Paper Code: BCA 206

Paper: Java Programming

UNIT-I

Java Programming: Introduction, Data types, access specifiers, operators, control statements, arrays.Classes: Fundamentals, objects, methods, constructors.Inheritance: Super class, sub class, this and super operator, method overriding, use of final, packages, abstract class, interface. Polymorphism: Method overloading, constructor overloading.

UNIT – II

Exception Handling: Exception Class, built in checked and unchecked exceptions, user defined exceptions, use of try, catch, throw, throws, finally.Multi threaded programming: Overview, comparison with multiprocessing ,Thread class and runnable interface, life cycle, creation of single and multiple threads, thread priorities, overview of Synchronization. Java Library: String handling (only main functions), String Buffer class. Elementary concepts of Input/output: byte and character streams, System. in and Sysem.out, print and println, reading from a file and writing in a file.

UNIT – III

Software Development using Java:

Applets: Introduction, Life cycle, creation and implementation, AWT controls: Button, Label, TextField, TextArea, Choice lists, list, scrollbars, check boxes, Layout managers,

Elementary concepts of Event Handling: Delegation Event Model, Event classes and listeners, Adapter classes, Inner classes. Swings: Introduction and comparison with AWT controls.

UNIT – IV

Networking Basics: Socket (datagram and TCP/IP based client and server socket), factory Methods, InetAddress JDBC: JDBC Architecture, JDBC Drivers, Connecting to the Database Introduction to Java Servlets: Life cycle, Interfaces and classes in javax.servlet package (only description) creating a simple servlet

UNIT-I

Java Programming: Introduction

Object Oriented Programming (OOP) is a programming paradigm that uses "objects" and their interactions to design applications and computer programs. It is based on several techniques, including encapsulation, modularity, polymorphism, and inheritance.

Inheritance „Subclasses" are more specialized versions of a class, which inherit attributes and behaviors from their parent classes, and can introduce their own. Inheritance is inheriting a class into another class.

Encapsulation: Encapsulation conceals the functional details of a class from objects that send messages to it. This is hiding the internal details.

Abstraction: Abstraction is simplifying complex reality by modeling classes appropriate to the problem, and working at the most appropriate level of inheritance for a given aspect of the problem. Building class is an abstraction process.

Polymorphism: Multiple objects exhibiting similar features in different ways is known as polymorphism. Polymorphism is the process of using an operator or function in different ways for different set of inputs given.

Data types:

The Java programming language is strongly-typed, which means that all variables must first be declared before they can be used. This involves stating the variable's type and name, as you've already seen :

int gear = 1;

Doing so tells your program that a field named "gear" exists, holds numerical data, and has an initial value of "1". A variable's data type determines the values it may contain, plus the operations that may be performed on it. In addition to int, the Java programming language supports seven other primitive data types. A primitive type is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values. The eight primitive data types supported by the Java programming language are :

byte : The byte data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive).

short : The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with

int : The int data type is a 32-bit signed two's complement integer. It has a minimum value of -2,147,483,648 and a maximum value of 2,147,483,647 (inclusive). For integral values, this data type is generally the default choice unless there is a reason (like the above) to choose something else

long : The long data type is a 64-bit signed two's complement integer. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive). Use this data type when you need a range of values wider than those provided by int.

float : The float data type is a single-precision 32-bit IEEE 754 floating point As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency.

double : The double data type is a double-precision 64-bit IEEE 754 floating point. For decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.

Boolean : The Boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false

conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.

char : The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).

In addition to the eight primitive data types listed above, the Java programming language also provides special support for character strings via the java.lang.String class.

Access specifiers :

One of the techniques in object-oriented programming is encapsulation. It concerns the hiding of data in a class and making this class available only through methods. In this way the chance of making accidental mistakes in changing values is minimized. Java allows you to control access to classes, methods, and fields via so-called access specifiers.

Java offers four access specifiers, listed below in decreasing accessibility:

public

protected

default (no specifier)

private

public: public classes, methods, and fields can be accessed from everywhere. The only constraint is that a file with Java source code can only contain one public class whose name must also match with the filename. If it exists, this public class represents the application or the applet, in which case the public keyword is necessary to enable your Web browser or appletviewer to show the applet. You use public classes, methods, or fields only if you explicitly want to offer access to these entities and if this access cannot do any harm. An example of a square determined by the position of its upper-left corner and its size:

```
public class Square {  // public class

  public x, y, size;   // public instance variables

}
```

Protected: protected methods and fields can only be accessed within the same class to which the methods and fields belong, within its subclasses, and within classes of the same package, but not from anywhere else. You use the protected access level when it is appropriate for a class's subclasses to have access to the method or field, but not for unrelated classes.

default (no specifier): If you do not set access to specific level, then such a class, method, or field will be accessible from inside the same package to which the class, method, or field belongs, but not from outside this package. This access-level is convenient if you are creating packages. For example, a geometry package that contains Square and Tiling classes, may be easier and cleaner to implement if the coordinates of the upper-left corner of a Square are directly available to the Tiling class but not outside the geometry package.

private

private methods and fields can only be accessed within the same class to which the methods and fields belong. private methods and fields are not visible within subclasses and are not inherited by subclasses. So, the private access specifier is opposite to the public access specifier. It is mostly used for encapsulation: data are hidden within the class and accessor methods are provided. An example, in which the position of the upper-left corner of a square can be set or obtained by accessor methods, but individual coordinates are not accessible to the user.

```
public class Square {    // public class

  private double x, y   // private (encapsulated) instance variables


  public setCorner(int x, int y) {  // setting values of private fields

    this.x = x;

    this.y = y;

  }
```

```
 public getCorner() {  // setting values of private fields
   return Point(x, y);
 }
}
```

Operators:

Operators are special symbols that perform specific operations on one, two, or three operands, and then return a result.

The operators in the following table are listed according to precedence order Operators with higher precedence are evaluated before operators with relatively lower precedence. Operators on the same line have equal precedence. When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated from left to right; assignment operators are evaluated right to left.

