent(Object *src,* int *type,* String *cmd,* long *when,* int *modifiers)* Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type,* and its command string is *cmd.* The argument *modifiers* indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. The *when* parameter specifies when the event occurred

- **The AdjustmentEvent Class** An **AdjustmentEvent** is generated by a scroll bar

- **The ComponentEvent Class** A **ComponentEvent** is generated when the size, position, or visibility of a component is changed. There are four types of component events
- **The ContainerEvent Class** A **ContainerEvent** is generated when a component is added to or removed from a container
- **The FocusEvent Class :** A **FocusEvent** is generated when a component gains or loses input focus
- **The InputEvent Class :** The abstract class **InputEvent** is a subclass of **ComponentEvent** and is the superclass for component input events. Its subclasses are **KeyEvent** and **MouseEvent.**
- **The ItemEvent Class :** An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected
- **The KeyEvent Class** A **KeyEvent** is generated when keyboard input occurs.
- **The MouseEvent Class** There are eight types of mouse events
- **The MouseWheelEvent Class** The **MouseWheelEvent** class encapsulates a mouse wheel event.
- **The TextEvent Class** Instances of this class describe text events. These are generated

by text fields and text areas when characters are entered by a user or program.

- **The WindowEvent Class** There are ten types of window events. The **WindowEvent**

class defines integer constants that can be used to identify them.

Sources of Events

Event Source Description :

- **Button -** Generates action events when the button is pressed.
- **Checkbox -** Generates item events when the check box is selected or deselected.
- **Choice -** Generates item events when the choice is changed.
- **List -** Generates action events when an item is double-clicked; generates item events
- when an item is selected or deselected.
- **Menu Item -** Generates action events when a menu item is selected; generates item

events when a checkable menu item is selected or deselected.

- **Scrollbar -** Generates adjustment events when the scroll bar is manipulated.
- **Text components -** Generates text events when the user enters a character.
- **Window -** Generates window events when a window is activated, closed, deactivated,

deiconified, iconified, opened, or quit.

Event Listener Interfaces

Listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package.

When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument.

Interface Description

ActionListener - Defines one method to receive action events.

AdjustmentListener - Defines one method to receive adjustment events. **ComponentListener -** Defines four methods to recognize when a component is hidden, moved, resized, or shown.

ContainerListener - Defines two methods to recognize when a component is added to or removed from a container.

FocusListener - Defines two methods to recognize when a component gains or loses keyboard focus.

ItemListener - Defines one method to recognize when the state of an item changes.

KeyListener - Defines three methods to recognize when a key is pressed, released, or typed.

MouseListener - Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.

MouseMotionListener - Defines two methods to recognize when the mouse is dragged or moved.

MouseWheelListener - Defines one method to recognize when the mouse wheel is moved.

TextListener - Defines one method to recognize when a text value changes. **WindowFocusListener -** Defines two methods to recognize when a window gains or loses input focus.

WindowListener Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Using the Delegation Event Model

Applet programming using the delegation event model is done following these two steps:

1. Implement the appropriate interface in the listener so that it will receive the type of event desired.

2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

Adapter Classes

Definition : An adapter class provides an empty implementation of all methods in an event listener interface.

- Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface.

- New class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

- For example, the **MouseMotionAdapter** class has two methods, **mouseDragged( )** and **mouseMoved( ).** The signatures of these empty methods are exactly as defined in the **MouseMotionListener** interface. If you were interested in only mouse drag events, then you could simply extend **MouseMotionAdapter** and implement **mouseDragged( ).** The empty implementation of **mouseMoved( )** would handle the mouse motion events for you.

The following example demonstrates an adapter.

```
// Demonstrate an adapter. import java.awt.*;

import java.awt.event.*; import java.applet. *;

/*

<applet code="AdapterDemo" width=300 height=1 00>

</applet>

*/

public class AdapterDemo extends Applet {

public void init() {

addMouseListener(new MyMouseAdapter(this));

addMouseMotionListener(new MyMouseMotionAdapter(this));

} }

class MyMouseAdapter extends MouseAdapter {

AdapterDemo adapterDemo;
```

```
public MyMouseAdapter(AdapterDemo adapterDemo) {
this.adapterDemo = adapterDemo;
}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
adapterDemo.showStatus("Mouse clicked");
} }
class MyMouseMotionAdapter extends MouseMotionAdapter {
AdapterDemo adapterDemo;
public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
this.adapterDemo = adapterDemo;
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
adapterDemo.showStatus("Mouse dragged");
} }
```

Inner Classes

Consider the applet shown in the following listing. It *does not* use an inner class. Its goal is to display the string

"Mouse Pressed" in the status bar of the applet viewer or browser when the mouse is pressed. There are two top-level classes in this program. **MousePressedDemo** extends **Applet,** and **MyMouseAdapter** extends **MouseAdapter.** The **init( )** method of **MousePressedDemo** instantiates **MyMouseAdapter** and provides this object as an argument to the **addMouseListener( )** method.

