Chapter 1: Java Basics

A) OOPS Fundamentals

Abstraction

Through the process of abstraction, a programmer hides all but the relevant data about an object in order to reduce complexity and increase efficiency. Abstraction means creating a named entity (a class) made of selected attributes (data members) and behaviors (methods) specific to a particular usage.

E.g. perform abstraction on a Rectangle:

The Rectangle becomes an entity now. Decide which properties of the Rectangle are important and relevant to the current programming context and discard the rest. Now decide what important behaviors the Rectangle can exhibit and include them, leaving the rest.

Entity: Rectangle
Attributes: length, breadth, area
Behaviors: findarea(), print()

Encapsulation

Encapsulation is the process of enveloping the abstracted attributes and behaviors in a “Class”. By performing encapsulation we can provide proper access control specifications to the attributes and behaviors. Attributes are even called as “data members” and the behaviors are called as “methods” or “member functions”. We should not allow the class members to be accessed directly by the external world as the data becomes unsafe. Thus a common practice is to keep the data members private and provide public getters and setters. Thus it is the class which provides data binding and security.

E.g. a sample Rectangle class having some data members and a behavior

```
class Rectangle{
int length,breadth,area;
void findArea(){
area = length*breadth;
}
void print(){
System.out.println(“Area is ”+area);
}
```

Inheritance

It is the property of deriving properties from a parent and creating a new child class. Often it happens that two classes have most of the things in common except a few specific attributes. So why to create two separate classes having the common as well as the specific data members? It results in data redundancy, memory wastage and code ambiguity. Thus it is easier to create a common parent (base) class and store all the common/general properties inside it. We should create the separate child classes containing only the specific non-common members. The child class can use the base class data members as if its own. He can also use the base class methods directly. We have another advantage of using inheritance; we get a discrete hierarchical structure of the classes and their responsibilities.

Principles of Inheritance:

i. Reusability
ii. Hierarchical definition of classes and their roles.

Polymorphism

It means a same task is performed in a different fashion at different levels of hierarchy. A parent class may have a behavior defining a set of tasks. A child inheriting properties from the parent can use the methods directly as well as its own modifications thus "overriding" the method from base class. It is commonly associated with method overriding. Dynamic polymorphism cannot exist without Inheritance.

History of Java

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called "Oak," but was renamed "Java" in 1995.

B) Java Features

Compiled and Interpreted

Java is a compiled programming language, but rather than compiling straight to executable machine code, it compiles to an intermediate binary form called JVM bytecode. The byte code is then compiled and/or interpreted to run the program.

Platform independent

Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere.

e.g.: Java code can run on multiple platforms e.g. Windows, Linux, Sun Solaris, Mac OS etc.

Object Oriented

Object-oriented Programming means we organize our software as a combination of different types of objects that incorporates both data and behavior. Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

I. Object
II. Class
III. Inheritance
IV. Polymorphism
V. Abstraction
VI. Encapsulation

Multithreaded and interactive

A thread is a smallest dispatchable unit of code, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it shares the same memory. Threads are important for multi-media, Web applications etc. Java makes use of awt, swing and Java FX to promote GUI API thus making the applications interactive.

High performance

Java code is faster than other codes of interpreted languages in UNIX environments such as perl, tcl, etc. or script languages as bash, sh, csh, etc. Java has high performance also because it uses threads.

C) Java Architecture

1. The byte code

A java class after being passed to the compiler is converted into a .class file. Bytecodes are the machine language conversions done by the Java virtual machine. The bytecodes for a method are executed when that method is invoked during the course of running the program.

2. Java Development Kit (JDK)

The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (javadoc) and other tools needed in Java development.

3. Java Runtime Environment (JRE)

The Java Runtime Environment (JRE) is a set of software tools for development of Java applications. It combines the Java Virtual Machine (JVM), platform core classes and supporting libraries. JRE is part of the Java Development Kit (JDK), but can be downloaded separately.

4. Java Virtual Machine (JVM)

The Java Virtual Machine (JVM) is an abstraction layer between a Java application and the underlying platform. JVM acts as a “virtual” machine or processor. To the bytecodes comprising the program, they are communicating with a physical machine; however, they are actually interacting with the JVM.

D) Constants & Variables

To create a constant in Java, final keyword is used.

E.g. final int constant = 10;

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility and a lifetime. In Java, all variables must be declared before they can be used. The basic form of a variable declaration is:

Type identifier [= value][, identifier [= value] ...];

E.g.

```
int a, b, c;             // declares three ints, a, b, and c.
byte z = 22;             // initializes z.
double pi = 3.14159;     // declares pi.
char x = 'x';            // the variable x has the value 'x'.
float f=12.2f;           // f determines that the literal is a float
```

E) Operators and Loops

Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

The Bitwise Operators

Java defines several bitwise operators that can be applied to the integer types, long, int, short, char and byte. These operators act upon the individual bits of their operands.

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

The Bitwise Logical Operators

The bitwise logical operators are &, |, ^, and ~. The following table shows the outcome of each operation. The bitwise operators are applied to each individual bit within each operand.

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

Relational Operators

The relational operators determine the relationship that one operand has to the other. Specifically, they determine equality and ordering.

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Conditional Operator

Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements. This operator is the ?. It can seem somewhat confusing at first, but the ? can be used very effectively once mastered. The ? has this general form: expression1 ? expression2 : expression3 Here, expression1 can be any expression that evaluates to a boolean value. If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated. The result of the ? operation is that of the expression evaluated. Both expression2 and expression3 are required to return the same type, which can't be void.
E.g.

```
int a, b;

 a = 10;

b = (a == 1) ? 20: 30;
System.out.println ("Value of b is: "+ b );
```
Output: Value of b is: 30;

If-else statement:

if-else statements are also called as selection statements as they select a block of code to execute as per the mentioned condition.

General form:

```
if(condition){
        //logic
}
else{
        //logic
}
```

If block evaluates the condition and executes the block if the condition evaluates true. If is independent of else/else-if; but an else needs one if. Else cannot be shared across various if blocks. Thus if you want to check many conditions separately pertaining to a same part of logic, please use an if-else if ladder.

If-else if statement:

```
if(condition){
        //logic
}
else if(condition){
        //logic
}
else{
        //logic
}
```

Nested if-else:

We also can use one if-else or if-else if block inside one another creating a nested block.

E.g.:

```
if(condition){
        if(condition){
                //logic
                    }
else {
        //logic}
}
```

Loops:

When we need a block of code to execute repeatedly based upon some repeat condition, we use looping mechanism.

There are four basic forms of loops:

For loop

General form:

```
for(initial value;  termination;  updation){ // logic  }
```

We use a counter for counting the iterations of the loop. The counter variable can be created and initialized and updated at the proper place inside the for loop. The loop checks the termination condition per each iteration and executes the body if the condition evaluates true.

Note: We use a for loop when we know exactly how many times we have to iterate the loop.

While loop

General form:

```
while(termination condition){  // logic  }
```

While loop works exactly same on the same principle as a for loop; but the primary difference is that it doesn't have a special place to update or initialize the counter.

Note: We use a while loop when we are bothered only about the termination of the loop.

Do-while loop

General form:

```
do {
 // logic
}
while (termination condition);
```

Do block holds the logic to execute and while contains the termination logic. The main difference is that the condition is checked prior to execution in while loop whereas, the logic is executed first and the condition is checked later. So even if the condition is false, the do loop executes at least once.

Note: We use a do-while loop when we want to execute the logic at least once.

Typically it is used when we want our user to control whether the same logic needs to be executed again. We accept a choice from user and depending on the nature of the choice, we decide whether the do loop executes again or not. While loop generally checks the choice forwarded by the user.

switch-case

General form:

```
switch(choice) {
            case 1:
            //logic
            [break;]
            case 2:
            //logic
            [break;]
            case n:
            //logic
            [break;]
            default:
            //logic
}
```

When we have a set of operations to be performed depending upon what the user chooses from a list of options, we can use a switch case construct. We must display the choices to the user and accept a choice and pass it to the switch. Create cases relevant to the choices and each case can have a separate logic pertaining to the choice. If the choice entered by the user doesn't match any of the mentioned cases, the control is transferred to the default case. If we do not use break keyword, we fall through the switch cases executing all the cases which don't have a break.

- ✓ Switch supports int, float, char, long, double and even Strings inside Switch.
- ✓ A switch case program is also called a "Menu Driven" program.

Break:

The break keyword can be used only inside for, while, do-while or switch case loops. When the compiler encounters a break, it stops the execution of the loop and takes the program out of the loop.

Usage:

It is used when the logic demands immediate discontinuation of the remaining iterations of the loop. If it is used inappropriately, we get a compile time error stating illegal use of break or break statement misplaced.

Continue:

The continue keyword is opposite to break. It is used in the exact same place where break is used, but the main difference is that by encountering continue, the compiler skips to the next iteration instead of continuing the loop. Statements written after continue in the same block are unreachable.

usage:

It is used when the logic demands immediate discontinuation of the current iteration of the loop and skip to the next iteration. If it is used inappropriately, we get a compile time error stating illegal use of break or break statement misplaced.

F) Class Fundamentals

A class is a generic template that holds the abstracted details of a real life entity. A class is used as an envelope to encapsulate the attributes (data members) and behaviours(methods). A class can communicate with the external world (other classes) with the medium of object and that's why this paradigm of programming is called as "Object Oriented Programming"

A class can access its own members anytime within itself. We can define access specifications using specifiers like public, private, protected and default.

General form of a class

- ✓ As mentioned earlier, a class contains data members and member functions and constructors.
- ✓ Conventionally, each class name should start with Capital Case and should provide some meaning to the class. (Never use individual names as class names!)
- ✓ Avoid having too big names as they are hard to remember and document.
- ✓ Typical observation is that Java usually uses nouns as class names, verbs as method names and adverbs as Interface names.

```
class ClassName
{
// instance variable declaration
type1 varName1 = value1;
:
typeN varNameN = valueN;

//  Constructors
ClassName()
{
// body of default constructor
}
ClassName(param 1,...,param n)
{
// body of constructor
}
:
// Methods
return-type methodName(params 1, … ,params n)
{
// body of method
}
}
```

Class indicates that a class named ClassName is being declared. It must follow the java naming conventions for identifiers. Instance variables named varName1 through varNameN are normal variable declaration syntax. Each variable must be assigned a type shown as type1 through typeN and may be initialized to a value shown as valueN. Constructors always have the same name as the class. They do not have return values. param 1 through param N are optional parameter lists. Methods named methodName can be included. The return type of the methods are return-type and params 1 through params n are optional parameter lists.

Simple class Student

We have created a simple class Student having three instance variables; namely: id, name and marks. We also created a default constructor which will initialize the objects and the instance variables by default values. We have a parameterized constructor which will initialize the instance variables by the values of the parameters it contains. We have a print method which prints the values of Student's variables on the console.

```
public class Student
{
int id;
String name;
float marks;
Student(){
System.out.println("Student Class Default Constructor");
}
Student(int id,String name,float marks){
System.out.println("Student Class Parameterized Constructor");
        this.id=id;
        this.name=name;
        this.marks=marks;
}
void print(){
System.out.println("Printing Student Details");
System.out.println("Id: "+ id);
System.out.println("Name: "+ name);
System.out.println("Marks: "+ marks);
}
}
```

G) Access Control

It is always important to specify the access limitations over each member of a class. A class may contain sensitive data which shouldn't be accessed directly through an object by the outer world. Thus we use access specifiers like public, private, protected and default. Following table summarizes the nature of the access specifiers.

Access Specifiers	Same Class	Same Package	Subclass	Other packages
public	Y	Y	Y	Y
protected	Y	Y	Y	N
default	Y	Y	N	N
private	Y	N	N	N

Note: A class should never be assigned private because; no other class will be able to even create objects of a private class. Instead go for a protected class.

Methods taking parameters:

We have an Area class having two methods, which find the area of a circle and a rectangle. Both methods accept a parameter in order to calculate the respective areas. TestArea class has the main method which will control the flow of execution. We have created an object of the Area class and invoked circle() and rectangle() methods. Both methods print the resulting area to the console.

Area.java	TestArea.java

class Area
{
double area;
void circle(int radius){
area=radius*radius*3.14f;
System.out.println("Area of circle:" + area);
}
void rectangle(int length, int breadth){
area=length*breadth;
System.out.println("Area of rectangle:"+ area);
}
}
	class TestArea
	{
	public static void main(String args[])
	{
	Area a = new Area();
	a.circle(12);
	a.rectangle(10,5);
	}
	}

Output

```
Area of circle: 452.1600036621094

Area of rectangle: 50.0
```

H) Constructors

Constructors are essential for creating and initializing objects. When an object is instantiated using the new keyword, the respective constructor is invoked automatically. Constructor invokes itself as many times as the object is created. It creates memory for the instance variables on the HEAP memory and assigns that memory to the object.