Operator Precedence

Operators	Precedence
Postfix	expr++ expr--
Unary	++expr --expr +expr -xpr ~ !
Multiplicative	* / %
Additive	+ -
Shift	<< >> >>>
Relational	< > <= >= instanceof
Equality	== !=
Bitwise AND	&
Bitwise exclusive OR	^
Bitwise inclusive	|

OR

Logical AND	&&
Logical OR	||
Ternary	? :
Assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Control statements:

A Java program is a set of statements, which are normally executed sequentially in the order in which they appear. However, in practice, we have a number of situations, where we may have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specified conditions are met.

Conditional Constructs : Java language possesses decision making capabilities and supports the following statements known as control or decision making statements :

(1) If Statement : It allows the computer to evaluate the expression first and then, depending on whether the value of the expression is „true" or „false". The general form is : if (test expression)

The if statement may be implemented in different forms :

(a) Simple If Statement :

The general form is :

```
if (test expression)
{
statement-block;
}
statement-x;
```

(b) The If---Else Statement :

The general form is :

```
if (test expression)
```

```
{
true block statements;
}
else
{
false block statements;
}
statement-x;
```

(c) Nested If---Else Statement :

```
if (test condition1)
{
if(test condition2)
{
statement-1;
}
else
{
statement-2;
}
}
statement-x;
```

(d) Else If Ladder :

```
if (condition1)
statement-1;
else if (condition2)
statement-2;
else if (condition)
statement-n;
```

else

default-statement;

statement-x;

(2) The Switch Statement : When one of the many alternatives is to be selected, we can design a program using if statements to control the selection. However, when the number of alternatives increases, the program becomes difficult to read and follow. Then we can use switch statement in such situations.

The general form is :

switch(expression)

{

case value1:

block-1;

break;

case value-2:

block-2;

break;

……..

……..

default:

default-block;

break;

}

statement-x;

(3) The ?: Operator :

The general form is :

Conditional expression ? expression1:expression2;

Looping Constructs : The process of repeatedly executing a block of statements is known as looping. The statements in the block may be executed any number of times, from zero to infinite number.

(a) The While Statement : The simplest of all looping structures in Java is the while statement.

The general format is

Initialization;

while (test condition)

{

body of the loop

}

(b) The Do Statement : In this construct the body of the loop will execute first and then the test condition is evaluated.

Initialization;

do

{

body of the loop

}

while(test condition);

(c) For Statement : This is another entry-controlled loop like while loop. The general format is :

For(initialization; test condition; increament/decrement)

{

body of the loop

}

Jumps in Loops : Loops perform a set of operations repeatedly until the control variable fails to satisfy the test condition.Sometimes, it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs.

Jumping out of a loop---We can use the break statement which will immediately exit the control from the loop and the program continues with the statement immediately following the loop.

e.g. while(……)

{ ……

if(condition)

break;

…….

……

}

……

Skipping a Part of Loop : During the loop operation it may be necessary to skip a part of the body of the loop under certain conditions. We can use continue statement for this.

e.g. while(……..)

{……

if(……..)

continue;

……

}

The statements below continue statement are skipped and control jumps to header part of loop.

Arrays:

An array is a group of contiguous or related data items that share a common name. For instance, we can define an array name sal to represent a set of salaries of a group of employees. A particular value is indicated by writing a number called index number or subscript in brackets after the array name.

For example : sal[10] represents the salary of the 10 employeeS. While the complete set of values is referred to as an array, the individual values are called elements.

The arrays are of different types :

One Dimensional (A list of items can be given one variable name using only one subscript and such a variable is called a single-subsubscripted variable or a one-dimensional array.)

Two Dimensional (Where a table of values will be stored and two subscripts are required to represent an element of array)

Three Dimensional (Where many table of values will be stored and three subscripts are required to represent an element of an array)

Creating an Array : It involves three steps :

(1) Declaring the Array

(2) Creating Memory Locations

(3) Putting Values into the Memory Locations

Declaring an Array : Arrays in Java are declared in one of the two forms :

Form 1 type arrayname[];

Form 2 type[]arrayname;

e.g. int number[];

int[] counter;

Creating Memory Allocations : To create an array we use new operator :

arrayname =new type[size];

e.g. number =new int[5];

Initialization of Array : The final step is to put values into the array created.

arrayname[subscript]=value;

e.g. number[0]=35;

…….

……...

number[4]=55;

Example of Two Dimensional Array :

int a[][]=new int[3][3] ; creates a table that can store 9 integer values.

Or

int a[2][3]={0,0,0,1,1,1}; will initializes the elements of the first row to zero and the second row to one .

Classes: Fundamentals

In the real world, you'll often find many individual objects all of the same kind.

All the objects that have similar properties and similar behavior are grouped together to form a class.

In other words we can say that a class is a user defined data type and objects are the instance variables of class.

There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an instance of the class of objects known as bicycles. A class is the blueprint from which individual objects are created.

The following Bicycle class is one possible implementation of a bicycle :

```
class Bicycle {

int cadence = 0;

int speed = 0;

int gear = 1;

void changeCadence(int newValue) {

cadence = newValue;

}

void changeGear(int newValue) {

gear = newValue;

}

void printStates() {

System.out.println("cadence:"+cadence+"speed:"+speed+" gear:"+gear);

}

}
```

The fields cadence, speed, and gear represent the object's state, and the methods (changeCadence, changeGear, speedUp etc.) define its interaction with the outside world.

You may have noticed that the Bicycle class does not contain a main method. That's because it's not a complete application; it's just the blueprint for bicycles that might be used in an application.