Notice that a reference to the applet is supplied as an argument to the **MyMouseAdapter** constructor. This reference is stored in an instance variable for later use by the **mousePressed( )**

method.

```
// This applet does NOT use an inner class.
import java.applet. *;
```

```
import java.awt.event.*;
/*
<applet code="MousePressedDemo" width=200 height=100> </applet>
*/
public class MousePressedDemo extends Applet {
public void init() {
addMouseListener(new MyMouseAdapter(this));
} }
class MyMouseAdapter extends MouseAdapter { MousePressedDemo mousePressedDemo;
public MyMouseAdapter(MousePressedDemo mousePressedDemo) { this.mousePressedDemo
= mousePressedDemo;
}
public void mousePressed(MouseEvent me) { mousePressedDemo.showStatus("Mouse
Pressed.");
```

UNIT-4: SWINGS

SWINGS:

The origins of Swing; Two key Swing features;

Components and Containers; The Swing Packages;

A simple Swing Application; Create a Swing Applet;

Jlabel and ImageIcon;

JTextField;

The Swing Buttons;

JTabbedpane;

JScrollPane;

JList;

JComboBox;

JTable.

Swing is built on top of AWT and is entirely written in Java, using AWT's lightweight component support. In particular, unlike AWT, t he architecture of Swing components makes it easy to customize both their appearance and behavior. Components from AWT and Swing can be mixed, allowing you to add Swing support to existing AWT-based programs. For example, swing components such as JSlider, JButton and JCheckbox could be used in the same program with standard AWT labels, textfields and scrollbars.

4.1 Three parts

Component set (subclasses of JComponent) Support classes, Interfaces

Swing Components and Containers

Swing components are basic building blocks of an application. Swing toolkit has a wide range of various widgets. Buttons, check boxes,sliders, list boxes etc. Everything a programmer needs for his job. In this section of the tutorial, we will describe several useful components.

JLabel Component

JLabel is a simple component for displaying text, images or both. It does not react to input events.

JCheckBox

JCheckBox is a widget that has two states. On and Off. It is a box with a label JSlider

JSlider is a component that lets the user graphically select a value by sliding a knob within a bounded interval

JComboBox

Combobox is a component that combines a button or editable field and a drop-down list. The user can select a value from the drop-down list, which appears at the user's request.

JProgressBar

A progress bar is a widget that is used, when we process lengthy tasks. It is animated so that the user knows, that our task is progressing

JToggleButton

JToggleButton is a button that has two states. Pressed and not pressed. You toggle between these two states by clicking on it

Containers

Swing contains a number of components that provides for grouping other components together.

In AWT, such components extended java.awt.Container and included Panel, Window, Frame, and Dialog.

1.1 A Simple Container

JPanel is Swing's version of the AWT class Panel and uses the same default layout, FlowLayout. JPanel is descended directly from JComponent.

JFrame is Swing's version of Frame and is descended directly from that class. The components added to the frame are referred to as its contents; these are managed by the contentPane. To add a component to a JFrame, we must use its contentPane instead.

JInternalFrame is confined to a visible area of a container it is placed in. It can be iconified , maximized and layered.

JWindow is Swing's version of Window and is descended directly from that class. Like Window, it uses BorderLayout by default.

JDialog is Swing's version of Dialog and is descended directly from that class. Like Dialog, it uses BorderLayout by default. Like JFrame and JWindow,

JDialog contains a rootPane hierarchy including a contentPane, and it allows layered and glass panes. All dialogs are modal, which means the current

thread is blocked until user interaction with it has been completed. JDialog class is intended as the basis for creating custom dialogs; however, some

of the most common dialogs are provided through static methods in the class JOptionPane.

Jlabel and ImageIcon

Syntax : public class **JLabel**

extends JComponent

implements SwingConstants, Accessible

- It is a display area for a short text string or an image, or both.
- A label does not react to input events. As a result, it cannot get the keyboard focus.
- A label can display a keyboard alternative as a convenience for a nearby component that has a keyboard alternative but can't display it.
- A JLabel object can display either text, an image, or both.
- By default, labels are vertically centered in their display area.
- Text-only labels are leading edge aligned, by default; image-only labels are

  horizontally centered, by default.

• Can use the setIconTextGap method to specify how many pixels should appear between the text and the image. The default is 4 pixels.

ImageIcon

Syntax :

public **ImageIcon**(String filename)

{

this(filename, filename);

}

Creates an ImageIcon from the specified file. The image will be preloaded by using MediaTracker to monitor the loading state of the image.

The specified String can be a file name or a file path. When specifying a path, use the Internet-standard forward-slash ("/") as a separator. (The string is converted to an URL, so the forward-slash works on all systems.)

For example, specify:

new ImageIcon("images/myImage.gif")

The description is initialized to the filename string.

Example of JLabel with ImageIcon :

import java.awt.FlowLayout; **import** java.awt.HeadlessException;

import javax.swing.Icon;

import javax.swing.ImageIcon; **import** javax.swing.JFrame; **import** javax.swing.JLabel;

public class Main **extends** JFrame {

public Main() **throws** HeadlessException { setSize(300, 300);

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); setLayout(**new** FlowLayout(FlowLayout.LEFT));

Icon icon = **new** ImageIcon("a.png");

JLabel label1 = **new** JLabel("Full Name :", icon, JLabel.LEFT);

JLabel label2 = **new** JLabel("Address :", JLabel.LEFT); label2.setIcon(**new** ImageIcon("b.png"));

getContentPane().add(label1); getContentPane().add(label2);

}

public static void main(String[] args) { **new** Main().setVisible(**true**);

}

}

JTextField

- JTextField is a lightweight component that allows the editing of a single line of text.
- JTextField is intended to be source-compatible with java.awt.TextField where it is reasonable to do so. This component has capabilities not found in the java.awt.TextField class.
- JTextField has a method to establish the string used as the command string for the action event that gets fired.
- The java.awt.TextField used the text of the field as the command string for the ActionEvent.
- JTextField will use the command string set with the setActionCommand method if not null, otherwise it will use the text of the field as a compatibility with java.awt.TextField.