Creating Constructors:

- ✓ Must have the same name as class name.
- ✓ Can't have a return type. Constructors having a return type are treated as methods.

Types of Constructors:(refer sample Student class)

```
public class Student{

int id;

String name;

float marks;

}
```

i. **Default Constructor:**

```
Student(){
System.out.println("Student Class Default Constructor");
}
```

- ✓ Default Constructor initializes instance variable values to default values (0 for numeric and character, false for boolean and null for any reference of another class).
- ✓ It is always written by the compiler if we don't write it.
- ✓ Should be used when we don't know the exact values of the instance variables.

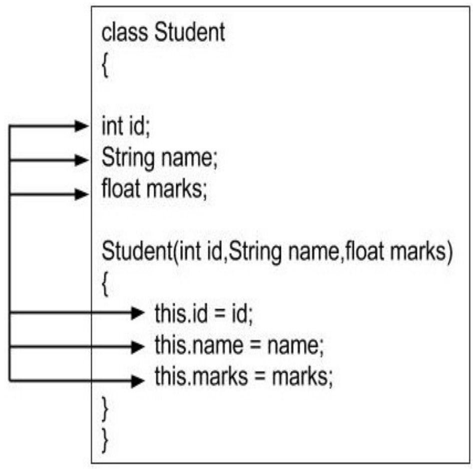
ii. **Parameterized Constructor:**

```
Student(int id,String name,float marks){
System.out.println("Student Class Parameterized Constructor");
this.id=id;
this.name=name;
this.marks=marks;
}
```

- ✓ Parameterized constructor initializes the instance variables by the values of the parameters it contains. It uses "this" keyword to resolve the ambiguity of which variable to use as we can see two sets of id, name and marks.
- ✓ If parameterized Constructor is written then default Constructor is not written by compiler.
- ✓ Should be used when we know the exact values of the instance variables.

this keyword

definition: Current object of the class which is alive in memory.



```
class Student
{

int id;
String name;
float marks;

Student(int id,String name,float marks)
{
    this.id = id;
    this.name = name;
    this.marks = marks;
}
}
```

When we want to point to the current object of a class from within the same class, no need to create an object; instead we can point to the members (variables or methods) using "this" keyword.

this.id indicates the instance variable id and so on. "this" is used primarily to differentiate between instance variables and local variables when both have same names.

As we know that instance variables are accessible via an object, we can differentiate both the type of variables with the help of this keyword, as this keyword is just a temporary name for some object which will be alive in memory and hold the values of the actual instance variables.

Instance Variable hiding

Encapsulation concept says that the data members should be enveloped in a class and should not be directly available to the outside classes. Hence we create our instance variables as private and use the concept of getter-setter methods to access them outside class.

(Re factored Student class using instance variables hiding concept)

```
public class Student {
private int id;
private String name;
private float marks;
public Student()
{
System.out.println("Default Constructor");
}
public Student(int id,String name,float marks)
{
System.out.println("Parameterized Constructor");
	this.id=id;
	this.name=name;
	this.marks=marks;
}
public int getId() {
	return id;
}
public void setId(int id) {
	this.id = id;
}
public String getName() {
	return name;
}
public void setName(String name) {
	this.name = name;
}
public float getMarks() {
	return marks;
}
public void setMarks(float marks) {
	this.marks = marks;
}
public void print(){
System.out.println("Id: "+id+" Marks: "+marks
+" Name: "+name);
}
}
```

```
public class TestStudent
{
public static void main(String[] args)
{
Student s1 = new Student(1,"John",67.35f);
s1.print();
}
}
```

Output:

```
Parameterized Constructor
Id: 1 Marks: 67.35 Name: John
```

I) Method and Constructor Overloading

Method overloading

definition

When more than one method have same name, but difference in parameters mentioned in the following ways:

1. ***Number of parameters***

   e.g. void area(int s){.....}
   void area(int l, int b){.....}

2. ***Type of parameters***

   e.g. void add(int a, int b){.....}
   void add(float a, float b){.....}

3. ***Sequence of parameters***

   e.g. void add(int a, float b){.....}
   void add(float a, int b){.....}

When multiple methods have a similar tasks to perform but they perform it in a slightly different way, we can keep the method names same having difference in parameters and create easy-to-understand codes. When we invoke one of the overloaded methods, compiler dissolves the same name ambiguity on the basis of parameters. Return types may vary from method to method.

Arith.java	**TestArith.java**

class Arith{
void add(int a, int b){
System.out.println("Sum of int: "+ (a+b));
}
 	
void add(float a, float b){
System.out.println("Sum of float: "+ (a+b));
}
}
	class TestArith
	{
	public static void main(String args[])
	{
	    Arith arith = new Arith();
	    arith.add(5,5);
	    arith.add(15.5f,15.06f);
	}
	}

Output

```
Sum of int: 10
Sum of float: 30.560001
```

Constructor Overloading

Definition

When more than one constructors are present in a class, they must have difference in the parameters they accept. Differences in parameters follow the rules of method overloading. We create multiple constructors when we want to facilitate the user to be able to create objects using various ways. Mostly, common occurrences of constructor overloading are found when we have one default (also called as no argument constructor) and at least one parameterized constructor created in the class. We can also create one parameterized constructor for every instance variable if needed. But this is not a common practice. Usually two-three constructors are sufficient as per the programmer's requirement.

e.g.

```
public class Student
{
int id;
String name;
float marks;
Student()
{
        System.out.println("Student Class Default Constructor");
}

Student(int id,String name,float marks)
{
        System.out.println("Student Class Parameterized Constructor");
        this.id=id;
        this.name=name;
        this.marks=marks;
}
```

e.g. of 1st constructor usage: Student s = new Student();

e.g. of 2nd constructor usage: Student s = new Student(1,"John",66.55f);

J) Containment (has-a-relationship)

When we include Object of another class as an attribute of one class it is called as containment. E.g. Employee Class needs a Date of Birth field. We can use the default java.lang.Date class but we can't modify it as per our requirements. Hence the best option is to create a new class Date and add attributes as needed.

Employee.java	**Date.java**

public class EmployeeDate {
private int id;
private String name;
private float salary;
private **Date dob**;
 	
public EmployeeDate() {
 	
}
 	
public EmployeeDate(int id, String name, float salary, Date dob) {
 	
this.id = id;
this.name = name;
this.salary = salary;
	public class Date {
	private int day,month,year;
	public Date() {
	}
	public Date(int day, int month, int year) {
	this.day = day;
	this.month = month;
	this.year = year;
	}
	public int getDay() {
	return day;
	}
	public void setDay(int day) {
	this.day = day;
	}
	public int getMonth() {

```
this.dob = dob;
}

//Getters and Setters can go here

public void print() {
System.out.println("Printing Employee's Details");
System.out.println("Id: "+id);
System.out.println("Name: "+name);
System.out.println("Salary: "+salary);
System.out.println("Printing the Date of birth");
dob.print();
}
}
```

```
return month;
}
public void setMonth(int month) {
this.month = month;
}
public int getYear() {
return year;
}

public void setYear(int year) {
this.year = year;
}

public void print() {
System.out.println(day+"-"+month+"-"+year);
}
```

TestEmployeeDate.java

```
import java.util.Scanner;
public class TestEmployeeDate {
public static void main(String[] args) {
int id=0,dd,mm,yy;
String name="";
float salary=0.0f;
System.out.println("Enter the Employee
details:(id,name,salary)");
Scanner scan = new Scanner(System.in);
id=scan.nextInt();
name=scan.next();
salary=scan.nextFloat();
System.out.println("Enter the date of birth
(day,month,year)");
dd=scan.nextInt();
mm=scan.nextInt();
yy=scan.nextInt();
EmployeeDate e1 = new EmployeeDate
(id,name,salary, new Date(dd, mm, yy));
e1.print();
        }
}
```

Output:

```
Enter the Employee details:(id,name,salary)
1 John 12345.6

Enter the date of birth (day,month,year)
1 1 2011
Printing Employee's Details
Id: 1
Name: John
Salary: 12345.6
Printing the Date of birth
1-1-2011
```

K) Data Types and Wrapper Classes

Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean. The primitive types are also commonly referred to as simple types. These can be put in four groups:

Integers: This group includes byte, short, int, and long, which are for whole-valued signed numbers.

Floating-point numbers: This group includes float and double, which represent numbers with fractional precision.

Characters: This group includes char, which represents symbols in a character set, like letters and numbers. Java uses a UNICODE format for its characters. Each character takes 2 bytes of storage. They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators.

E.g. a-z, Z-Z, 0-9 and all symbols that can be printed on screen can be stored as a character.

Boolean: This group includes boolean, which is a special type for representing true/false values.

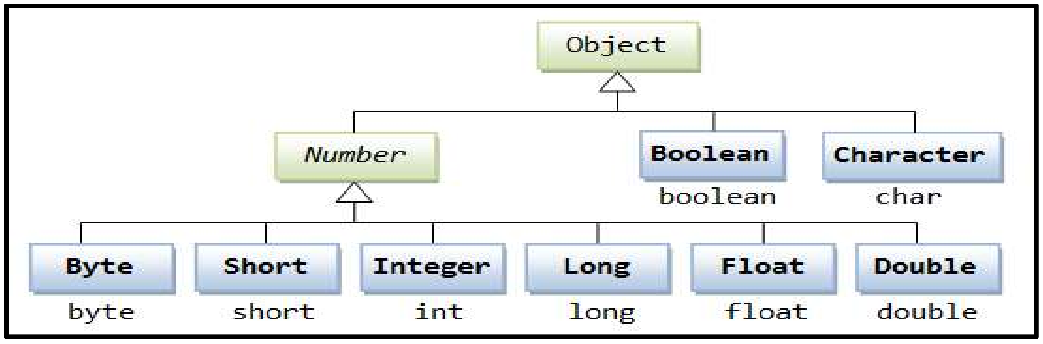
In Java, a wrapper class is defined as a class in which a primitive value is wrapped up. These primitive wrapper classes are used to represent primitive data type values as objects. The Java platform provides wrapper classes for each of the primitive data types. For example, Integer wrapper class holds primitive ‘int’ data type value so on for other primitive types.

1. Wrapper classes provide safety to the primitives

2. Ready made methods to convert primitives into special data types like String.

When a primitive is converted into a Wrapper object, the process is called **BOXING**

```
int i = 26;    // Primitive data type 'int'
Integer ii = new Integer(i);
OR
Integer ii = Integer.valueOf(i);
```



When a Wrapper object is converted into a primitive, the process is called **UNBOXING**

```
int i2 = ii.intValue();

String s = “26”;

int i3 = Integer.parseInt(s);
```

Java automatically performs both the steps JDK version 1.5 onwards. These processes are called **AUTOBOXING** and AUTO **UNBOXING**

The valueOf() and intValue() methods are available for all Wrapper Classes excluding Character. Hence these methods are said to have a form xxxValue() where xxx represents any Wrapper data type.

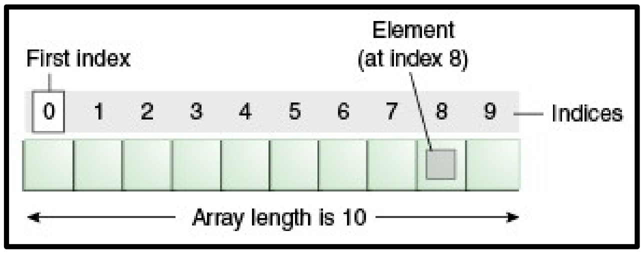
L) Arrays

Arrays are a group of elements belonging to a same data type. We use arrays when it is difficult to create multiple variables having different names but they belong to a same logical group of properties. E.g. A student has 5 subjects. To store the respective marks we need to create 5 variables namely, int m1,m2,…,m5. Instead of doing that, we can create an array of those 5 variables. E.g. int marks[5];

We use subscript notation i.e. [ ] to indicate that the variable is of an array type.