Here's a BicycleDemo class that creates two separate Bicycle objects and invokes their methods :

```
class BicycleDemo {

public static void main(String[] args) { / Create two different Bicycle objects

Bicycle bike1 = new Bicycle();

Bicycle bike2 = new Bicycle();

// Invoke methods on those objects
```

```
bike1.changeCadence(50);

bike1.changeGear(2);

bike1.printStates();

bike2.changeCadence(50);

bike2.changeGear(2);

bike2.changeCadence(40);

bike2.changeGear(3);

bike2.printStates();

}

}
```

The output of this test prints the ending pedal cadence, speed, and gear for the two bicycles :

Cadence : 50 speed : 10 gear : 2

Cadence : 40 speed : 20 gear : 3

Objects:

Objects are key to understand object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle. So, anything that exists in real world is an object. In other words an object is a real life entity.

Real-world objects share two characteristics: They all have identity, state and behavior. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

Real-world objects vary in complexity. e.g. your desktop lamp may have only two possible states (on and off) and two possible behaviors (turn on, turn off), but your desktop radio might have additional states (on, off, current volume, current station) and behavior (turn on, turn off, increase volume, decrease volume, seek, scan, and tune). Some objects, in turn, will also contain other objects. These real-world observations all translate into the world of object-oriented programming.

A software Object.

Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in fields (variables in some programming languages) and exposes its behavior through methods (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as data encapsulation — a fundamental principle of object-oriented programming.

Consider a bicycle, for example:

A Bicycle Modeled as a Software Object

By attributing state (current speed, current pedal cadence, and current gear) and providing methods for changing that state, the object remains in control of how the outside world is allowed to use it. For example, if the bicycle only has 6 gears, a method to change gears could reject any value that is less than 1 or greater than 6.

Methods:

A Java method is a collection of statements that are grouped together to perform an operation. When you call the System. out. println method, for example, the system actually executes several statements in order to display a message on the console.

Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, overload methods using the same names, and apply method abstraction in the program design.

Creating a Method:

In general, a method has the following syntax:

```
modifier return Value Type method Name(list of parameters) {
  // Method body;
}
```

Constructors:

Constructor declarations look like method declarations—except that they use the name of the class and have no return type. For example, Bicycle has one constructor :

```
public Bicycle(int startCadence, int startSpeed, int startGear) {
```

gear = startGear;

cadence = startCadence;

speed = startSpeed;

}

To create a new Bicycle object called myBike, a constructor is called by the new operator :

Bicycle myBike = new Bicycle(30, 0, 8);

new Bicycle(30, 0, 8) creates space in memory for the object and initializes its fields.

Although Bicycle only has one constructor, it could have others, including a no-argument constructor :

public Bicycle() {

gear = 1;

cadence = 10;

speed = 0;

}

Bicycle yourBike = new Bicycle(); invokes the no-argument constructor to create a new Bicycle object called yourBike.

Both constructors could have been declared in Bicycle because they have different argument lists. As with methods, the Java platform differentiates constructors on the basis of the number of arguments in the list and their types. You cannot write two constructors that have the same number and type of arguments for the same class, because the platform would not be able to tell them apart. Doing so causes a compile-time error.

You can use access modifiers in a constructor's declaration to control which other classes can call the constructor.

Inheritance: Super class, sub class

In the Java language, classes can be derived from other classes, thereby inheriting fields and methods from those classes.

Definitions : A class that is derived from another class is called a **subclass** (also a derived class, extended class, or child class). The class from which the subclass is derived is called a **super class** (also a base class or a parent class).

Excepting Object, which has no super class, every class has one and only one direct super class (single inheritance). In the absence of any other explicit super class, every class is implicitly a subclass of Object.

Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, Object. Such a class is said to be descended from all the classes in the inheritance chain stretching back to Object.

The idea of inheritance is simple but powerful: When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.

A subclass inherits all the members (fields, methods, and nested classes) from its super class. Constructors are not members, so they are not inherited by subclasses, but the constructor of the super class can be invoked from the subclass.

The Java Platform Class Hierarchy : The Object class, defined in the java.lang package, defines and implements behavior common to all classes—including the ones that you write. In the Java platform, many classes derive directly from Object, other classes derive from some of those classes, and so on, forming a hierarchy of classes.

All Classes in the Java Platform are Descendants of Object

At the top of the hierarchy, Object is the most general of all classes. Classes near the bottom of the hierarchy provide more specialized behavior.

```
/*An Example of Inheritance*/

public class Bicycle {

// the Bicycle class has three fields

public int cadence;

public int gear;

public int speed;

// the Bicycle class has one constructor

public Bicycle(int startCadence, int startSpeed, int startGear) {

gear = startGear;

cadence = startCadence;

speed = startSpeed;

}

// the Bicycle class has four methods
```

```
public void setCadence(int newValue) {
cadence = newValue;
}
public void setGear(int newValue) {
gear = newValue;
}
public void applyBrake(int decrement) {
speed -= decrement;
}
public void speedUp(int increment) {
speed += increment;
}
}
```

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this :

```
public class MountainBike extends Bicycle {
// the MountainBike subclass adds one field
public int seatHeight;
// the MountainBike subclass has one constructor
public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) {
super(startCadence, startSpeed, startGear);
seatHeight = startHeight;
}
// the MountainBike subclass adds one method
public void setHeight(int newValue) {
seatHeight = newValue;
}
}
```

MountainBike inherits all the fields and methods of Bicycle and adds the field seatHeight and a method to set it.

This and super operator:

Within an instance method or a constructor, this is a reference to the current object — the object whose method or constructor is being called. You can refer to any member of the current object from within an instance method or a constructor by using „this'.

Using 'this' with a Field : The most common reason for using the this keyword is because a field is shadowed by a method or constructor parameter.

For example, the Point class was written like this

```
public class Point {

public int x = 0;

public int y = 0;

//constructor

public Point(int a, int b) {

x = a;

y = b;

}

}
```

but it could have been written like this :

```
public class Point {

public int x = 0;

public int y = 0; //constructor

public Point(int x, int y) {

this.x = x;

this.y = y;

}

}
```

Each argument to the second constructor shadows one of the object's fields—inside the constructor x is a local copy of the constructor's first argument. To refer to the Point field x, the constructor must use this.x.