1. Swing Package

Syntax :

public class **JButton** extends AbstractButton implements Accessible

An implementation of a "push" button. Buttons can be configured, and to some degree controlled, by Actions. Using an Action with a button has many benefits beyond directly configuring a button.

package com.ack.gui.swing.simple;

import java.awt.*;

import java.awt.event.WindowAdapter; import java.awt.event.WindowEvent; import javax.swing.*;

public class SimpleSwingButtons extends JFrame {

public static void main(String[] argv) {

SimpleSwingButtons myExample = new SimpleSwingButtons("Simple Swing Buttons");

}

```
public SimpleSwingButtons( String title ) {

super( title );

setSize( 150, 150 );

add WindowListener( new WindowAdapter() { public void windowClosing( WindowEvent we
) { dispose();

System.exit( 0 );

}

} );

init();

setVisible( true );

}

private void init() {

JPanel my_panel = new JPanel();

my_panel.setLayout( new GridLayout( 3, 3 ) ); for( int i = 1; i < 10; i++ ) {

ImageIcon icon = new ImageIcon( i + ".gif" ); JButton jb = new JButton( icon );

jb.setToolTipText( i + ".gif" );

my_panel.add( jb );

}

getContentPane().add( my_panel );

my_panel.setBorder( BorderFactory.createEtchedBorder() );

}

}
```

JTabbedpane

Syntax : public class JTabbedPane

extends JComponent

implements Serializable, Accessible, SwingConstants

- A component that lets the user switch between a group of components by clicking on a tab with a given title and/or icon.

- Tabs/components are added to a TabbedPane object by using the addTab and insertTab methods.
- A tab is represented by an index corresponding to the position it was added in, where the first tab has an index equal to 0 and the last tab has an index equal to the tab count minus
- The TabbedPane uses a Single SelectionModel to represent the set of tab indices and the currently selected index. If the tab count is greater than 0, then there will always be a selected index, which by default will be initialized to the first tab. If the tab count is 0, then the selected index will be -1.

JScrollPane

Syntax : public class JScrollPane

extends JComponent

implements ScrollPaneConstants, Accessible

- Provides a scrollable view of a lightweight component.
- A JScrollPane manages a viewport, optional vertical and horizontal scroll bars, and optional row and column heading viewports.
- The JViewport provides a window, or "viewport" onto a data source -- for example, a text file. That data source is the "scrollable client" (aka data model) displayed by the JViewport view.
- A JScrollPane basically consists of JScrollBars, a JViewport, and the wiring between them, as shown in the diagram at right.

JList

Syntax : public class JList

extends JComponent

implements Scrollable, Accessible

A component that allows the user to select one or more objects from a list. A separate model, ListModel, represents the contents of the list.

// Create a JList that displays the strings in data[]

String[] data = {"one", "two", "three", "four"}; JList dataList = new JList(data);

JComboBox

Syntax : public class JComboBox

extends JComponent

implements ItemSelectable, ListDataListener, ActionListener, Accessible

- A component that combines a button or editable field and a drop-down list.
- The user can select a value from the drop-down list, which appears at the user's request.
- If you make the combo box editable, then the combo box includes an editable field into which the user can type a value.

JTable

Syntax : public class JTable

extends JComponent

implements TableModelListener, Scrollable, TableColumnModelListener, ListSelectionListener, CellEditorListener, Accessible

- The JTable is used to display and edit regular two-dimensional tables of cells.
- The JTable has many facilities that make it possible to customize its rendering and

editing but provides defaults for these features so that simple tables can be set up

easily.

- For example, to set up a table with 10 rows and 10 columns of numbers:

```
TableModel dataModel = new AbstractTableModel() {
public int getColumnCount() { return 10; }
public int getRowCount() { return 10;}
public Object getValueAt(int row, int col) { return new Integer(row*col); }
}
JTable table = new JTable(dataModel); JScrollPane scrollpane = new JScrollPane(table);
```

UNIT-5: JAVA 2 ENTERPRISE EDITION OVERVIEW, DATABASE ACCESS:

Overview of J2EE and J2SE.

The Concept of JDBC

JDBC Driver Types;

JDBC Packages; A Brief Overview of the JDBC process; Database Connection; Associating the JDBC/ODBC Bridge with the Database; Statement Objects; ResultSet; Transaction Processing; Metadata, Data types; Exceptions.

Overview of J2EE and J2SE

- Java™2 Platform, Enterprise Edition (J2EE™) technology provides a component-based approach to the design, development, assembly, and deployment of enterprise applications.
- The J2EE platform gives you a multitiered distributed application model, the ability to reuse components, integrated XML-based data interchange, a unified security model, and flexible transaction control.
- Vendors and customers enjoy the freedom to choose the products and components that best meet their business and technological requirements.

try {

5. 1 The Concept of JDBC

- The JDBC (Java Database Connectivity) API defines interfaces and classes for writing database applications in Java by making database connections.
- Using JDBC you can send SQL, PL/SQL statements to almost any relational database. JDBC is a Java API for executing SQL statements and supports basic SQL functionality.
- It provides RDBMS access by allowing you to embed SQL inside Java code.

Overview of JDBC Process

Before you can create a java jdbc connection to the database, you must first import the java.sql package.

import java.sql.*; The star (*) indicates that all of the classes in the package java.sql are to be imported.