Valid Declarations:

```
int arr[ ];                 //creates an empty array
int[ ] arr;                 //creates an empty array
int arr[ ] = null;          //creates an empty array and initializes to null
int arr[ ] = {1,2,3,4,5};   //creates and initializes arr array with 5 values.
int arr[ ] = new int[5];    //creates and initializes arr array with 5 values.
```



An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.

```
int a[ ] = new int[10];

a[8]=10;
```

Here we have created an array of 10 values. We have only inserted a value at 8th index. Thus referring to any other index will give us "ArrayIndexOutOfBoundsException" meaning the element index we are trying to access doesn't contain any value.

One-dimensional array

When we have an array containing n number of elements, we call it as 1D Array or One-dimensional array.

E.g.

SumDemo class accepts n marks from the user and prints their sum.

```
import java.util.Scanner;
class SumDemo{
public static void main(String args[])  {
Scanner scan = new Scanner(System.in);
int sum = 0, n = 0;
System.out.println("Enter the no. of marks:");
n = scan.nextInt();
int array[ ] = new int[n];
System.out.println("Enter the elements:");
for (int i=0; i<n; i++)
    array[i] = scan.nextInt();
for( int i=0; i<n; i++)
    sum = sum+array[i];
```

Output

```
Enter the no. of marks:
5
Enter the elements:
50
50
50
50
50
Sum of array elements is:250
```

```
System.out.println("Sum of array elements is:"+sum);
          }
}
```

Two-dimensional array

When we want to store group of array as one array, we go for 2D or Two - dimensional array. For e.g. we want to store marks of 4 subjects for 3 students each. So we either need to create 3 single arrays for 4 subjects or in other words, create 3 rows having 4 columns each.

Declaration & initialization of 2D arrays

```
int a[ ][ ];                      //creates m*n empty array
int[ ][ ] a;                      //creates m*n empty array
int a[ ][ ] = null;               //creates m*n empty array and assigns to null
int a[ ][ ]={{1,2},{3,4},{5,6}};  //creates 3*2 array
int a[ ][ ] = new int[3][3];      //creates 3*3 empty array
```

E.g. SumDemo2D class accepts marks for n subjects for m students from the user and prints their sum.

```
import java.util.Scanner;
class SumDemo2D{
          public static void main(String args[])  {
                    Scanner scan = new Scanner(System.in);
                    int sum = 0, m = 0, n = 0;
          System.out.println("Enter the no. of students:");
                    m = scan.nextInt();
                    System.out.println("Enter the no. of subjects:");
                    n = scan.nextInt();
                    int array[ ][ ] = new int[m][n];
                    System.out.println("Enter the marks:");
          for (int i=0; i<m; i++)   {
                    for( int j=0; j<n; j++)   {
                              array[i][j] = scan.nextInt();
                              sum += array[i][j];
                    }
System.out.println("Sum of array elements is: " + sum);
                              sum=0;
                    }         }         }
```

Output

```
Enter the no. of students:
3
Enter the no. of subjects:
3
Enter the marks:
10
10
10
Sum of array elements is:30
20
20
20
Sum of array elements is:60
30
30
30
Sum of array elements is:90
```

Array of Objects

When we want to store multiple objects that can be part of a same logical group, we can store them in an array. While declaring an array, the array Data Type will be the class name of the object we are trying to store. Every array index must be either instantiated, or placed with an object. Failing to do so may give us a NullPointerException while trying to retrieve the object using array index as no object was found.

E.g. ArrayOfEmployees class creates an array of EmployeeDate objects of size 'n'. User can store id, name, salary and his date of birth in the array.

ArrayOfEmployees.java

```
import java.util.Scanner;
public class ArrayOfEmployees {
        public static void main(String[] args) {
                int id=0,dd,mm,yy,n=0;
                String name="";
                float salary=0.0f;
                Scanner scan = new Scanner(System.in);
                System.out.println("Enter no. of employees");
                n=scan.nextInt();
        EmployeeDate[] emp = new EmployeeDate[n];
                for(int i=0;i<n;i++){
System.out.println("Enter the Employee details :
(id,name,salary)");
                id=scan.nextInt();
                name=scan.next();
                salary=scan.nextFloat();
System.out.println("Enter the date of birth (day,month,year)");
                dd=scan.nextInt();
                mm=scan.nextInt();
                yy=scan.nextInt();
emp[i]=new EmployeeDate(id,name,salary,
new Date(dd, mm, yy));
                }
                for(int i=0;i<n;i++)
                emp[i].print();
System.out.println("Print using enhanced for loop");
                for(EmployeeDate e:emp)
                        e.print();
        }
}
```

Output

```
Enter no. of employees:2
Enter the Employee
details:(id,name,salary)
1
John
12345.6
Enter the date of birth
(day,month,year)
10
10
2000
Enter the Employee
details:(id,name,salary)
2
Adam
23456.3
Enter the date of birth
(day,month,year)
3
6
2003
Printing Employee's Details
Id: 1
Name: John
Salary: 12345.6
Printing the Date of birth
10-10-2000
Printing Employee's Details
Id: 2
Print using enhanced for loop:
Name: Adam
Salary: 23456.3
Printing the Date of birth
3-6-2003
Printing Employee's Details
Id: 1
Name: John
Salary: 12345.6
Printing the Date of birth
10-10-2000
Printing Employee's Details
Id: 2
Name: Adam
Salary: 23456.3
Printing the Date of birth
3-6-2003
```

M) CLA (Command Line Arguments)

A Java application can accept any number of arguments from the command line. This allows the user to specify configuration information when the application is launched. The user enters command-line arguments when invoking the application and specifies them after the name of the class to be run.

E.g. A java program accepts some set of numbers and displays their sum on running.

Then, java CLA 1 2 3 4 5

The parameters 1 2 3 4 5 will be stored in sequence in the String[] args array. We can convert them in ints using parseInt() from Integer class.

```
public class TestCLA {
	public static void main(String[] args) {
		int sum=0;
		for(int i=0;i<args.length;i++)
		{
			sum+=Integer.parseInt(args[i]);
		}
		System.out.println("Sum is :"+sum);
	}
}
```

Output

```
Sum is :21
```

N) VARARGS (Variable Arguments)

When we don’t know the exact number of parameters to be passed to a function, the concept of varargs can be used.

syntax

void add(int... a)

Here, int...a means any number of int parameters can be accepted

Rules:

- ✓ There can be only one variable argument in the method.
- ✓ Variable argument (varargs) must be the last argument.

Sum.java	**TestSum.java**

public class Sum{
void add(int... a){
 int sum=0;
 for(int i:a){
  sum+=i;
 }
System.out.println("Sum is: " + sum);
}
}
	public class TestSum {
	
	public static void main(String[] args)
	{
	 Sum sum1 = new Sum();
	 sum1.add(1,2,3,4,5,6);
	 }
	}

Output:

Sum is: 21

O) Pass by value and Pass by reference

When we pass the variable value as parameters, its called as pass by value. We don't have pointers in java thus, pass by address is not possible. If we want to modify multiple values and update them without returning, we should put those values in an array, operate on the array. An array is auto-updated in the calling function thus no need to explicitly return it. This is pass by reference in java.

SquareNumbers.java	TestSquare.java

public class SquareNumbers {
    void findSquare(int n[]) {
        for(int i=0;i<n.length;i++)
        n[i]=n[i]*n[i];
    }
}
	public class TestSquare {
	public static void main(String[] args) {
	int n[]={2,3,4};
	s.findSquare(n);
	for(int i:n)
	System.out.println("Squared Numbers: " + i);
	    }
	}

Output:

```
Squared Numbers: 4
Squared Numbers: 9
Squared Numbers: 16
```

P) Using Packages

It's very important to use proper packages when the number of classes/files is more. It becomes confusing to understand and manage a huge set of classes or files in a same project. We should logically group the classes and keep them under a package. Use appropriate access specifiers to control the visibility of the members.
Syntax

package package_name

E.g. package com;

class Test{...}

You can see the JVM building a new folder with the package name as soon as we create a package. We can import the files in one package from another package by using the import statement.

Syntax

import package_name.class_name;

E.g. import com.Test;

We also include built in java classes using import. *E.g.* import java.util.Scanner;

import com.* will import all classes under com. But it will not include classes under sub packages. We need to explicitly import them.

Static Import

We can import a static member which is used frequently by using the concept of static import. E.g. We use System.out.println() many times in our code. Thus we can import static member out.

Syntax

import static classname.member_name;

E.g. import static java.lang.System.out;

Chapter 2: Inheritance

A) What is Inheritance?

The process of deriving properties from an existing class to create a new class which may add his own properties is called as Inheritance. 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 superclass (also a base class or a parent class).Except Object class, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, 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 superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass (using implicit/explicit super()).

E.g. we have to computerize an automobile showroom which sells cars and bikes. Thus we create respective classes.

class Car
{
String name;
String color;
int speed;
String gearType;
}
	class Bike
	{
	String name;
	String color;
	int speed;
	int stroke;
	}

If we notice the above scenario, 90% of the fields are duplicated which results in high redundancy. What may be the solution?

Welcome to Inheritance.

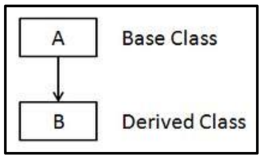
- ✓ Create a Vehicle class which has all the common things from both Car and Bike.
- ✓ Only the entity specific properties will be held in Child classes like Bike and Car.

class Car extends Vehicle
{
String gearType;
}
	class Bike extends Vehicle
	{
	int stroke;
	}
		class Vehicle
		{
		String name;
		String color;
		int speed;
		}

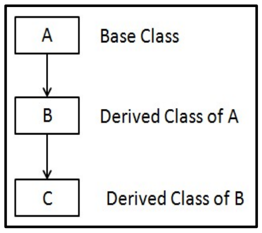
By using the extends keyword, a class becomes an immediate child class and inherits all the non private properties and methods of the immediate base class.

Simple/Single Inheritance

- ✓ When a class extends another class, single inheritance occurs.
- ✓ Class A is the base class here and class B is the derived class.
- ✓ B will derive all properties of the A class.



Multi-level Inheritance



- ✓ When a class extends another sub class which is extending another class, multi-level inheritance occurs.
- ✓ Class A is the base class here and class B is the derived class.
- ✓ C will derive all properties of the B class. But as we know, all the non private properties of A are already inherited by B class, those properties will be inherited by class C also.

Simple Inheritance sample program:

```
class Vehicle
{
String name, color;
int speed;
Vehicle()
{
}
void print()
{
System.out.println("Name: " + name);
System.out.println("Color: " + color);
System.out.println("Speed: " + speed);
}
void accelerate()
{
System.out.println("Vehicle speed is increasing");
}
}
```

```
class Car extends Vehicle
{
String gearType;
Car()
{
}
void print()
{
super.print();
System.out.println("Gear type: " + gearType);
}
void accelerate()
{
speed+=30;
System.out.println("Car's speed is increasing by
30 km/h: " + speed);
}
}
```

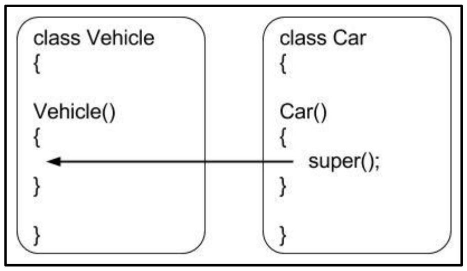
```
class Bike extends Vehicle
{
int stroke;
Bike()
{
}
void print()
{
super.print();
System.out.println("Stroke type: " + stroke);
}
void accelerate()
{
speed+=15;
System.out.println("Bike's speed is increasing by
15 km/h: " + speed);
}
}
```

```
class TestVehicles
{
public static void main(String args[ ])
{
Car c = new Car();
c.name="City";
c.color="White";
c.speed=25;
c.gearType="Auto";
System.out.println("Printing Car Details");
c.print();
c.accelerate();

Bike b = new Bike();
b.name="Pulsar";
b.color="Silver";
b.speed=15;
b.stroke=4;
System.out.println("Printing Bike Details");
b.print();
b.accelerate();
}
```

Output

```
Printing Car Details
Name: City
Color: White
Speed: 25
Gear type: Auto
Car’s speed is increasing by 30 km/h: 55
Printing Bike Details
Name: Pulsar
Color: Silver
Speed: 15
Stroke type: 4
Bike’s speed is increasing by 15 km/h: 30
```

B) Use of super



```
class Vehicle              class Car
{                          {

Vehicle()                  Car()
{                          {
                             super();
}                          }

}                          }
```

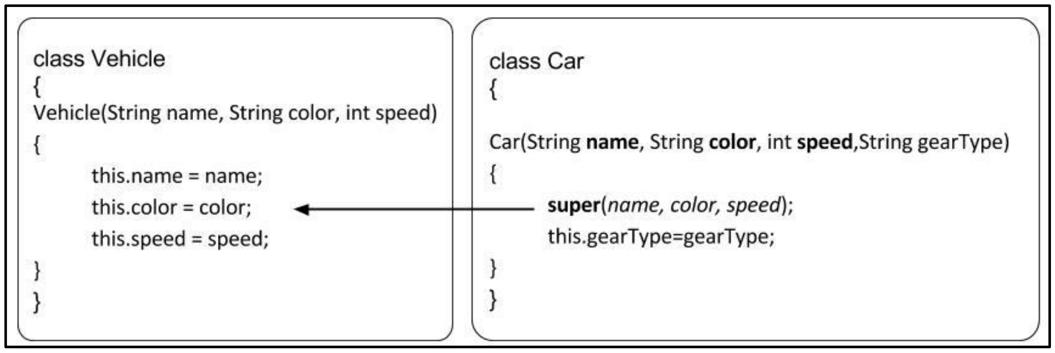
As we know, the child class will always have more properties than the base class. It is pointless to create an object of base class, because it lacks the specific details. Instead, we tend to create objects of child classes. But this raises a problem. It is the constructor which creates and initializes the instance variables. If we never create object of the base class, the constructor will never execute which in turn never initializes the instance variables.