Method overriding:

Method Overriding : Method inheritance enables us to define and use methods repeatedly subclasses without having to define the methods again in subclass.

However, there may be occasions when we want an object to respond to the same method but have different behaviour when that method is called. That means , we should override the method defined in the superclass. This is possible by defining a method in the subclass that has the same name, same arguments and same return type as a method in the superclass. This is known as overriding.

e.g. /* An example of method overriding*/

```
class Super
{
int x;
Super(int x)
{
this.x=x;
}
void disp()
{
System.out.println("Super x="+x);
}
}
class Sub extends Super
{
int y;
Sub(int x, int y)
{ super(x);
```

```
this.y=y;
}
void disp()
{
System.out.println("Super x="+x);
System.out.println("Sub y="+y);
}
}
class OverrideTest
{
public static void main(String args[])
{
Sub s1=new Sub(100,200);
s1.display();
}
}
```

Use of final:

Writing Final Classes and Methods :

Final Methods : You can declare some or all of a class's methods final. You use the final keyword in a method declaration to indicate that the method cannot be overridden by subclasses. The Object class does this—a number of its methods are final.

You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object. For example, you might want to make the getFirstPlayer method in this ChessAlgorithm class final :

```
class ChessAlgorithm {
enum ChessPlayer { WHITE, BLACK }
...
final ChessPlayer getFirstPlayer() {
```

```
return ChessPlayer.WHITE;

}

...

}
```

Methods called from constructors should generally be declared final. If a constructor calls a non-final method, a subclass may redefine that method with surprising or undesirable results.

Final Variables : To prevent the subclasses from overrding the member variables of the superclass, we can declare them as final using the final as a modifier.

e.g. final int SIZE =55;

Final Classes : You can also declare an entire class final — this prevents the class from being subclassed. This is particularly useful, for example, when creating an immutable class like the String class. You can use final modifier with class as follows:

```
e.g. final class A

{

}
```

Packages:

Creating and Using Packages : To make types easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of related types into packages.

Definition: A package is a grouping of related types providing access protection and name space management. Note that types refers to classes, interfaces, enumerations, and annotation types.

The fundamental classes are in java.lang. Classes for reading and writing (input and output) are in java.io, and so on. You can put your types in packages too.

Creating a Package : To create a package, you choose a name for the and put a package statement with that name at the top of every source file that contains the types (classes, interfaces, enumerations, and annotation types) that you want to include in the package.

The package statement (for example, package graphics;) must be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

Note : If you put multiple types in a single source file, only one can be public, and it must have the same name as the source file.

e.g.

If you put the graphics interface and classes in a package called graphics, you would need to be written in source files, like this :

```
//in the Draggable.java file

package graphics;

public interface Draggable {

. . .

}

//in the Graphic.java file

package graphics;

public abstract class Graphic {

. . .

}

//in the Circle.java file

package graphics;

public class Circle extends Graphic implements Draggable {

. . .

}
```

If you do not use a package statement, your type ends up in an unnamed package. Generally speaking, an unnamed package is only for small or temporary applications or when you are just beginning the development process. Otherwise, classes and interfaces belong in named packages.

Abstract class:

Abstract Methods and Classes : An abstract class is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.

An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this :

abstract void moveTo(double deltaX, double deltaY);

If a class includes abstract methods, the class itself must be declared abstract, as in :

public abstract class GraphicObject {

// declare fields

// declare non-abstract methods

abstract void draw();

}

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, the subclass must also be declared abstract.

Note : All of the methods in an interface are implicitly abstract, so the abstract modifier is not used with interface methods (it could be—it's just not necessary).

Abstract Classes versus Interfaces : Unlike interfaces, abstract classes can contain fields that are not static and final, and they can contain implemented methods. Such abstract classes are similar to interfaces, except that they provide a partial implementation, leaving it to subclasses to complete the implementation. If an abstract class contains only abstract method declarations, it should be declared as an interface instead.

Multiple interfaces can be implemented by classes anywhere in the class hierarchy, whether or not they are related to one another in any way. But in Java a single class can be inherited whether the class being extended is abstract or not.

By comparison, abstract classes are most commonly subclassed to share pieces of implementation. A single abstract class is subclassed by similar classes that have a lot in common (the implemented parts of the abstract class), but also have some differences (the abstract methods).

e.g.

abstract class GraphicObject {

int x, y;

...

void moveTo(int newX, int newY) {

...

}

```
abstract void draw();

abstract void resize();

}
```

Each non-abstract subclass of GraphicObject, such as Circle and Rectangle, must provide implementations for the draw and resize methods :

```
class Circle extends GraphicObject {

void draw() {

...

}

void resize() {

...

}

}

class Rectangle extends GraphicObject {

void draw() {

...

}

void resize() {

...

}

}
```

Interface:

Defining an Interface : An interface declaration consists of modifiers, the keyword interface, the interface name, a comma-separated list of parent interfaces (if any), and the interface body. For example:

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {

// constant declarations
```

```
double E = 2.718282; // base of natural logarithms

// method signatures

void doSomething (int i, double x);

int doSomethingElse(String s);

}
```

The public access specifier indicates that the interface can be used by any class in any package. If you do not specify that the interface is public, your interface will be accessible only to classes defined in the same package as the interface.

An interface can extend other interfaces, just as a class can extend or subclass another class. However, whereas a class can extend only one other class, an interface can extend any number of interfaces. The interface declaration includes a comma-separated list of all the interfaces that it extends.

The Interface Body : The interface body contains method declarations for all the methods included in the interface. A method declaration within an interface is followed by a semicolon, but no braces, because an interface does not provide implementations for the methods declared within it. All methods declared in an interface are implicitly public, so the public modifier can be omitted.

An interface can contain constant declarations in addition to method declarations. All constant values defined in an interface are implicitly public, static, and final. Once again, these modifiers can be omitted.

Implementing an Interface : To declare a class that implements an interface, you include an implements clause in the class declaration. Your class can implement more than one interface, so the implements keyword is followed by a comma-separated list of the interfaces implemented by the class.