Java application calls the JDBC library. JDBC loads a driver which talks to the database. We can change database engines without changing database code.

Establishing Database Connection and Associating JDBC/ODBC bridge **1. Loading a database driver,**

- In this step of the jdbc connection process, we load the driver class by calling Class.forName() with the Driver class name as an argument. Once loaded, the Driver class creates an instance of itself.
- A client can connect to Database Server through JDBC Driver. Since most of the Database servers support ODBC driver therefore JDBC-ODBC Bridge driver is commonly used.
- The return type of the Class.forName (String ClassName) method is “Class”. Class is a class in java.lang package.

```
Class.forName(”sun.jdbc.odbc.JdbcOdbcDriver”); //Or any other driver

}

catch(Exception x) {

System.out.println( “Unable to load the driver class!” );

}
```

5.2 Creating a oracle jdbc Connection

The JDBC DriverManager class defines objects which can connect Java applications to a JDBC driver. DriverManager is considered the backbone of JDBC architecture. DriverManager class manages the JDBC drivers that are installed on the system.

- Its getConnection() method is used to establish a connection to a database. It uses

a username, password, and a jdbc url to establish a connection to the database and

returns a connection object.

- A jdbc Connection represents a session/connection with a specific database. Within the context of a Connection, SQL, PL/SQL statements are executed and results are returned. An application can have one or more connections with a single database, or it can have many connections with different databases.
- A Connection object provides metadata i.e. information about the database, tables, and fields. It also contains methods to deal with transactions.

•Each subprotocol has its own syntax for the source. We’re using the jdbc odbc subprotocol, so the DriverManager knows to use the sun.jdbc.odbc.JdbcOdbcDriver.

```
try{

Connection dbConnection=DriverManager.getConnection(url,”loginName”,”Pas sword”)

}

catch( SQLException x ){

System.out.println( “Couldn’t get connection!” );

}
```

5.3. Creating a JDBC Statement object

- Once a connection is obtained we can interact with the database.

Connection interface defines methods for interacting with the database via the established connection.

- To execute SQL statements, you need to instantiate a Statement object from your connection object by using the createStatement() method.
- Statement statement = dbConnection.createStatement();
- A statement object is used to send and execute SQL statements to a database.

Three kinds of Statements

- **Statement:** Execute simple sql queries without parameters.

  Statement createStatement()

  Creates an SQL Statement object.

- **Prepared Statement:** Execute precompiled sql queries with or without parameters. Prep aredS tatement prepare Statement(String sql)

  returns a new PreparedStatement object. Prep aredStatement objects are precompiled SQL statements.

- **Callable Statement:** Execute a call to a database stored procedure. CallableStatement prepareCall(String sql)

  returns a new CallableStatement object. CallableStatement objects are SQL stored procedure call statements.

5.4. Executing a SQL statement with the Statement object, and returning a jdbc resultSet.

- Statement interface defines methods that are used to interact with database via the execution of SQL statements.
- The Statement class has three methods for executing statements:

  executeQuery(), executeUpdate(), and execute().

- For a SELECT statement, the method to use is executeQuery .
- For statements that create or modify tables, the method to use is executeUpdate. Note: Statements that create a table, alter a table, or drop a table are all examples of DDL statements and are executed with the method executeUpdate. execute() executes an SQL statement that is written as String object.

ResultSet provides access to a table of data generated by executing a Statement. The table rows are retrieved in sequence. A ResultSet maintains a cursor pointing to its current row of data. The next() method is used to successively step through the rows of the tabular results.

ResultSetMetaData Interface holds information on the types and properties of the columns in a ResultSet. It is constructed from the Connection object.

UNIT -6: SERVLETS

Background; The Life Cycle of a Servlet; Using Tomcat for

Servlet Development;

A simple Servlet; The Servlet API; The Javax.servlet

Package; Reading Servlet Parameter;

The Javax.servlet.http package;

Handling HTTP Requests and Responses; Using Cookies; Session Tracking

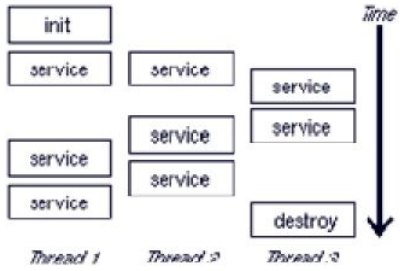
Background

Definition : Servlets are modules of Java code that run in a server application (hence the name "Servlets", similar to "Applets" on the client side) to answer client requests.

- Servlets are not tied to a specific client-server protocol but they are most commonly used with HTTP and the word "Servlet" is often used in the meaning of "HTTP Servlet".
- Servlets make use of the Java standard extension classes in the packages

  javax. servlet (the basic Servlet framework) and javax. servlet .http
- Typical uses for HTTP Servlets include:
  - Processing and/or storing data submitted by an HTML form.
  - Providing dynamic content, e.g. returning the results of a database query to the client.
  - Managing state information on top of the stateless HTTP, e.g. for an online shopping cart system which manages shopping carts for many concurrent customers and maps every request to the right customer.

6.1 Servlet Life Cycle



The life cycle of a servlet is controlled by the container in which the servlet has been deployed. When a request is mapped to a servlet, the container performs the following steps:

1. If an instance of the servlet does not exist, the web container:
   a. Loads the servlet class
   b. Creates an instance of the servlet class
   c. Initializes the servlet instance by calling the init method. Initialization is covered in Initializing a Servlet
2. Invokes the service method, passing a request and response object.
3. If the container needs to remove the servlet, it finalizes the servlet by calling the servlet's destroy method.