Thus, it is very important that the base class constructor is executed implicitly by the child class. Hence, the compiler adds a super() statement which does the same.

Note: Child class constructor calls immediate base class constructor, which in turns calls its base class constructor (if any). Finally the top level super class constructor completes successfully and it gives the control back to the underlying class constructor.

super() gives a call to the DEFAULT constructor of the base class, this super is automatically written by the compiler, if the base class has a DEFAULT constructor. It is also called as implicit super call.

Explicit super call:



```
class Vehicle
{
Vehicle(String name, String color, int speed)
{
        this.name = name;
        this.color = color;
        this.speed = speed;
}
}
```

```
class Car
{

Car(String name, String color, int speed,String gearType)
{
        super(name, color, speed);
        this.gearType=gearType;
}
}
```

When we have the exact values of the instance variables, we create an object of the child class using the parameterized constructor. Again we face the same problem. User will pass all the values assuming values belong to the child class. But in reality we need to pass the base class variable values from the child class parameterized constructor. In this case, we define the base class variables in the child class parameterized constructor and use an explicit super call containing those values and pass those values to the base class parameterized constructor.

Following program depicts how the parameterized constructor of the base class can be invoked from the parameterized constructor of the child class.

```
class Vehicle
{
String name,color;
int speed;
Vehicle()
{
}
Vehicle(String name, String color, int speed)
{
	this.name = name;
	this.color = color;
	this.speed = speed;
}
}

class Car extends Vehicle
{
String gearType;
Car()
{
}
Car(String name, String color, int speed,String gearType)
{
	super(name, color, speed);
	this.gearType=gearType;
}
}
```

Note: print() and accelerate methods are assumed to be there in the classes.
Explicit super call is not written automatically by the compiler.

New TestClass after using parameterized constructors:

```
public class TestVehicle
{
	public static void main(String[] args)
	{
		Car c = new Car("City","White",25,"Auto");
		System.out.println("Printing Car Details");
		c.print();
		c.accelerate();

		Bike b = new Bike("Pulsar","Silver",15,4);
		System.out.println("Printing Bike Details");
		b.print();
		b.accelerate();
	}
}
```

It produces the same output as before:

```
Printing Car Details
Name: City
Color: White
Speed: 25
Gear type: Auto
Car's speed is increasing by 30 km/h: 55
Printing Bike Details
Name: Pulsar
Color: Silver
Speed: 15
Stroke type: 4
Bike's speed is increasing by 15 km/h: 30
```

Constructor chaining

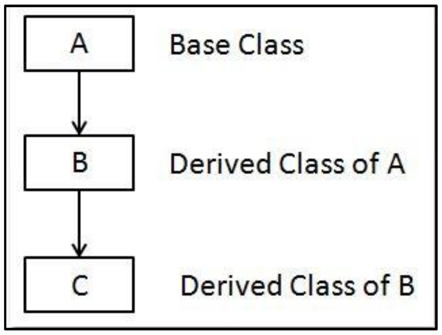
When we want to invoke constructor of a class from constructor of another class, we use this()

E.g.

```
class Test
{
int x;
Test()
{
this(0); // invokes the parameterized constructor
}
Test(int x)
{
 this.x = x;
}
}
```

Multilevel Hierarchy

```
class Vehicle{
void print(){
System.out.println("Vehicle");
}
}
class Car extends Vehicle{
void print(){
System.out.println("Car ");
}
}
class Sedan extends Car{
void print()
{
System.out.println("Sedan");
}
}
```



Final in inheritance

1. *final in variables* : creates constants

```
final float pi=3.14f;
```

Once the value is assigned, no other value can be assigned to the final variable. It behaves as a constant.

2. *final in methods* : methods can't be overridden

```
final void task(){...}
```

This method task() can't be overridden by any child class. It is typically used when we don't want the originality of the methods to be modified.

3. *final in classes* : class can't be extended

```
final class Test{...}
```

This Test class can't be extended by anyone, in other words, we don't want any child class to be created. We use this feature when we want to stop the hierarchy.

C) Rules of method overriding

- ✓ The signatures of the methods must be same.
- ✓ Overriding focuses on distinguishing the logic at every level of hierarchy.

D) Polymorphic Reference

Vehicle v = new Vehicle();//Creates a vehicle object having vehicle properties

Car c = new Car();//Creates a car object having vehicle+car properties

Thus,

- ✓ Vehicle v1 = new Car(); // is allowed

But

- × Car c1 = new Vechicle(); // is not allowed as new Vehicle() creates only vehicle properties

And,

Car c2 = (Car)v1; // is allowed as v1 was a car object but the reference was of vehicle.

These references are called as polymorphic references.

But,

Car c3 = (Car)v; // generates ClassCastException as v was never a Car object.

Also,

Vehicle get(){

return new Car(); // allowed as car object contains Vehicle properties also.

}

This is also called as ***dynamic*** polymorphism.

E) Interface and Abstract Class

Interface

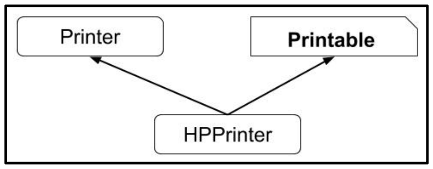
An interface is a group of related methods with empty bodies. The methods which don't have bodies are also called as abstract methods. An interface is a piece of contract which contains what the method should do; it doesn't state how it should do it. An interface is an abstract type that is used to specify a set of tasks that classes must implement. Interfaces are declared using the interface keyword, and may only contain method signature and constant declarations (variable declarations that are declared to be both static and final). All methods of an Interface do not contain implementation (method bodies). Interfaces cannot be instantiated, but rather are implemented. Interfaces don't have constructors. A class that implements an interface must implement all of the methods described in the interface, or be an abstract class. One benefit of using interfaces is that they simulate multiple inheritances. All classes in Java must have exactly one base class, but can implement any number of interfaces. All the methods are by default public abstract. These modifiers cannot be changed.

```
interface Bicycle {
void changeGear(int newValue);
void speedUp(int increment);
void applyBrakes(int decrement);
}
class HerculesBicycle implements Bicycle {
   int speed = 0;
   int gear = 1;
   void changeGear(int newValue) {
       gear = newValue;
   }
   void speedUp(int increment) {
       speed = speed + increment;
   }
   void applyBrakes(int decrement) {
       speed = speed - decrement;
   }
   void print() {
       System.out.println("cadence:" + cadence + "
speed:" + speed + " gear:" + gear);
   }
}

class TestBicycle
{
public static void main(String args[ ]){
HerculesBicycle h = new HerculesBicycle ();
         h.speed=10;
         h.gear=1;
         changeGear(2);
System.out.println("Gear Changed to: "+h.gear);
speedUp(10);
System.out.println("Speed increased to:
"+h.speed);
applyBrakes(5);
System.out.println("Speed decreased to:
"+h.speed);
}
}
```

Output:

```
Gear Changed to: 2
Speed increased to: 20
Speed decreased to: 15
```

Multiple inheritance in Java:



When a class extends more than one class, it is said to be multiple inheritance. Java doesn't support multiple inheritance. We can extend one class and implement an interface providing the concrete methods; it creates an illusion of multiple inheritance. We have a class HPPrinter which extends Printer class. HPPrinter will implement Printable interface providing the concrete methods.

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

```
class Printer{
String name;
int pages;
}
```

```
class HPPrinter extends Printer implements Printable{
public void print(){
System.out.println("Printing page one");
}
}
```

Abstract Class

- ✓ 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)
- ✓ If a class includes abstract methods, then the class itself must be declared abstract, as in:

public abstract class **Shape** {
double area;
abstract void findArea();
}

Shape.java

```
public abstract class Shape {
protected double area;
protected String name;
public abstract void findArea();
        public Shape(String name) {
                this.name = name;
        }
        public double getArea() {
                return area;
        }
        public void setArea(double area) {
                this.area = area;
        }
        public String getName() {
                return name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public void printShape(){
System.out.println("Shape name: " + name);
System.out.println("Area: " + area);
        }
}
```

Circle.java

```
public class Circle extends Shape{
float radius;
public Circle(float radius,String name) {
                super(name);
                this.radius = radius;
        }
public void findArea() {
                setArea(radius*radius*3.14f);
        }
public void printShape() {
                super.printShape();
System.out.println("Radius: " + radius);
        }
}
```

public class CreateShapes {
 	
public static void main(String[] args) {
 	
    Shape circle = new Circle(5.6f,"Circle");
        circle.findArea();
        circle.printShape();
    }
}
	Output
	
	Shape name: Circle
	Area: 98.4

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, then the subclass must also be declared abstract.

F) Object class and methods

The Object class contains many methods that are useful for child classes.

i. **hashcode()**: Every object is stored in at a memory location which is created from a unique code called as hashcode. Every object has a hashcode. If two objects are under equaity; i.e. e1=e1; both share a same hashcode.

ii. **equals(Object o):** When we want to compare two objects for equality, the "=" operator just checks the hashcode. But as we want to compare objects logically, hashcode alone can't determine the equality. In this case, we should override the equals method and check for the member values for equality.

```
public boolean equals(Object obj) {
Employee temp=null;
if(obj instanceof Employee){
temp=(Employee)obj;
if(this.id==temp.id && this.name.equals(temp.name) && this.salary==temp.salary)
return true;
}
return false;
}
```

iii. **toString():** When we need to convert an object into its String representation, we can override this method.

```
public String toString() {
return "Id: "+id+"\n"+"Name: "+name+"\n"+"Salary: "+salary+"\n" ;
}
```

Chapter 3: Miscellaneous

A) String, StringBuffer and StringBuilder

i. String

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects.

Creating String objects

```
String s = null;                  //creates a null string
String s = "";                    //creates an empty string
String s = new String();          //creates an empty string
String s = new String(“”);        //creates an empty string
String s = "Hello";               //creates string literal object assigning Hello
String s = new String(“Hello”);   //creates a string object and assigns Hello
```

String class is final i.e. we can’t extend it. All the methods of String are final i.e. can’t be overridden. This is done to stop the originality of the Strings from being modified.

String is an immutable class

Once you assign a string object, that object cannot be changed in memory. You cannot change the object itself, but you can change the reference to the object.

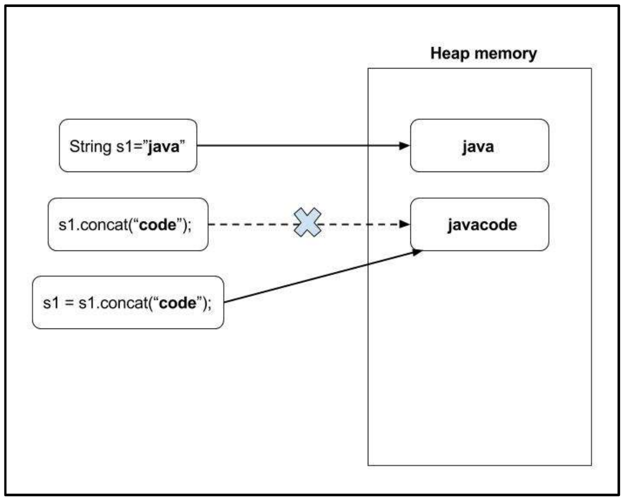
When a String s1 is concatenated with s2, the result of the operation is “javacode” but no is pointing (referencing) to it. Thus, even after the new string object being created in memory, s1 is still holding its old value.

```
String s1 = “java”;
String s2 = “code”;
s1.concat(s2);
System.out.println(s1);
```

Guess the output?
It prints java instead of printing javacode.