```
public interface Relatable

{

public int isLargerThan(Relatable other);

}

public class RectanglePlus implements Relatable {

public int width = 0;

public int height = 0;

public Point origin;

// four constructors
```

```
public RectanglePlus() {
origin = new Point(0, 0);
}
public RectanglePlus(Point p) {
origin = p;
}
public RectanglePlus(int w, int h) {
origin = new Point(0, 0);
width = w;
height = h;
}
public RectanglePlus(Point p, int w, int h) {
origin = p;
width = w;
height = h;
}
// a method for moving the rectangle
public void move(int x, int y) {
origin.x = x;
origin.y = y;
}
```

```
// a method for computing the area of the rectangle
public int getArea() {
return width * height;
}
// a method to implement Relatable
public int isLargerThan(Relatable other) {
RectanglePlus otherRect = (RectanglePlus)other;
if (this.getArea() < otherRect.getArea())
return -1;
else if (this.getArea() > otherRect.getArea())
return 1;
else
return 0;
}
}
```

Because Rectangle Plus implements Relatable, the size of any two RectanglePlus objects can be compared.

Polymorphism: Method overloading, constructor overloading

Method Overloading : In Java, it is possible to create methods that have the same name, but different parameter lists and different definitions. This is called method overloading. Method overloading is used when objects are required to perform similar tasks but using different input parameters. When we call a method in an object, Java matches up the method name first and then the number and type of parameters to decide which one of the definitions to execute. This process is known as polymorphism.

To create an overloaded method , all we have to do is to provide several different method definitions in the class, all with the same name, but with different parameter lists.

e.g. **/* An example of constructor overloading*/**

```
class Room
{
float len;
float wid;
Room (float a, float b)
{
len=a;
wid=b;
}
Room(float x)
{
len=wid=x;
}
int area()
{
return (len*wid);
}
}
```

Exception

The term *exception* is shorthand for the phrase "exceptional event."

Definition: An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.

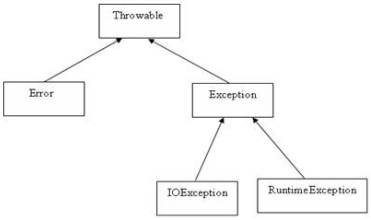
Common Exceptions:

In java it is possible to define two categories of Exceptions and Errors.

- JVM Exceptions: - These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples : NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException,
- Programmatic exceptions. These exceptions are thrown explicitly by the application or the API programmers Examples: IllegalArgumentException, IllegalStateException

An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following:

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications, or the JVM has run out of memory.



Checked exceptions: A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.

Advantages of Exceptions:

Advantage 1: Separating Error-Handling Code from "Regular" Code
Advantage 2: Propagating Errors up the Call Stack
Advantage 3: Grouping and Differentiating Error Types

Exception class has two main subclasses:

1. IOException class

2. RuntimeException Class

IOException class: IO stands for input/output and has to do with sending and receiving data. IO is a security problem and should not be used

Runtime exceptions: A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation

The 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:

syntax:

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

The catch Blocks

The catch block contains code that is executed if and when the exception handler is invoked. The runtime system invokes the exception handler when the handler is the first one in the call stack whose *Exception Type* matches the type of the exception thrown. The system considers it a match if the thrown object can legally be assigned to the exception handler's argument

syntax:

```
try {

} catch (Exception Type name) {

} catch (Exception Type name) {

}
```

The throws/throw Keywords:

If a method does not handle a checked exception, the method must declare it using the throws keyword. The throws keyword appears at the end of a method's signature.

syntax:

```
import java.io.*;
public class className
{
  public void deposit(double amount) throws RemoteException
  {
    // Method implementation
    throw new RemoteException();
  }
  //Remainder of class definition
}
```

The finally Keyword

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred.

syntax:

```
try
{
  //Protected code
}catch(ExceptionType1 e1)
{
  //Catch block
}catch(ExceptionType2 e2)
{
  //Catch block
}catch(ExceptionType3 e3)
{
  //Catch block
}finally
{
  //The finally block always executes.
}
```

Multiple catch Blocks:

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks

```
try
{
  //Protected code
}catch(ExceptionType1 e1)
```

```
{
  //Catch block
}catch(ExceptionType2 e2)
{
  //Catch block
}catch(ExceptionType3 e3)
{
  //Catch block
}
```

Remember the following:

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses when ever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

Multi threaded programming

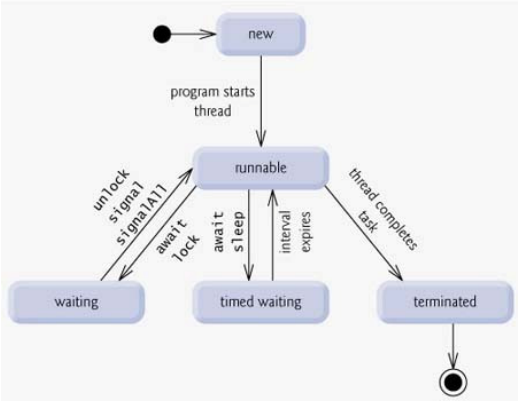
A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. A multithreading is a specialized form of multitasking. Multithreading requires less overhead than multitasking processing.

Using Multithreading:

1. The key to utilizing multithreading support effectively is to think concurrently rather than serially.
2. We can create very efficient programs.
3. We can create too many threads,
4. We can actually degrade the performance of your program rather than enhance it.

Life Cycle of a Thread:

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.



- New: A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- Runnable: After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- Waiting: Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- Timed waiting: A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transition back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- Terminated: A runnable thread enters the terminated state when it completes its task or otherwise terminates

Thread Priorities:

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java priorities are:

1. MIN_PRIORITY (a constant of 1)
2. MAX_PRIORITY (a constant of 10).
3. By default, every thread is NORM_PRIORITY (a constant of 5).

Overview of Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this synchronization is achieved is called *thread synchronization*. The synchronized keyword in Java creates a block

of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object. To enter a critical section, a thread needs to obtain the corresponding object's lock.

This is the general form of the synchronized statement:

```
Synchronized (object) {

  // statements to be synchronized

}
```

Unit-III

Software Development using Java:

Applet

An applet is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

Life Cycle of an Applet:

Four methods in the Applet class give you the framework on which you build any serious applet:

- init: This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
- start: This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
- stop: This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- destroy: This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.
- paint: Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

A "Hello, World" Applet:

The following is a simple applet named HelloWorldApplet.java:

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

public class HelloWorldApplet extends Applet
{
  public void paint (Graphics g)
  {
    g.drawString ("Hello World", 25, 50);
  }
}
```

Adding components to applets

Applets can be used as a container for visual components, such as those provided by the *java.awt* package. Components are created, and then added to the applet container. The positioning of applet components is controlled by a layout manager.

Layout managers

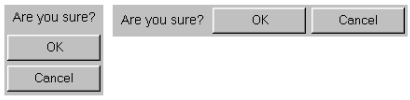
Components are "laid out" within a Container is described by the LayoutManager class. Container is described by the LayoutManager class. Since the LayoutManager class is abstract, we cannot use it directly. We must sub-class of it and provide its own functionality or use a *derived class of LayoutManager* (i.e BorderLayout, CardLayout, GridLayout, etc) already created for you.

There are many different layout schemes, but the ones pre-defined for us are

- BorderLayout This scheme lays out the Component in 5 ways
    1. North - Northern part of the Container
    2. South - Southern part of the Container
    3. East - Eastern part of the Container
    4. West - Western part of the Container
    5. Center - Centered in the Container
- CardLayout - Allows for what Windows programmers have called for years "tabbed dialogs" or dynamic dialogs (now available on all versions of Netscape).
- GridLayout - Allows for the layout of Components in a grid -like fashion rather than "North" or "Center".
- GridBagLayout - HTMLTable-ish style of layout
- FlowLayout - Allows for Component to be laid out in a row(or flow) and aligned(left, right, center).
- None - No layout, the Container will not attempt to reposition the Components during a update.

```
// Set layout manager to flow layout
setLayout(new FlowLayout());

// Add three components to applet
add(new Label("Are you sure?"));
add(new Button("Ok"));
add(new Button("Cancel"));
```



Example of FlowLayout, with different applet sizes

Delegation event model

Event model is based on the concept of an 'Event Source' and 'Event Listeners'. Any object that is interested in receiving messages (or events) is called an Event Listener. Any object that generates these messages (or events) is called an Event Source Event Delegation Model is based on four concepts:
The Event Classes

The Event Listeners

Explicit Event Enabling

Adapters

AWT Controls

There are a wide variety of AWT components. A selection of more common components from the java.awt package is presented below:

Label

Labels are great for providing user prompts, and for displaying information that should not be modified by users. If you want to have modifiable labels, consider using text fields instead. In almost all of the examples presented here, a status bar containing a label is used to convey information to the user.

Button

Buttons are useful when an application must recieve specific instructions from the user before proceeding. For example, when a user must enter data, and then wants to perform a task, a button can be used to instruct the applet / application to proceed when the data entry is complete.

Example source

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

// ButtonApplet

public class ButtonApplet extends Applet
{
	// Private member variables
	private Button m_button;
	private StatusBar m_status;

	public static final String state1 = "Click me";
	public static final String state2 = "Clicked";

	public void init()
	{
		// Panel creation
		Panel panel = new Panel();

		// Create the button
		m_button = new Button (state1);
		panel.add(m_button);

		// Create status bar
		m_status = new StatusBar ("Value : unclicked" );

		// Set colors
		setBackground( Color.white );
		setForeground( Color.black );
		panel.setBackground( Color.white );
		panel.setForeground( Color.black );
		m_status.setBackground( Color.white );
		m_status.setForeground( Color.black );
		m_button.setBackground( Color.blue );
		m_button.setForeground( Color.white );
```

```
                // Set applet layout
                setLayout ( new BorderLayout() );
                add(panel, "North");
                add(m_status, "South");

                layout();
        }

        public boolean action (Event evt, Object what)
        {
                if ( ( evt.target == m_button ) && ( evt.id == 1001 ) )
                {
                        // State 1?
                        if (m_button.getLabel().compareTo(state1) == 0)
                        {
                                // Set state
                                m_button.setLabel(state2);
                                m_status.setStatus(state1);
                        }
                        else
                        // State 2
                        {
                                // Set state
                                m_button.setLabel(state1);
                                m_status.setStatus(state2);
                        }

                        return true;
                }
                else return false;
        }
}
```

Choice

Choice components take up very little screen space, and can present the user with a series of choice from which a single selection can be made.

Example source

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

//
//
```

```
// ChoiceApplet
//
//
public class ChoiceApplet extends Applet
{
        // Private member variables
        private Choice m_choice;
        private StatusBar m_status;

        public void init()
        {
                // Panel creation
                Panel panel = new Panel();

                // Create the choice
                m_choice = new Choice();
                panel.add(m_choice);

                // Populate choice with data
                m_choice.addItem ("Make a selection");
                m_choice.addItem ("Selection 1");
                m_choice.addItem ("Selection 2");
                m_choice.addItem ("Selection 3");

                // Create status bar
                m_status = new StatusBar ("Value : " );

                // Set colors
                setBackground( Color.white );
                setForeground( Color.black );
                panel.setBackground( Color.white );
                panel.setForeground( Color.black );
                m_status.setBackground( Color.white );
                m_status.setForeground( Color.black );

                // Set applet layout
                setLayout ( new BorderLayout() );
                add(panel, "North");
                add(m_status, "South");

                layout();
        }

        public boolean action (Event evt, Object what)
        {
                if ( evt.target == m_choice )
                {
```

```
                m_status.setStatus ( "Value : " + m_choice.getSelectedItem() );

                return true;
        }
```

List

List components fulfill much the same functionality as choice components. Items can be added, and the currently selected item can be returned. However, they take up more space than a choice component. This can be useful at times, however, as the items can be presented to the user, without he or she having to actually click on the component to see. Furthermore, multiple selections are allowable with the list component (though this feature must be enabled).