6.1.1 A servlet example

```
import java.io.*; import javax.servlet.*; import javax.servlet.http. *;

public class HelloClientServlet extends HttpServlet

{

protected void doGet(HttpServletRequest req,

HttpServletResponse res)

throws ServletException, IOException

{

res.setContentType("text/html"); PrintWriter out = res.getWriter();
out.println("<HTML><HEAD><TITLE>Hello Client !</TITLE>"+

"</HEAD><BODY>Hello Client !</BODY></HTML>");

out.close();

}

public String getServletInfo()

{

return "HelloClientServlet 1.0 by Stefan Zeiger";

}

}
```

6.1.2. Servlet API

- Two packages contain the classes and interfaces that are required to build servlets. They are javax.servlet and javax.servlet.http.
- The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines life cycle methods.

6.1.3 The servlet packages :

- The *javax.servlet package* contains a number of classes and interfaces that describe

  and define the contracts between a servlet class and the runtime environment provided for an instance of such a class by a conforming servlet container.

- The *Servlet* interface is the central abstraction of the servlet API.

- All servlets implement this interface either directly, or more commonly, by extending a class that implements the interface.
- The two classes in the servlet API that implement the *Servlet* interface are *GeneriISErvlet* and *HttpServlet* .
- For most purposes, developers will extend *HttpServlet* to implement their servlets while implementing web applications employing the HTTP protocol.
- The basic *Servlet* interface defines a *service* method for handling client requests. This method is called for each request that the servlet container routes to an instance of a servlet.

6.1.4 Handling HTTP requests and responses :

- Servlets can be used for handling both the GET Requests and the POST Requests.
- The HttpServlet class is used for handling HTTP GET Requests as it has som specialized methods that can efficiently handle the HTTP requests. These methods are;

doGet()

doPost()

doPut()

doDelete() doOptions() doTrace() doHead()

An individual developing servlet for handling HTTP Requests needs to override one of these methods in order to process the request and generate a response. The servlet is invoked dynamically when an end-user submits a form.

Example:

<form name="F1" action=/servlet/ColServlet> Select the color:

<select name = "col" size = "3">

<option value = "blue">Blue</option> <option value = "orange">Orange</option> </select>

<input type = "submit" value = "Submit"> </form>

Here's the code for ColServlet.java that overrides the doGet() method to retrieve data from the HTTP Request and it then generates a response as well.

// import the java packages that are needed for the servlet to work

import java.io .*;

import javax.servlet. *;

import javax.servlet.http. *;

```
// defining a class

public class ColServlet extends HttpServlet {

public void doGet(HttpServletRequest request,HttpServletResponse response) throws
ServletException, IOException

// request is an object of type HttpServletRequest and it's used to obtain information

// response is an object of type HttpServletResponse and it's used to generate a response //
throws is used to specify the exceptions than a method can throw

{

String colname = request.getParameter("col");

// getParameter() method is used to retrieve the selection made by the user
response.setContentType("text/html");

PrintWriter info = response.getWriter();

info .println("The color is: ");

info .println(col);

info.close();

}

}
```

6.2 Cookies

Cookies are small bits of textual information that a Web server sends to a browser and that the browser returns unchanged when visiting the same Web site or domain later.

By having the server read information it sent the client previously, the site can provide visitors with a number of conveniences like:

- Identifying a user during an e-commerce session..
- Avoiding username and password.
- Customizing a site.
- Focusing advertising.

To send cookies to the client, a servlet would create one or more cookies with the appropriate names and values via **new Cookie (name, value)**

6.2.1 Placing Cookies in the Response Headers

- The cookie is added to the Set-Cookie response header by means of the addCookie method of HttpServletResponse. For example:

  Cookie userCookie = new Cookie("user", "uid 1234"); response.addCookie(userCookie);

- To send cookies to the client, you created a Cookie then used addCookie to send a Set-Cookie HTTP response header.

- To read the cookies that come back from the client, call getCookies on the HttpServletRequest. This returns an array of Cookie objects corresponding to the values that came in on the Cookie HTTP request header

- Once this array is obtained, loop down it, calling getName on each Cookie until find one matching the name you have in mind. You then call getValue on the matching Cookie, doing some processing specific to the resultant value.

  public static String getCookieValue(Cookie[] cookies, String cookieName,

  String defaultValue) {

  for(int i=0; i<cookies.length; i++) {

  Cookie cookie = cookies[i];

  if (cookieName.equals(cookie.getName()))

  return(cookie.getValue());

  }

  return(defaultValue);

  }

- **Session Tracking**

- Why id it needed : In many internet applications it is important to keep track of the session when the user moves from one page to another or when there are different users logging on to the same website at the same time

- There are three typical solutions to this problem.

Cookies.

- HTTP cookies can be used to store information about a shopping session, and

- each subsequent connection can look up the current session and then extract

- information about that session from some location on the server machine.

- This is an excellent alternative, and is the most widely used approach.

URL Rewriting.