Right alternative?

```
String s1 = “java”;
String s2 = “code”;
s1=s1.concat(s2);
System.out.println(s1);
```



As we can see, when s1.concat(“code”) is performed, new string is present in memory but s1 isn’t referencing it.

When s1 = s1.concat(“code”) is performed, s1 gets updated as it references the new string “javacode”

This property where an object once created can never be modified; but can be referenced and unreferenced is called as **Immutability**.

String class methods

Assume :

String s1 = “java”;
String s2 = “code”;

Return Type	Name	Description
char	charAt(int index)	Returns the char value at the specified index s1.charAt(2) returns the char v
int	compareTo(String b)	Compares two strings lexicographically. Returns the value 0 if the argument string is equal to this string; a value less than 0 if this string is less than the string 2; and a value greater than 0 if this string is greater than the string 2. If s1=”b” and s2=”a”, s1.compareTo(s2) will return 1.
String	concat(String str)	Concatenates the specified string to the end of this string.
boolean	equals(Object anObject)	Compares this string to the specified object. Returns true if both strings match else returns false
int	length()	Returns the length of this string.
String	replace(char c1, char c2)	Returns a string resulting from replacing all occurrences of c1 in this string with c2. s1.replace(‘a’,’A’) returns “jAvA”
String	replace(CharSequence target, CharSequence replacement)	Returns a string resulting from replacing all occurrences of target string in this string with replacement. if s1=”java” then s1.replace(“ja”,”JA”) returns “JAva”
String	toUpperCase()	Converts all of the characters in this String to uppercase. s1.toUpperCase() returns “JAVA”
String	toLowerCase()	Converts all of the characters in this String to lowercase. If s1=”JAVA” then, s1.toLowerCase() returns “java”

ii. StringBuffer class

When we don’t know how much amount of data is to be copied as characters, we should go for using StringBuffer objects instead of String. These objects have a buffer memory. StringBuffer is **mutable**.

Constructors:

Name	Description
StringBuffer()	Constructs a string buffer with no characters in it and an initial capacity of 16 characters.
StringBuffer(int capacity)	Constructs a string buffer with no characters in it and the specified initial

	capacity.
StringBuffer(String str)	Constructs a string buffer initialized to the contents of the specified string.

Methods

Return Type	Name	Description
int	capacity()	Returns the current capacity.
StringBuffer	replace(int start, int end, String str)	Replaces the characters in a substring of this sequence with characters in the specified String.
StringBuffer	delete(int start, int end)	Removes the characters in a substring of this sequence.
StringBuffer	reverse()	Causes this character sequence to be replaced by the reverse of the sequence.
String	toString()	Returns a string representing the data in this sequence
StringBuffer	insert(int offset, String str)	Inserts the string into this character sequence.

iii. StringBuilder Class

There is only one difference in StringBuffer and StringBuilder class. StringBuilder is not synchronized thus it is not thread safe, but it is faster. All methods are same.

B) Inner classes

Consider that we have to computerize Café Coffee Day outlet. We create a class CafeCoffeeDay which can have properties address,capacity etc. A CafeCoffeeDay outlet sells ColdCoffe which can be described using name, flavour and cost. Then the class CafeCoffeeDay looks like:

```
class CafeCoffeeDay
{
String address;
int capacity;
String name;
String flavour;
float price;
}
```

But then name,flavour,price don't belong directly to CafeCoffeeDay. Those properties describe a ColdCoffee ! Thus we will create a class ColdCoffee situated inside the class CafeCoffeeDay. This is the concept of **INNER** class.

CafeCoffeeDay is the **OUTER** class and ColdCoffee is the **INNER** class.
When we have a group of properties which don't directly belong to one class; rather they constitute to another class which belong to the previous class, we go for inner classes.

Note: Inner classes aren't compulsory. They make your code more readable.

How to create object of inner class using outer class

- ✓ Inner class can access all the properties of the outer class including private ones directly, but the converse is not true.
- ✓ External classes can't create objects of inner classes directly without using the outer class object.

Assume outer class is named as Outer and inner class is named as Inner.

Option 1:

```
Outer out = new Outer();
Outer.Inner in = out.new Inner();                //creating explicit object of outer class
```

Option 2:

```
Outer.Inner in = new Outer().new Inner();    //creating anonymous object of outer class
```

The CafeCoffeDay and ColdCoffee class:

CafeCoffeeDay.java

```
public class CafeCoffeeDay {
String address;
int capacity;
CafeCoffeeDay(){
}
CafeCoffeeDay(String address, int capacity) {
this.address = address;
this.capacity = capacity;
}
void printCafeCoffeeDay()
{
System.out.println("Cafe Coffee Day address: "+address);
System.out.println("Cafe Coffee Day seating capacity: "+capacity);
}
class ColdCoffee{
String name;
String flavour;
float price;
ColdCoffee(){
}
public ColdCoffee(String name, String flavour, float price){
this.name = name;
this.flavour = flavour;
this.price = price;
}
void printColdCoffee(){
System.out.println("Coffee Name: "+name);
System.out.println("Coffee Flavour: "+flavour);
System.out.println("Coffee Price: "+price);
                }
        }
}
```

TestCafeCoffeeDay.java

```
public class TestCCD {
public static void main(String[] args) {
CafeCoffeeDay ccd = new CafeCoffeeDay("Dadar", 30);
CafeCoffeeDay.ColdCoffee coffee = ccd.new ColdCoffee("Mocha", "Vanilla", 125.3f);
ccd.printCafeCoffeeDay();
coffee.printColdCoffee();
        }
}
```

Output

```
Cafe Coffee Day address: Dadar
Cafe Coffee Day seating capacity: 30
Coffee Name: Mocha
Coffee Flavour: Vanilla
Coffee Price: 125.3
```

Nested classes

When you use inner classes, user must create an object of the outer class first and then use it to create inner class object. In some rare cases, if we want to grant the facility to directly create an inner class object bypassing the outer class, we can mark the inner class as static. Since it is static, the outer class name is sufficient to access the inner class. Static inner classes are also called as static nested classes.

Note:

- ✓ Creating static inner class doesn't mean that all the members inside that class become static. They still exhibit their own modifier properties; just the class becomes static.
- ✓ Only inner classes can be marked as static.

How to create object of a static inner class

Assume outer class is named as Outer and static nested class is named as StaticInner.

Option 1:

Outer.StaticInner in = new Outer.StaticInner();

The Outer and StaticInner class:

Outer.java	**TestOuter.java**

public class Outer{
static class StaticInner{
int i;
StaticInner(){
}
void printStaticInner(){
System.out.println("Static Inner Class variable i = " + i);
}
}
}
	public class TestOuter{
	public static void main(String[] args){
	Outer.StaticInner in = new Outer.StaticInner();
	in.i=10;
	in.printStaticInner();
	}
	}

Output
Static Inner Class variable i = 10

iv. Anonymous Inner Classes

Assume interface Bicycle and the concrete class BMX.

Then,

Bicycle b = new BMX();

Similarly, if we want to just add the body of the abstract method, without actually creating a class,

```
Bicycle b = new Bicycle(){
void changeGear(int gear){
//method 1 implementation
}// Rest methods
};
```

These classes are very important in reducing the lines of code and adding fast-coding.

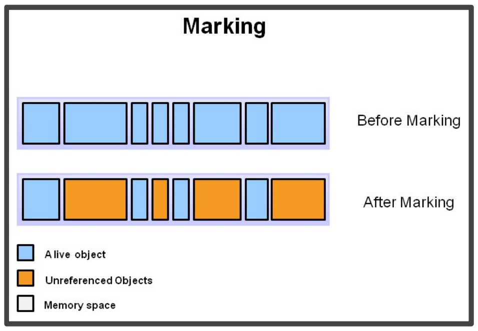
Java 8 supports this feature as “Lambda Expressions”

C) Garbage Collection in Java

Java has its own Garbage Collector module which operates independently of user’s control. Garbage Collection in java is automatic.

Automatic garbage collection is the process of looking at heap memory, identifying which objects are in use and which are not, and deleting the unused objects. An in use object, or a referenced object, means that some part of your program still maintains a pointer to that object. An unused object, or unreferenced object, is no longer referenced by any part of your program. So the memory used by an unreferenced object can be reclaimed.

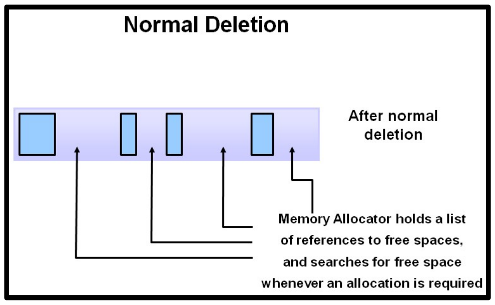
Step 1: Marking



The first step in the process is called marking. This is where the garbage collector identifies which pieces of memory are in use and which are not.

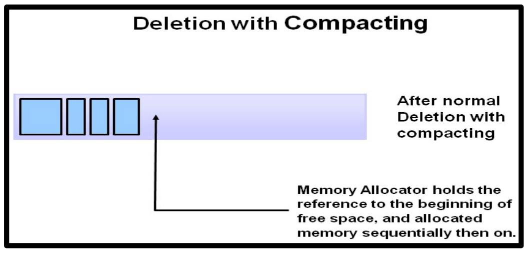
Referenced objects are shown in blue. Unreferenced objects are shown in gold. All objects are scanned in the marking phase to make this determination. This can be a very time consuming process if all objects in a system must be scanned.

Step 2: Normal Deletion


Normal Deletion
After normal deletion
Memory Allocator holds a list of references to free spaces, and searches for free space whenever an allocation is required

Normal deletion removes unreferenced objects leaving referenced objects and pointers to free space. The memory allocator holds references to blocks of free space where new object can be allocated.

Step 3: Deletion with Compacting


Deletion with Compacting
After normal Deletion with compacting
Memory Allocator holds the reference to the beginning of free space, and allocated memory sequentially then on.

To further improve performance, in addition to deleting unreferenced objects, you can also compact the remaining referenced objects. By moving referenced object together, this makes new memory allocation much easier and faster.

Being an automatic process, programmers need not initiate the garbage collection process explicitly in the code. System.gc() and Runtime.gc() are used to request the JVM to initiate the garbage collection process. Though this request mechanism provides an opportunity for the programmer to initiate the process but the decision is on the JVM. It can choose to reject the request and so it is not guaranteed that these calls will do the garbage collection. This decision is taken by the JVM based on the space availability in heap memory. We cannot force the JVM to perform Garbage Collection under any circumstance.

When an object becomes eligible for garbage collection?

1. Any instances that cannot be reached by a live thread.
2. Circularly referenced instances that cannot be reached by any other instances.

finalize()

The finalize method is called when an object is about to get garbage collected. That can be at any time after it has become eligible for garbage collection. This method is located in the Object class and needs to be overridden by the subclass. It's entirely possible that an object never gets garbage collected (and thus finalize may never be called). This can happen when the object never becomes eligible for gc (because it's reachable through the entire lifetime of the JVM) or when no garbage collection actually runs between the time the object become eligible and the time the JVM stops running...(this often occurs with simple programs). A subclass overrides the finalize method to dispose of system resources or to perform other cleanup.