Example source

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

//
//
// ListApplet
//
//
public class ListApplet extends Applet
{
        // Private member variables
        private List m_list;
        private StatusBar m_status;

        public void init()
        {
                // Panel creation
                Panel panel = new Panel();

                // Create the list, five rows deep without multiple selections
                m_list = new List(5, false);
                panel.add(m_list);

                // Populate list with data
                m_list.addItem ("Selection 1");
                m_list.addItem ("Selection 2");
                m_list.addItem ("Selection 3");
                m_list.addItem ("Selection 4");
                m_list.addItem ("Selection 5");
```

```
                // Create status bar
                m_status = new StatusBar ("Value : " );

                // Set colors
                setBackground( Color.white );
                setForeground( Color.black );
                panel.setBackground( Color.white );
                panel.setForeground( Color.black );
                m_status.setBackground( Color.white );
                m_status.setForeground( Color.black );

                // Set applet layout
                setLayout ( new BorderLayout() );
                add(panel, "North");
                add(m_status, "South");

                layout();
        }

        public boolean action (Event evt, Object what)
        {
                if ( evt.target == m_list)
                {
                        m_status.setStatus ( "Value : " + m_list.getSelectedItem() );

                        return true;
                }
                else return false;
        }
}
```

TextField

Textfield components are useful for obtaining short pieces of information from users. They are limited to single line input (for multiple lines, use a TextArea instead). Whenever a user types data into a text field, and hits enter or changes focus, the textfield will generate an event.

Example source

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

//
//
// TextFieldApplet
```

```
//
//
public class TextFieldApplet extends Applet
{
        // Private member variables
        private TextField m_textfield;
        private StatusBar m_status;

        public void init()
        {
                // Panel creation
                Panel panel = new Panel();

                // Create the textfield
                m_textfield = new TextField(15);
                panel.add(m_textfield);

                // Create status bar
                m_status = new StatusBar ("Value : " );

                // Set colors
                setBackground( Color.white );
                setForeground( Color.black );
                panel.setBackground( Color.white );
                panel.setForeground( Color.black );
                m_status.setBackground( Color.white );
                m_status.setForeground( Color.black );

                // Set applet layout
                setLayout ( new BorderLayout() );
                add(panel, "North");
                add(m_status, "South");

                layout();
        }

        public boolean action (Event evt, Object what)
        {
                if ( evt.target == m_textfield )
                {
                        m_status.setStatus ( "Value : " + m_textfield.getText() );

                        return true;
                }
                else return false;
        }
}
```

TextArea

Textarea components are useful for getting a large amount of data from a user, or for displaying a large amount of data that is free to be modified. They take up more screen real estate than textfields, but can conserve quite a lot of room because they have support for scrollbars.

Example source

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

//
//
// TextAreaApplet
//
//
public class TextAreaApplet extends Applet
{
        // Private member variables
        private TextArea m_textarea;
        private StatusBar m_status;
        private Button m_calculate;

        public void init()
        {
                // Panel creation
                Panel panel = new Panel();

                // Create the textarea + button
                m_textarea = new TextArea(5, 20);
                panel.add(m_textarea);
                m_calculate = new Button ("Calculate");
                panel.add(m_calculate);

                // Create status bar
                m_status = new StatusBar ("Length : 0" );

                // Set colors
                setBackground( Color.white );
```

```
                setForeground( Color.black );
                panel.setBackground( Color.white );
                panel.setForeground( Color.black );
                m_status.setBackground( Color.white );
                m_status.setForeground( Color.black );

                // Set applet layout
                setLayout ( new BorderLayout() );
                add(panel, "North");
                add(m_status, "South");

                layout();
        }

        public boolean action (Event evt, Object what)
        {
                if ( evt.target == m_calculate )
                {
                        m_status.setStatus ( "Length : " + m_textarea.getText().length() );

                        return true;
                }
                else return false;
        }
}
```

UNIT-IV

Networking Basics: Socket (datagram and TCP/IP based client and server socket):

A network socket is an endpoint of an inter-process communication flow across a computer network.

on the Internet Protocol; therefore most network sockets are **Internet sockets**.

A **socket API** is an application programming interface (API), usually provided by the operating system, that allows application programs to control and use network sockets. Internet socket APIs are usually based on the Berkeley sockets standard.

A **socket address** is the combination of an IP address and a port number, much like one end of a telephone connection is the combination of a phone number and a particular extension. Based on this address, internet sockets deliver incoming data packets to the appropriate application process or thread.

An Internet socket is characterized by a unique combination of the following:

- Local socket address: Local IP address and port number
- Remote socket address: Only for established TCP sockets. this is necessary since a TCP server may serve several clients concurrently. The server creates one socket for each client, and these sockets share the same local socket address.
- Protocol: A transport protocol (e.g., TCP, UDP, raw IP, or others). TCP port 53 and UDP port 53 are consequently different, distinct sockets.

Socket types

There are several Internet socket types available:

- Datagram sockets, also known as connectionless sockets, which use User Datagram Protocol (UDP)
- Stream sockets, also known as connection-oriented sockets, which use Transmission Control Protocol (TCP) or Stream Control Transmission Protocol (SCTP).
- Raw sockets (or *Raw IP sockets*), typically available in routers and other network equipment. Here the transport layer is bypassed, and the packet headers are made accessible to the application.

Socket states and the client-server model

Computer processes that provide application services are called as servers, and create sockets on start up that are in *listening state*. These sockets are waiting for initiatives from client programs.

A TCP server may serve several clients concurrently, by creating a child process for each client and establishing a TCP connection between the child process and the client. Unique *dedicated sockets* are created for each connection. These are in *established* state, when a socket-to-socket virtual connection or virtual circuit (VC), also known as a TCP session, is established with the remote socket, providing a duplex byte stream.

A server may create several concurrently established TCP sockets with the same local port number and local IP address, each mapped to its own server-child process, serving its own client process. They are treated as different sockets by the operating system, since the remote socket address (the client IP address and/or port number) are different; i.e. since they have different socket pair tuples.

A UDP socket cannot be in an established state, since UDP is connectionless. Therefore, netstat does not show the state of a UDP socket. A UDP server does not create new child processes for every concurrently served client, but the same process handles incoming data packets from all remote clients sequentially through the same socket. It implies that UDP sockets are not identified by the remote address, but only by the local address, although each message has an associated remote address.

Factory methods:

Factory method is just a fancy name for a method that instantiates objects. Like a factory, the job of the factory method is to create -- or manufacture -- objects.

Let's consider an example.

Every program needs a way to report errors. Consider the following interface:

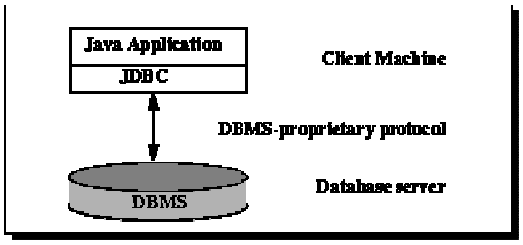
Listing 1