- You can append some extra data on the end of each URL that identifies the session, and the server can associate that session identifier with data it has stored about that session.
- This is also an excellent solution, and even has the advantage that it works with browsers that don't support cookies or where the user has disabled cookies.
- However, it has most of the same problems as cookies, namely that the server-side program has a lot of straightforward but tedious processing to do. In addition, you have to be very careful that every URL returned to the user (even via indirect means like
- Hidden form fields.
- HTML forms have an entry that looks like the following: <INPUT TYPE="HIDDEN" NAME="session" VALUE="...">.
- This means that, when the form is submitted, the specified name and value are included in the GET or POST data.
- This can be used to store information about the session.
  - However, it has the major disadvantage that it only works if every page is dynamically generated, since the whole point is that each session has a unique identifier.

• Servlets solution :

- The HttpSession API. is a high-level interface built on top of cookies or URL-rewriting. In fact, on many servers, they use cookies if the browser supports them, but automatically revert to URL-rewriting when cookies are unsupported or explicitly disabled.
- The servlet author doesn't need to bother with many of the details, doesn't have to explicitly manipulate cookies or information appended to the URL, and is automatically given a convenient place to store data that is associated with each session.

example,

```
HttpSession session = request.getSession(true); ShoppingCart previousItems =

(ShoppingCart)session.getValue("previousItems"); if (previousItems != null) {

doSomethingWith(previousItems);

} else {

previousItems = new ShoppingCart(...);

doSomethingElseWith(previousItems);

}
```

UNIT-7: JSP, RMI

Java Server Pages (JSP):

JSP, JSP Tags, Tomcat, Request String, User Sessions, Cookies, Session Objects.

Java Remote Method Invocation:

Remote Method Invocation concept; Server side, Client side.

- JavaServer Pages (JSP) is a Sun Microsystems specification for combining Java with HTML to provide dynamic content forWeb pages.
- When you create dynamic content, JSPs are more convenient to write than HTTP servlets because they allow to embed Java code directly into HTML pages, in contrast with HTTP servlets, in which you embed HTML inside Java code.
- JSP is part of the Java 2 Enterprise Edition (J2EE).
- JSP enables to separate the dynamic content of aWeb page from its presentation.
- It caters to two different types of developers: HTML developers, who are responsible for the graphical design of the page, and Java developers, who handle the development of software to create the dynamic content.

7.1 JSP Tags

The following table describes the basic tags that you can use in a JSP page. Each shorthand tag has an XML equivalent.

JSP Tag	Syntax	Description
Scriptlet	<% java_code %> . . . or use the XML equivalent: <jsp:scriptlet> java_code </jsp:scriptlet>	Embeds Java source code Scriptlet in yourHTML page. The Java code is executed and its output is inserted in sequence with the rest of the HTML in the page.
Directive	<% @ *dir-type dir-attr* %> . . . or use the XML equivalent: <jsp:directive.*dir_type dir_attr* />	*Directives* contain messages to the application server. A directive can also contain name/value pair attributes in the form attr=”value”, which provides additional instructions to the application server.
Declarations	<%! declaration %> . . . or use XML equivalent... <jsp:declaration> declaration; </jsp:declaration>	Declares a variable or method that can be referenced by other declarations, scriptlets, or expressions in the page
Expression	<%= expression %>	Defines a Java
	. . . or use XML	that is
Actions	<jsp:useBean ... >	Provide access to

Tomcat

- Apache Tomcat is an implementation of the Java Servlet and JavaServer Pages technologies.
- The Java Servlet and JavaServer Pages specifications are developed under the Java Community Process.
- Apache Tomcat is developed in an open and participatory environment and released under the Apache Software

Simple example

```
<% @ page info="a hello world example" %>
<html>
<head><title>Hello, World</title></head>
<body bgcolor="#ffffff" background="background.gif">
<% @ include file="dukebanner.html" %>
<table>
<tr>
<td width=150>   </td>
<td width=250 align=right> <h1>Hello, World !</h1> </td> </tr>
</table>
</body></html>
```

Request string

```
<%@ page import="hello.NameHandler" %> <jsp:useBean id="mybean" scope="page"
class="hello.NameHandler" />
<jsp :setProperty name="mybean" property="*" />
<html>
<head><title>Hello, User</title></head>
<body bgcolor="#ffffff" background="background.gif">
<% @ include file="dukebanner.html" %>
<table border="0" width="700">
<tr>
<td width="150">   </td>
```

```
<td width="550">
<h1>My name is Duke. What's yours?</h1>
</td> </tr> <tr>
<td width="150"   </td>
<td width="550">
<form method="get">
<input type="text" name="username" size="25">
<br>
<input type="submit" value="Submit">
<input type="reset" value="Reset">
</td> </tr> </form>
</table>
<%
if ( request.getParameter("username") != null ) {
%>
<% @ include file="response.jsp" %>
<%
}
%> </body> </html>
```

The data the user enters is stored in the request object, which usually implements javax.servlet.HttpServletRequest (or if your implementation uses a different protocol, another interface that is subclassed from javax.servlet.ServletRequest). You can access the request object directly within a scriptlet.

Sessions and Cookies in JSP

- In JSP cookie are the object of the class **javax.servlet.http.Cookie.**
- This class is used to creates a cookie, a small amount of information sent by a servlet to a Web browser, saved by the browser, and later sent back to the server.
- A cookie has a name, a single value, and optional attributes such as a comment, path and domain qualifiers, a maximum age, and a version number.