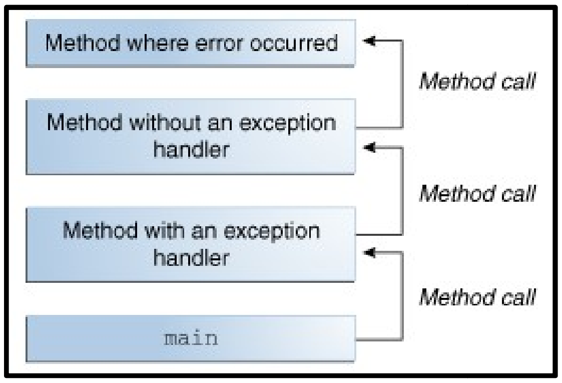
```
class Animal {
public static void main(String[] args) {
      Animal lion = new Animal();
      System.out.println("Main is completed.");
  }

protected void finalize() {
      System.out.println("Rest in Peace!");
   }
}
```

Chapter 4: Exception Handling

A) What is an exception?

definition

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

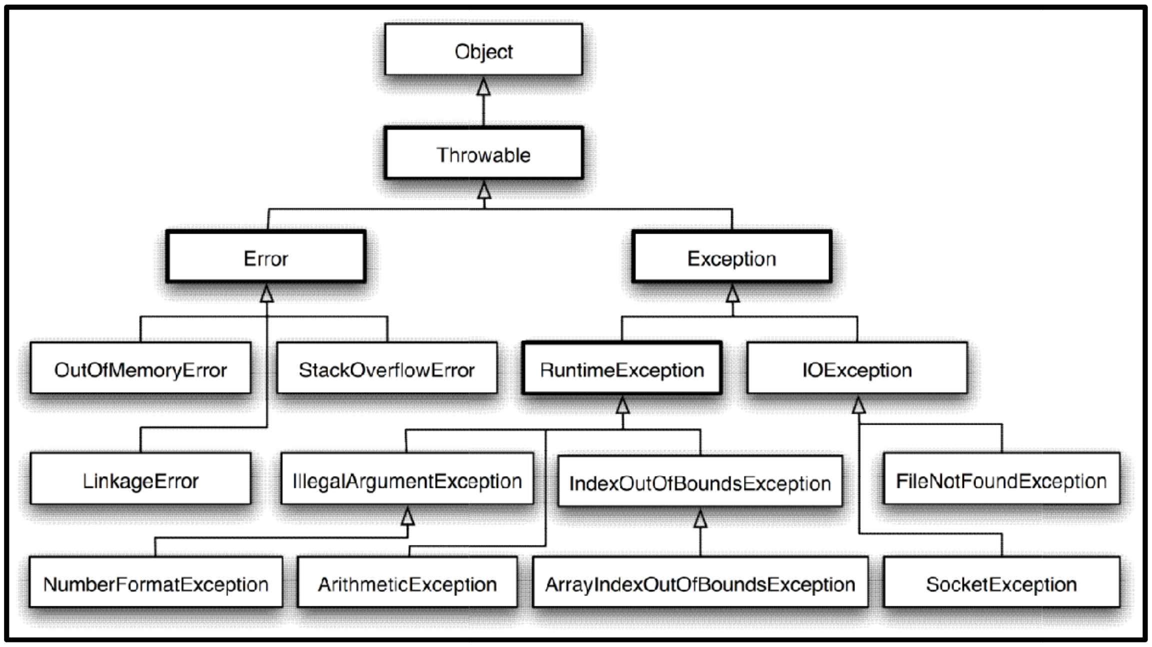

Method where error occurred
Method call
Method without an exception handler
Method call
Method with an exception handler
Method call
main

Exceptions are usually runtime logical errors which occur either because of wrong or incorrect format in input or when a programmer makes a logical mistake.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception.

After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the call stack.

Exception hierarchy :



Error vs Exception

- ✓ An Error "indicates serious problems that a reasonable application should not try to catch."
- ✓ An Exception "indicates conditions that a reasonable application might want to catch."

Handling Exceptions

We have two ways of handling exceptions.

1. Handle:

We handle the exceptions using try-catch-finally statements

```
class TestException{
public void static main(String args[])
{
try {
//	code that can generate exceptions
}
catch(ExceptionType e1)
{
//	exception handling or user friendly
messages
}
catch(ExceptionType e1)
{
//	exception handling or user friendly
messages
}
finally{
//	clean up code
}
```

When the code inside the catch block generates an exception, a relevant object of the Exception hierarchy is created and thrown to the catch block using throw keyword. The relevant catch block accepts the object and executes itself.

2. Declare:

We can use throws keyword to denote the JVM that we wouldn’t be handling the exceptions.

E.g.

```
void method() throws Exception,..,..,..
{
…..
…..//exception vulnerable logic
….
}
```

B) Try-catch-finally & throw-throws

We have already seen the use of try-catch and throws in the above e.g. Finally block executes irrespective of whether the exception occurs or not. It is used to clean up the resources used by the code like closing of files, IO Streams, Network or Database connections.

E.g.

```
public class TestExceptions{
public static void main(String args[]){
try{
int a = Integer.parseInt(“123e”); // either 1
int b = 30/0;                      // either 2
}
catch(ArithmeticException e){
System.out.println("Cannot divide by 0");
}
catch(NumberFormatException e){
System.out.println("Incorrect number is
entered");
}
catch(Exception e){
System.out.println("Some general problem has
occurred");
}
finally{
System.out.println("Always Printed");
}
}
}
```

Output:

```
in case 1:
Incorrect number is entered
in case 2:
Cannot divide by 0
```

throw statement

If an exception occurs at some line, the JVM automatically creates a new object of the relevant Exception class and throws the object to the catch block with the help of throw keyword. We do not require to write the throw keyword.

E.g.

```
try{
int a = Integer.parseInt("123e"); // either 1
throw new NumberFormatException();
...
...
}
```

C) User defined Exceptions

When we want to create and handle our own type of exceptions, we must extend the Exception class and override its methods, if needed and we can define our own methods as well.

```
public class Transaction {
int deposit(int bal,int amount)throws
NegativeAmountException {
if(amount<0){
throw new NegativeAmountException();
                    }
else{
          bal+=amount;
System.out.println("After depositing,
Balance="+bal);
          }
return bal;
}

int withdraw(int bal,int amount) throws
InsufficientBalanceException {
if(bal-amount<1000){
throw new InsufficientBalanceException();
                    }
else{
bal-=amount;
System.out.println("After withdrawing,
Balance="+bal);
                    }
                    return bal;
          }
}
```

```
public class NegativeAmountException extends
Exception{
          void printError(){
System.out.println("Negative amount is not
allowed.");
          }
}
```

```
public class InsufficientBalanceException extends
Exception{
public void printStackTrace() {
super.printStackTrace();
System.out.println("Please have minimum funds in
your account to withdraw.");
          }
}
```

```
public class TestTransactions {
public static void main(String[] args) {
Transaction t = new Transaction();
          try{      t.deposit(10000, 5000);
                    t.withdraw(15000, 25000);
             }
          catch(NegativeAmountException nae){
                    nae.printError();
          }
          catch(InsufficientBalanceException iae){
                     iae.printStackTrace();
                    }
          }
}
```

Chapter 5: Generics

A) What are generics?

Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar formal parameters used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

Code that uses generics has many benefits over non-generic code:

1. Stronger type checks at compile time.

A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

2. Elimination of casts.

3. Enabling programmers to implement generic algorithms.

Consider an Exam class which needs to store an exam code. Different exams can have different code formats depicted using various data types like String,Integer etc. Thus we must use Object as a data type for the attribute code.

```
public class Exam {
        Object code;
        public Exam(Object code) {
                super();
                this.code = code;
        }
public Object getCode() {
                return code;
        }
public void printCode() {
        System.out.println("Code type: " +
code.getClass().getSimpleName());
        System.out.println("Code: " + code);
        }
}
```

```
public class TestExam {
public static void main(String[] args) {
        Exam e1 = new Exam("Maths 1");
        e1.printCode();
        Exam e2 = new Exam(new Integer(1108));
        e2.printCode();
        }
}
```

Output

```
Code type: String
Code: Maths 1
Code type: Integer
Code: 1108
```

What if we want to retrieve the code in main() ?

```
Integer i2 = e2.getCode();              // But getCode returns Object; thus we need typecasting.

Integer i2 = (Integer)e2.getCode();     // Works perfectly
```

But,

String s = (Integer)e2.getCode(); // Results in ClassCastException, as the compiler fails to identify that an Integer object can never be cast to String. This exception halts the execution at the Runtime creating more problems.

Solution: Generics

Generics concept states that use an imaginary placeholder for the attribute type name. E.g. Object code will be replaced by, say T.

Thus the syntax and following changes.

```
public class GenericExam <T> {
T code;
public GenericExam(T code) {
        super();
        this.code = code;
        }
public T getCode(){
        return code;
        }
public void printCode() {
System.out.println("Code type: " + code.getClass().
getSimpleName());
System.out.println("Code: " + code);
        }
}
```

The main() looks like:

```
GenericExam<String> e1 = new
GenericExam<String>("Maths 1");
e1.printCode();
GenericExam<Integer> e2 = new
GenericExam<Integer>(new
Integer(1108));
e2.printCode();
```

The parameter passed as a type in < > must be same at both reference and instantiation. Failing to do so results in a compile time error.

Also if we try to wrongly typecast:

String s = (Integer)e2.getCode();   // Results in compile error instead of Exception: Type mismatch: cannot convert from Integer to String.

It is brilliant to check the type safety at compile time instead of Runtime.

B) Bounded Generics:

When we want the imaginary type T to accept only a particular set of related types, we use Bounded Generics by adding extends keyword in the type declaration near the class. The actual types passed must be child classes of the Type declared in the declaration. Failing to do so results in a compile time error.

E.g. class Test <T extends Base>

Here the child classes of Base can be passed to the Test class as actual parameters.

```
public class Maths <T extends Number>{
T n;
public Maths(T n) {
        super();
        this.n = n;
}
public void printNumber() {
System.out.println("Number type: " +
n.getClass().getSimpleName());
System.out.println("Number: " + n);
```

```
public class TestExam {
public static void main(String[] args) {

Maths<Integer> m1 = new Maths<Integer>(10);
m1.printNumber();
        }
}
```

System.out.println("Square of the number: " + n.doubleValue() * n.doubleValue()); } }	

Output

```
Code type: Integer
Number: 10
Square of the number: 100.0
```

C) Wildcard Generics

Consider we want to add the exam object in a log:

```
public class ExamLog {

        void addExam(GenericExam <T> e){

                System.out.println("Exam added");

                e.printCode();

        }

}
```

The problem is, we can’t declare a generic type ‘T’ inside the parameter of a method in another class. But we don’t know the actual type of parameter.

Thus we are allowed to use ‘?’ in place of T. ‘?’ means that the generic type is unknown.

```
public class ExamLog {

        void addExam(GenericExam <?> e){

                System.out.println("Exam added");

                e.printCode();

        }

}
```

Topic 6: Collection Framework

A collection is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data. Typically, they represent data items that form a natural group, like card deck (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).

A) What Is a Collections Framework?

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

I. **Interfaces**: These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation generally forming a hierarchy.
II. **Implementations**: These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
III. **Algorithms**: These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.

Shortcomings of an array based approach to handle group of elements

× Size is always fixed; changing size is next to impossible.
× Heterogeneous data types can't be stored.
× No predefined algorithms to manipulate, sort data.

Benefits of the Java Collections Framework

✓ **Reduces programming effort**: By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work.

✓ **Increases program speed and quality**: This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms.

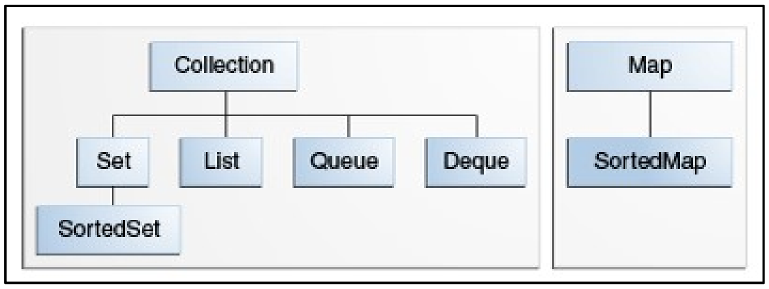
✓ **Allows interoperability among unrelated APIs**: The collection interfaces are the vernacular by which APIs pass collections back and forth.

✓ **Reduces effort to learn and to use new APIs**: Many APIs naturally take collections on input and furnish them as output.

✓ **Reduces effort to design new APIs**: Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.

✓ **Fosters software reuse**: New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

Collection Interfaces



i. **Collection**: The root of the collection hierarchy. A collection represents a group of objects known as its elements. The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List.

ii. **Set**: A collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the deck of cards, the courses making up a student's schedule, or the processes running on a machine.

iii. **List**: An ordered collection (sometimes called a sequence). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position).

iv. **Queue**: A collection used to hold multiple elements prior to processing. Besides basic Collection operations, a Queue provides additional insertion, extraction, and inspection operations. Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner.

v. **Deque**: A collection having one major difference to a Queue: All new elements can be inserted, retrieved and removed at both ends.

vi. **Map**: An object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value.

vii. **SortedSet**: A Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls.

viii. **SortedMap**: A Map that maintains its mappings in ascending key order. This is the Map analog of SortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories.

ix. **Iterator**: Underlying collection objects implement it and provide a way to traverse the collection. ListIterator is a richer specific Iterator meant to iterate only on Lists.

x. **Comparator**: A comparison function, which imposes a total ordering on some collection of objects. Used to order unnatural data members.