```
public interface Trace {
    // turn on and off debugging
    public void setDebug (boolean debug);
    // write out a debug message
    public void debug (String message);
    // write out an error message
    public void error (String message);
}
```

JDBC: JDBC Architecture

The JDBC API supports both two-tier and three-tier processing models for database access.

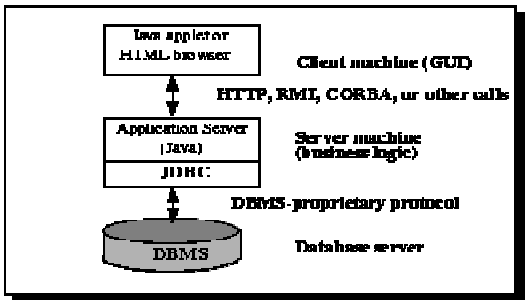
Two-tier Architecture for Data Access.



A Java application talks directly to the data source. This requires a JDBC driver that can communicate with the particular data source being accessed. A user's commands are delivered to the database or other data source, and the results of those statements are sent back to the user. The data source may be located on another machine to which the user is connected via a network. This is referred to as a client/server configuration,

In the three-tier model, commands are sent to a "middle tier" of services, which then sends the commands to the data source. The data source processes the commands and sends the results back to the middle tier, which then sends them to the user.

Three-tier Architecture for Data Access.

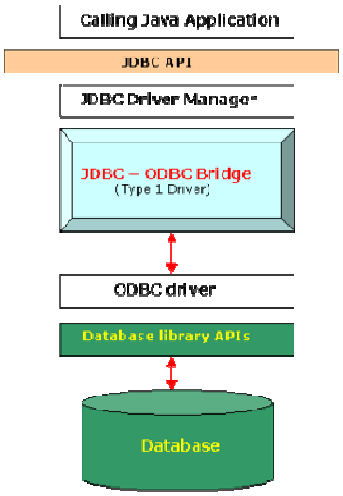


.

JDBC Drivers:

A **JDBC driver** is a software component enabling a Java application to interact with a database JDBC drivers are analogous to ODBC drivers, ADO.NET data providers, and OLE DB providers.

Type 1 Driver - JDBC-ODBC bridge



Schematic of the JDBC-ODBC Bridge

The JDBC type 1 driver, also known as the **JDBC-ODBC Bridge**, is a database driver implementation that employs the ODBC driver to connect to the database. The driver converts JDBC method calls into ODBC function calls.

Advantages

Almost any database, for which ODBC driver is installed, can be accessed.

Disadvantages

- Performance overhead since the calls have to go through the jdbc Overhead bridge to the ODBC driver, then to the native db connectivity interface (thus may be slower than other types of drivers).
- The ODBC driver needs to be installed on the client machine.
- Not suitable for applets, because the ODBC driver needs to be installed on the client.

Connecting to the Database:

First, you need to establish a connection with the data source you want to use. A data source can be a DBMS, a legacy file system, or some other source of data with a corresponding JDBC driver. Typically, a JDBC application connects to a target data source using one of two classes:

- DriverManager: This fully implemented class connects an application to a data source, which is specified by a database URL..
- DataSource: This interface is preferred over DriverManager because it allows details about the underlying data source to be transparent to your application.

Introduction to Java Servlets: Life cycle:

The lifecycle 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 Creating and Initializing a Servlet.
2. Invokes the service method, passing request and response objects. Service methods are discussed in Writing Service Methods.

Interfaces and classes in javax.servlet package (only description) Creating a simple servlet:

Package javax.servlet

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.

Package javax.servlet Description

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.

References:

Book Reference

- Patrick Naughton and Herbert Schildt, "Java-2 The Complete Reference", TMH.
- Krishnamoorthy R, Prabhu S ,"Internet and Java Programming", New Age Intl.

Web References:

1) *http://wiki.answers.com/Q/What_is_delegation_event_model_in_java#ixzz2XPErZ33e*
2) *http://www.boloji.com/index.cfm?md=Content&sd=Articles&ArticleID=560*
3) *http://www.tutorialspoint.com/servlets/servlets-first-example.htm*
4) *Cisco Networking Academy Program, CCNA 1 and 2 Companion Guide Revised Third Edition, P.480, ISBN 1-58713-150-1*
5) *www-306.ibm.com - AnyNet Guide to Sockets over SNA*
6) *books.google.com - UNIX Network Programming: The sockets networking API*
7) *books.google.com - Designing BSD Rootkits: An Introduction to Kernel Hacking*
8) *Wikipedia: Berkeley sockets*
9) *www.javaworld.com*
10) *www.docs.oracle.com*