The getCookies() method of the request object returns an array of Cookie objects. Cookies can be constructed using the following code:

<% @ page language="java" import="java.util. *"%> <%

String username=request.getParameter("username"); if(username==null) username="";

Date now = new Date();

String timestamp = now.toString();

Cookie cookie = new Cookie ("username",username); cookie.setMaxAge(365 * 24 * 60 * 60);

response.addCookie(cookie);

%>

<html> <head>

<title>Cookie Saved</title>

</head> <body>

<p>Next Page to view the cookie value<p>

</body>

Above code sets the cookie and then displays a link to view cookie page

7.2 RMI

- RMI applications are often comprised of two separate programs: a server and a client.
- A typical server application creates a number of remote objects, makes references to those remote objects accessible, and waits for clients to invoke methods on those remote objects.
- A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them. RMI provides the mecahnism by which the server and the client communicate and pass information back and forth. Such an applications is sometimes referred to as a distributed object application.

The java.rmi.Remote Interface :

In RMI, a remote interface is an interface that declares a set of methods that may be invoked from a remote Java virtual machine. A remote interface must satisfy the following requirements:

• A remote interface must at least extend, either directly or indirectly, the interface java.rmi.Remote.

• Each method declaration in a remote interface must satisfy the requirements

Server side and Client side

The interface ServerRef represents the server-side handle for a remote object implementation.

package java.rmi.server;

public interface ServerRef extends RemoteRef {

RemoteStub exportObject(java.rmi.Remote obj, Object data) throws java.rmi.RemoteException;

String getClientHost() throws ServerNotActiveException; }

- The method exportObject finds or creates a client stub object for the

  supplied Remote object implementation *obj.*

- The parameter *data* contains information necessary to export the object (such as port number).

- The method getClientHost returns the host name of the current client.

  When called from a thread actively handling a remote method invocation, the host name of the client invoking the call is returned.

- If a remote method call is not currently being service, then ServerNotActiveException is called.

There is no special configuration necessary to enable the client to send RMI calls through a firewall. The client can, however, disable the packaging of RMI calls as HTTP requests by setting the java.rmi.server.disableHttp property to equal the boolean value true.

UNIT – 8: ENTERPRISE JAVA BEANS

Enterprise java Beans

Deployment Descriptors

Session Java Bean

Entity Java Bean

Message-Driven Bean

The JAR File.

Definition : Enterprise JavaBeans™ (EJB) is a managed, server-side component architecture for modular construction of enterprise applications.

Enterprise JavaBeans (**EJB**) is a managed, server-side component architecture for modular construction of enterprise applications. The EJB specification is one of several Java APIs in the Java EE specification. EJB is a server-side model that encapsulates the business logic of an application. The EJB specification was originally developed in 1997 by IBM and later adopted by Sun Microsystems (EJB 1.0 and 1.1) in 1999[1] and enhanced under the Java Community Process as JSR 19 (EJB 2.0), JSR 153(EJB 2.1), JSR 220 (EJB 3.0) and JSR 318 (EJB 3.1).

The EJB specification intends to provide a standard way to implement the back-end 'business' code typically found in enterprise applications (as opposed to 'front-end' interface code). Such code addresses the same types of problems, and solutions to these problems are often repeatedly re-implemented by programmers. Enterprise JavaBeans are intended to handle such common concerns as persistence, transactional integrity, and security in a standard way, leaving programmers free to concentrate on the particular problem at hand.

In a typical J2EE application, Enterprise JavaBeans (EJBs) contain the application's business logic and live business data. Although it is possible to use standard Java objects to contain your business logic and business data, using EJBs addresses many of the issues you would find by using simple Java objects, such as scalability, lifecycle management, and state management.

There are three different types of EJB that are suited to different purposes:

- **Session EJB**—A Session EJB is useful for mapping business process flow (or equivalent application concepts). There are two sub-types of Session EJB — stateless and stateful represent "pure" functionality that is created as it is needed.
- **Entity EJB**—An Entity EJB maps a combination of data (or equivalent application concept) and associated functionality. Entity EJBs are usually based on an underlying data store and will be created based on that data within it.
- **Message-driven EJB**—A Message-driven EJB is very similar in concept to a Session EJB, but is only activated when an asynchronous message arrives.

1. Deployment Descriptors

Definition : A *deployment descriptor* is a file that defines the following kinds of information: **EJB structural information,** such as the EJB name, class, home and remote interfaces, bean type (session or entity), environment entries, resource factory references, EJB references, security role references, as well as additional information based on the bean type. **Application assembly information,** such as EJB references, security roles, security role references, method permissions, and container transaction attributes. Specifying assembly descriptor information is an optional task that an Application Assembler performs.

1.1 Session Bean

- Session bean is a type of enterprise bean; a type of EJB server-side component.
  - Session bean components implement the javax.ejb.SessionBean interface and can be stateless or stateful.
  - Stateless session beans are components that perform transient services; stateful session beans are components that are dedicated to one client and act as a server-side extension of that client.

  Session beans can act as agents modeling workflow or provide access to special transient business services. As an agent, a stateful session bean might represent a customer's session at an online shopping site.

- As a transitive service, a stateless session bean might provide access to validate and process credit card orders.
- Session beans do not normally represent persistent business concepts like Employee or Order. This is the domain of a different component type called an entity bean.

Example : package ejb.demo;

```
import javax.ejb.*;
import java.rmi.Remote;
import java.rmi.RemoteException; import java.util. *;
/* *
* This interface is extremely simple it declares only * one create method.
*/
public interface DemoHome extends EJBHome {
public Demo create() throws CreateException, RemoteException;
}
```

Entity Java Bean

- An *entity bean* represents a business object in a persistent storage mechanism.
- Some examples of business objects are customers, orders, and products.
- In the J2EE SDK, the persistent storage mechanism is a relational database.
- Typically, each entity bean has an underlying table in a relational database, and each instance of the bean corresponds to a row in that table.

Message-Driven Bean

- A *message-driven bean* is an enterprise bean that allows J2EE applications to process messages asynchronously.
- It acts as a JMS message listener, which is similar to an event listener except that it receives messages instead of events.
- The messages may be sent by any J2EE component--an application client, another enterprise bean, or a Web component--or by a JMS application or system that does not use J2EE technology.

Message-driven beans currently process only JMS messages, but in the future they may be used to process other kinds of messages

The JAR file

The Java Archive (JAR) file format enables you to bundle multiple files into a single archive file. Typically a JAR file contains the class files and auxiliary resources associated with applets and applications.

The JAR file format provides many benefits:

- *Security:* You can digitally sign the contents of a JAR file. Users who recognize your signature can then optionally grant your software security privileges it wouldn't otherwise have.
- *Decreased download time:* If your applet is bundled in a JAR file, the applet's class files and associated resources can be downloaded to a browser in a single HTTP transaction without the need for opening a new connection for each file.
- *Compression:* The JAR format allows you to compress your files for efficient storage.
- *Packaging for extensions:* The extensions framework provides a means by which you can add functionality to the Java core platform, and the JAR file format defines the packaging for extensions. By using the JAR file format, you can turn your software into extensions as well.
- *Package Sealing:* Packages stored in JAR files can be optionally sealed so that the package can enforce version consistency. Sealing a package within a JAR file means that all classes defined in that package must be found in the same JAR file.
- *Package Versioning:* A JAR file can hold data about the files it contains, such as vendor and version information.
- *Portability:* The mechanism for handling JAR files is a standard part of the Java platform's core API.

Execution

EJBs are deployed in an EJB container, typically but not necessarily, within an application server. The specification describes how an EJB interacts with its container and how client code interacts with the container/EJB combination. The EJB classes used by applications are included in the javax.ejb package. (The javax.ejb.spi package is a service provider interfaceused only by EJB container implementations.)

Clients of EJB beans do not instantiate those beans directly via Java's new operator, but instead have to obtain a reference via the EJB container. This reference is then not a reference to the implementation bean itself, but to a proxy, which either dynamically implements the local or remote business interface that the client requested or dynamically implements a sub-type of the actual bean. The proxy can then be directly cast to the interface or bean. A client is said to have a 'view' on the EJB, and the local interface, remote interface and bean type itself respectively correspond with the local view, remote view and no-interface view.

This proxy is needed in order to give the EJB container the opportunity to transparently provide cross-cutting (AOP-like) services to a bean like transactions, security, interceptions, injections, remoting, etc.

E.g. a client invokes a method on a proxy, which will then first start a transaction with the help of the EJB container and then call the actual bean method. When the actual bean method returns, the proxy ends the transaction (i.e. by committing it or doing a rollback) and transfers control back to the client.

Transactions

EJB containers must support both container managed ACID transactions and bean managed transactions.

Container-managed transactions (CMT) are by default active for calls to session beans. That is, no explicit configuration is needed. This behavior may be declaratively tuned by the bean via annotations and if needed such configuration can later be overridden in the deployment descriptor. Tuning includes switching off transactions for the whole bean or specific methods, or requesting alternative strategies for transaction propagation and starting or joining a transaction. Such strategies mainly deal with what should happen if a transaction is or isn't already in progress at the time the bean is called. The following variations are supported:

Declarative Transactions Management Types

Type	Explanation
MANDATORY	If the client has not started a transaction, an exception is thrown. Otherwise the client's transaction is used.
REQUIRED	If the client has started a transaction, it is used. Otherwise a new transaction is started. (this is the default when no explicit type has been specified)

REQUIRES_NEW	If the client has started a transaction, it is suspended. A new transaction is always started.
SUPPORTS	If the client has started a transaction, it is used. Otherwise, no transaction is used.
NOT_SUPPORTED	If the client has started a transaction, it is suspended. No new transaction is started.
NEVER	If the client has started a transaction, an exception is thrown. No new transaction is started.

Alternatively, the bean can also declare via an annotation that it wants to handle transactions programmatically via the JTA API. This mode of operation is called Bean Managed Transactions (BMT), since the bean itself handles the transaction instead of the container.

Events

JMS (Java Message Service) is used to send messages from beans to clients, to let clients receive asynchronous messages from these beans. MDBs can be used to receive messages from clients asynchronously using either a **JMS** Queue or a Topic.

Naming and directory services

As an alternative to injection, clients of an EJB can obtain a reference to the session bean's proxy object (the EJB stub) using JNDI. This alternative can be used in cases where injection is not available, such as non-managed beans or standalone remote Java SE clients, or when it's necessary to programmatically determine which bean to obtain.

JNDI names for EJB session beans are assigned by the EJB container via the following scheme:

JNDI names

Scope	Name pattern
Global	java:global[/<app-name>]/<module-name>/<bean-name>[!<fully-qualified-interface-name>]
Application	java:app/<module-name>/<bean-name>[!<fully-qualified-interface-name>]
Module	java:module/<bean-name>[!<fully-qualified-interface-name>]

(entries in square brackets denote optional parts)

A single bean can be obtained by any name matching the above patterns, depending on the 'location' of the client. Clients in the same module as the required bean can use the module scope

and larger scopes, clients in the same application as the required bean can use the app scope and higher, etc.