B) List: ArrayList, LinkedList and Vector

i. **ArrayList**: Offers constant-time positional access and is just plain fast. It does not have to allocate a node object for each element in the List.

ii. **LinkedList**: If you frequently add elements to the beginning of the List or iterate over the List to delete elements from its interior.

iii. **Vector**: Extremely similar to ArrayList, except it is synchronised creating thread safety.

Testing List:

```
import java.util.*;
public class TestList {
        public static void main(String[] args) {
                ArrayList a = new ArrayList();
//ArrayList<Integer> al = new ArrayList<Integer>(); use generics
                a.add("One");
                a.add(2);
                a.add(1,3);//inserting 3 at 1st index, 2 will be shifted
                a.add(5);
                a.add(4);
                System.out.println("List: "+a);
                System.out.println("Removing index '1'");
                a.remove(1);
                System.out.println(a);
                System.out.println("Removing '5'");
                a.remove(new Integer(5));
                System.out.println("List: "+a);
                Iterator iter = a.iterator();
                out.println("Using an iterator");
                while(iter.hasNext()){
                        System.out.println(iter.next());
                }
                System.out.println("Creating a linked list");
                LinkedList<Integer> linked = new LinkedList<Integer>();
                linked.add(1);
                linked.add(31);
                linked.add(12);
                linked.add(40);
                ListIterator listIter = linked.listIterator();
                System.out.println("Using a ListIterator");
                while(listIter.hasNext()){
                        System.out.println(listIter.next());
                }
        }
}
```

Output

```
List: [One, 3, 2, 5, 4]
Removing index '1'
[One, 2, 5, 4]
Removing '5'
List: [One, 2, 4]
Using an iterator
One
2
4
Creating a linked list
Using a ListIterator
1
31
12
40
```

Testing Vector

```
import java.util.*;
public class TestVector {
        public static void main(String[] args) {
        Vector<String> v = new Vector<String>();
        System.out.println("Adding values");
                v.addElement("a");
                v.addElement("b");
                v.addElement("c");
                v.addElement("d");
                v.addElement("e");
System.out.println("Printing Vector values:"+v);
```

Output

```
Adding values
Printing Vector values:[a, b, c, d, e]
Inserting 'j' at index 2
Removing Vector value at '3':c
Printing Vector values:[a, b, j, d, e]
```

System.out.println("Inserting 'j' at index 2");
v.add(2, "j");
System.out.println("Removing Vector value at '3':"+v.remove(3));
System.out.println("Printing Vector values:"+v);
}
}	

C) Set: HashSet, LinkedHashSet and TreeSet

i. **HashSet**: Doesn’t guarantee any ordering or sequence in storing objects.
ii. **LinkedHashSet**: Guarantees preservation of sequence as entered in storing objects.
iii. **TreeSet**: Guarantees ascending ordering upon stored objects.

Testing Set:

import java.util.*;
public class TestSets {
public static void main(String[] args) {
HashSet<Integer> hs = new HashSet<Integer>();
hs.add(1);
hs.add(3);
hs.add(2);
hs.add(6);
hs.add(5);
hs.add(9);
hs.add(86);
hs.add(56);
hs.add(77);
System.out.println("HashSet : " + hs);
 	
LinkedHashSet<Integer> lhs =
new LinkedHashSet<Integer>();
lhs.add(1); lhs.add(3);
	lhs.add(2); lhs.add(6);
	lhs.add(5); lhs.add(9);
	lhs.add(86); lhs.add(56);
	lhs.add(77);
	System.out.println("LinkedHashSet : " + lhs);
	
	TreeSet<Integer> ts = new TreeSet<Integer>();
	ts.add(1); ts.add(3);
	ts.add(2); ts.add(6);
	ts.add(5); ts.add(9);
	ts.add(86); ts.add(56); ts.add(77);
	System.out.println("TreeSet : " + ts);
	Iterator di = ts.descendingIterator();
	System.out.println("Printing in desc order");
	while(di.hasNext()){
	System.out.println(di.next()); }
	Set set = ts.descendingSet();
	System.out.println("Printing a desc set: " + set);
	} }

Output

```
HashSet : [1, 2, 3, 5, 6, 86, 56, 9, 77]
LinkedHashSet : [1, 3, 2, 6, 5, 9, 86, 56, 77]
TreeSet : [1, 2, 3, 5, 6, 9, 56, 77, 86]
Printing in desc order
86
77
56
9
6
5
3
2
1
Printing a desc set: [86, 77, 56, 9, 6, 5, 3, 2, 1]
```

D) Map: HashMap, LinkedHashMap and TreeMap

i. **HashMap**: Doesn't guarantee any ordering or sequence in storing objects.
ii. **LinkedHashMap**: Guarantees preservation of sequence as entered in storing objects.
iii. **TreeMap**: Guarantees ascending ordering upon stored objects.

Testing Map:

```
import java.util.*;
public class TestMaps {
public static void main(String[] args) {
HashMap<Integer, String> hm = new
HashMap<Integer, String>();
hm.put(1, "Rohit");     hm.put(2, "Rohan");
hm.put(3, "Mohit");     hm.put(5, "Jagjit");
hm.put(4, "Chirag");    hm.put(9, "Arun");
hm.put(7, "John");
System.out.println("Printing the HashMap: " +
hm);
LinkedHashMap<Integer, String> lhm = new
LinkedHashMap<Integer, String>();
lhm.put(1, "Rohit");    lhm.put(2, "Rohan");
lhm.put(3, "Mohit");    lhm.put(5, "Jagjit");
lhm.put(4, "Chirag");   lhm.put(9, "Arun");
lhm.put(7, "John");
System.out.println("Printing the LinkedHashMap: "
+ lhm);
```

```
TreeMap<Integer, String> tm = new
TreeMap<Integer, String>();

                tm.put(1, "Rohit");
                tm.put(2, "Rohan");
                tm.put(3, "Mohit");
                tm.put(5, "Jagjit");
                tm.put(4, "Chirag");
                tm.put(9, "Arun");
                tm.put(7, "John");

System.out.println("Printing the TreeMap: " +
tm);
        }
}
```

Output

```
Printing the HashMap: {1=Rohit, 2=Rohan, 3=Mohit, 4=Chirag, 5=Jagjit, 7=John, 9=Arun}
Printing the LinkedHashMap: {1=Rohit, 2=Rohan, 3=Mohit, 5=Jagjit, 4=Chirag, 9=Arun, 7=John}
Printing the TreeMap: {1=Rohit, 2=Rohan, 3=Mohit, 4=Chirag, 5=Jagjit, 7=John, 9=Arun}
```

E) Comparator

Classes planning to provide sorting technique upon its objects need to implement Comparator interface. public int compare(Object o1, Object o2) method needs to be concretized. This method returns 0 if comparing values are same, 1 if first value is greater than second and -1 otherwise.

```
class Student{
int rollno;
String name;
int age;
Student(int rollno,String name,int age){
this.rollno=rollno;
this.name=name;
this.age=age;
}
}
```

```
import java.util.*;
class AgeComparator implements Comparator{
public int compare(Object o1,Object o2){
Student s1=(Student)o1;
Student s2=(Student)o2;
if(s1.age==s2.age)
return 0;
else if(s1.age>s2.age)
return 1;
else
```

	return -1;
	}
	}

import java.util.Comparator;
 	
public class NameComparator implements Comparator{
    public int compare(Object o1, Object o2) {
        Student s1=(Student)o1;
        Student s2=(Student)o2;
    return s1.name.compareTo(s2.name);
    }
}
	import java.util.*;
	import java.io.*;
	class Simple{
	public static void main(String args[]){
	ArrayList al=new ArrayList();
	al.add(new Student(101,"Vijay",23));
	al.add(new Student(106,"Ajay",27));
	al.add(new Student(105,"Jai",21));
	System.out.println("Sorting by Name...");
	Collections.sort(al,new NameComparator());
	Iterator itr=al.iterator();
	while(itr.hasNext()){
	Student st=(Student)itr.next();
	System.out.println(st.rollno+" "+st.name+" "+st.age);
	}
	System.out.println("sorting by age...");
	Collections.sort(al,new AgeComparator());
	Iterator itr2=al.iterator();
	while(itr2.hasNext()){
	Student st=(Student)itr2.next();
	System.out.println(st.rollno+" "+st.name+" "+st.age);
	}
	}
	}

Output

```
Sorting by Name...
106 Ajay 27
105 Jai 21
101 Vijay 23
sorting by age...
105 Jai 21
101 Vijay 23
106 Ajay 27
```

Chapter 7: Multi Threading

A) Processes and Threads:

- ✓ *Process:* A set of task being executed by the processor. *e.g.* FireFox
- ✓ *Thread:* A subset or an independent instance of the process. A smallest dispatchable unit of work. *e.g.* Different Tabs in FireFox

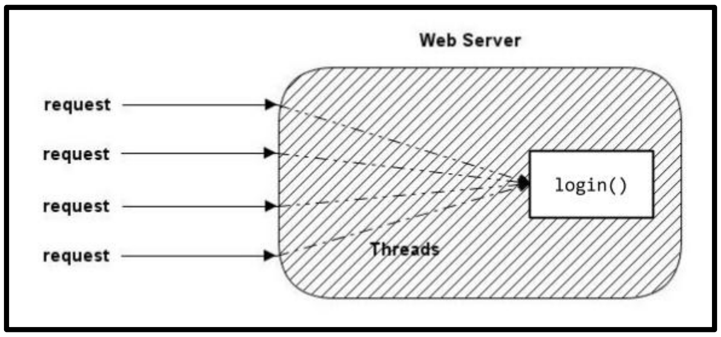
A computer system normally has many active processes and threads. This is true even in systems that only have a single execution processor core, and thus only have one thread actually executing at any given moment. Processing time for a single core is shared among processes and threads through an OS feature called time slicing/sharing. It's becoming more and more common for computer systems to have multiple processors or processors with multiple execution cores. This greatly enhances a system's capacity for concurrent execution of processes and threads — but concurrency is possible even on simple systems, without multiple processors or execution cores with the help of multiprocessing/multithreading environments.

Process

A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space. Processes are often seen as programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support Inter Process Communication (IPC) resources. Most implementations of the Java virtual machine run as a single process. A Java application can create additional processes.

Threads

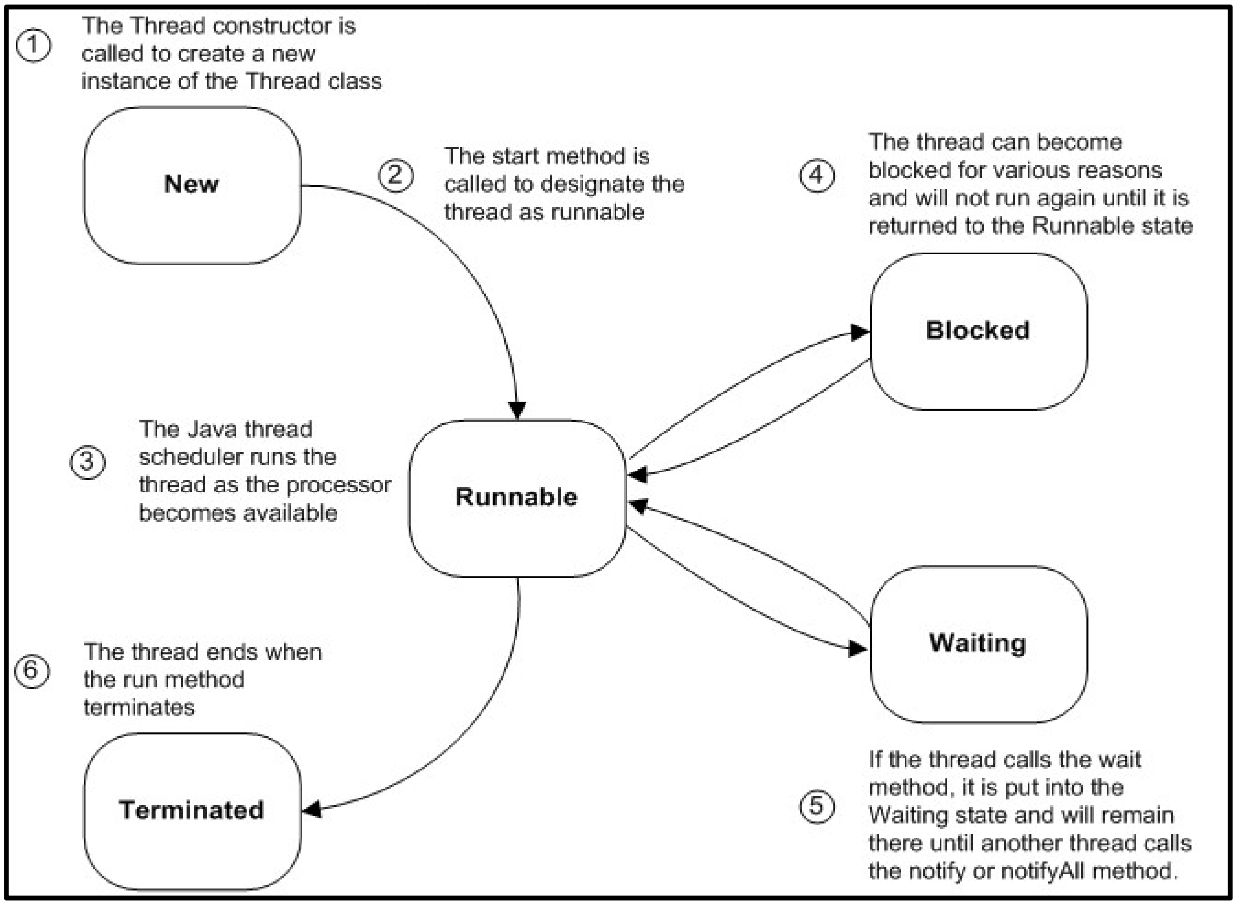
Threads are sometimes called lightweight processes. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process. Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.



Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling. You start with just one thread, called the main thread. This thread has the ability to create additional threads.

E.g. There a lot of users who want to login into Google account. But we have just one login method which checks the logic for authentication. Here, we can create multiple threads of the same login method per user request.

Thread Life cycle


① The Thread constructor is called to create a new instance of the Thread class
New
② The start method is called to designate the thread as runnable
④ The thread can become blocked for various reasons and will not run again until it is returned to the Runnable state
Blocked
③ The Java thread scheduler runs the thread as the processor becomes available
Runnable
Waiting
⑥ The thread ends when the run method terminates
Terminated
⑤ If the thread calls the wait method, it is put into the Waiting state and will remain there until another thread calls the notify or notifyAll method.

B) Creating Threads in Java

Thread class and Runnable interface are used to create threads in Java. Both are situated in java.lang package. Thread class extends the Object class. It has many useful methods which can be used for Thread handling. Runnable interface has only one method prototype public void run() which needs to be implemented by the concrete classes.

There are two options how we can create threads in java:

1. **Extending the Thread class:** Extend Thread class and override the public void run() method in the child class. Use the start() method of Thread class to start the execution of the run() method. start() calls run() implicitly. If run() is invoked, the overriden run() is called thus threading is not implemented.

class ThreadA extends Thread{
public void run(){
for(int i=1;i<=10;i++)
        System.out.println("Thread A:" + i);
}
}
	class ThreadB extends Thread{
	public void run(){
	for(int i=1;i<=10;i++)
	        System.out.println("Thread B:" + i);
	}
	}

class TestThreads{
public static void main(String args[]){
ThreadA ta = new ThreadA();
ThreadB tb = new ThreadB();
ta.start();
	Output:
	Thread A:1
	Thread A:2
	Thread A:3
	Thread A:4

```
tb.start();
}
}
```

```
Thread A:5
Thread B:1
Thread B:2
Thread B:3
Thread B:4
Thread B:5
Thread A:6
Thread A:7
Thread A:8
Thread A:9
Thread A:10
Thread B:6
Thread B:7
Thread B:8
Thread B:9
Thread B:10
```

The output may vary from machine to machine depending on the execution load and the hardware/software configuration.

2. **Implementing the Runnable interface:** Implement the Runnable interface and override the public void run() method. As we are not extending the Thread class, we don't have access to the start() method. So create an object of the Thread class and pass the object of the class implementing the run() method inside the parameterized constructor. Then call the start() with the help of Thread class object.

```
class ThreadA implements Runnable{
public void run(){
for(int i=1;i<=10;i++)
System.out.println("Thread A:" + i);
} }
```

```
class ThreadB implements Runnable{
public void run(){
for(int i=1;i<=10;i++)
        System.out.println("Thread B:" + i);
} }
```

```
class TestThreads{
public static void main(String args[]){
        ThreadA ta = new ThreadA();
        ThreadB tb = new ThreadB();
        Thread  t1 = new Thread(ta);
        Thread  t2 = new Thread(tb);
        t1.start();
        t2.start();
}
}
```

Output:

```
Thread A:1
Thread B:1
Thread B:2
Thread B:3
Thread A:2
Thread A:3
Thread A:4
Thread B:4
Thread B:5
Thread A:5
Thread A:6
Thread A:7
Thread A:8
Thread A:9
Thread A:10
Thread B:6
Thread B:7
Thread B:8
Thread B:9
```

C) Thread Class methods

Sr. no.	Method	Description
1	void start()	Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
2	void destroy()	Destroy this thread without any cleanup
3	void stop()	Causes it to unlock all of the monitors that it has locked
4	void suspend()	Thread is blocked temporarily, no thread can access this resource until the thread is resumed.
5	void resume()	This method exists solely for use with suspend() to resume the suspended thread.
6	static void yield()	A hint to the scheduler that the current thread is willing to yield its current use of a processor.
7	static void sleep(long millis)	Causes the currently executing thread to sleep (temporarily stop execution) for the specified number of milliseconds.
8	void setPriority(int newPriority)	Changes the priority of this thread.
9	int getPriority()	Returns this thread's priority.
10	String getName()	Returns this thread's name.
11	void join()	Waits for this thread to die.
12	void interrupt()	Interrupts this thread.

D) Thread Priorities:

We can set the priorities of the threads by using **setPriority**(*int newPriority*) method.

3 constants defined in Thread class:

- ✓ public static int MIN_PRIORITY :- 1
- ✓ public static int NORM_PRIORITY :-10
- ✓ public static int MAX_PRIORITY :- 5

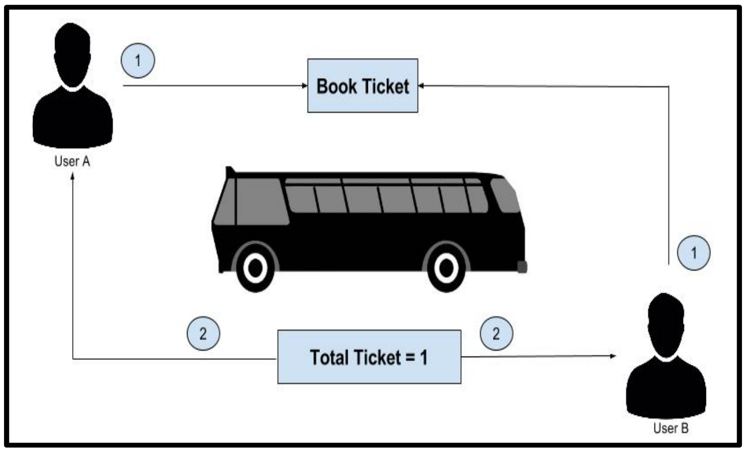
```
class TestMultiPriority extends Thread{
public void run(){
System.out.println("running thread priority
is:"+Thread.currentThread().getPriority());
}
```

```
 public static void main(String args[]){
 TestMultiPriority m1=new TestMultiPriority();
 TestMultiPriority m2=new TestMultiPriority();
 m1.setPriority(Thread.MIN_PRIORITY);
 m2.setPriority(Thread.MAX_PRIORITY);
 m1.start();
 m2.start();
} }
```

Output:

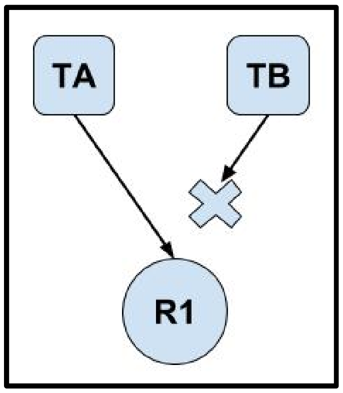
```
running thread priority is:10

running thread priority is:1
```

E) Thread Synchronization



User A and User B both want to book a ticket, but there is just one ticket left. Either one of the two should get a message stating tickets unavailable. As both users make requests, two threads are created. What may happen is, if the Thread A gets switched and B gets executed just before A checks the seat availability, both the users will be returned confirmation of the ticket! To avoid this from happening, we must implement Thread safety.

Thread safety usually refers to Mutual Exclusion meaning in times of a critical resource being shared, if a process/thread holds access to the resource, other thread will be denied access to the resource until previous thread completes execution.



But if the thread holding access to the resource gets blocked, it directly affects the performance of other threads wanting access to the same resource as they have to wait until the previous resource doesn't leave the resource.

We have a BookTicket class which has total ticket count as c. If a user thread wants to book the ticket, he invokes printTicket() method which decrements the ticket count. If the threads are not synchronized, the count may be displayed incorrectly to the users. Even after the tickets are exhausted, users can be returned with a positive ticket confirmation! This happens because we have not treated the threads safely. If we want to implement thread safety, we take help of synchronized keyword.

Thread Unsafe Implementation:

```
public class BookTicket extends Thread{
static int c=10;
public void run(){
    printTicket();
}
static void printTicket(){
        c--;
        if(c<0)
System.out.println("ticket not available for customer");
        else
System.out.println("ticket available for customer: "+c);
        }
}
```

```
public class TestMain{
public static void main(String[] args){
BookTicket bt[]=new BookTicket[11];
for(int i=0;i<bt.length;i++){
bt[i] = new BookTicket();
bt[i].start();
}
}
}
```

Output:

```
ticket available for customer:10
ticket available for customer:9
ticket available for customer:8
ticket available for customer:8
ticket available for customer:6
ticket available for customer:4
ticket available for customer:5
ticket available for customer:3
ticket available for customer:3
ticket available for customer:1
ticket available for customer:1
```

Synchronized keyword: used to implement thread safety (*usage*: methods and blocks)

1. It is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

2. When a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Thread Safe Implementation:

```java
public class BookTicket extends Thread{
static int c=10;
synchronized public void run(){
    printTicket();
}
synchronized static void printTicket(){
        c--;
        if(c<0)
System.out.println("ticket not available for customer");
        else
System.out.println("ticket available for customer: "+c);
        }
}
```

```java
public class TestMain{
public static void main(String[] args){
BookTicket bt[]=new BookTicket[11];
for(int i=0;i<bt.length;i++){
bt[i] = new BookTicket();
bt[i].start();
}
}
}
```

Output:

```
ticket available for customer:10
ticket available for customer:9
ticket available for customer:8
ticket available for customer:7
ticket available for customer:6
ticket available for customer:5
ticket available for customer:4
ticket available for customer:3
ticket available for customer:2
ticket available for customer:1
ticket not available for customer
```

Synchronized blocks of code

When we don't want the entire method to be thread safe, rather a block of code to implement thread safety, we use Synchronized blocks of code:

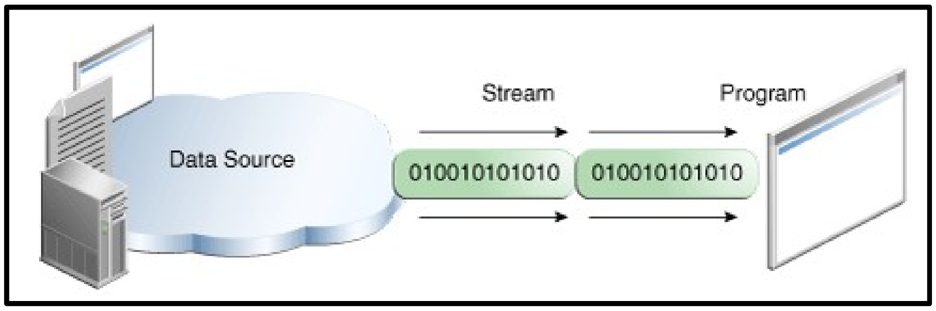
```
void someMethod(){

synchronized(Object o) {

        //logic

        }

}
```

Chapter 8: Files and I/O

A) What is a Stream ?

An I/O Stream represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays. Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways.

"A stream is a sequence of data. A program uses an inputstream to read data from a source, one item at a time"



Java classification of character and byte streams

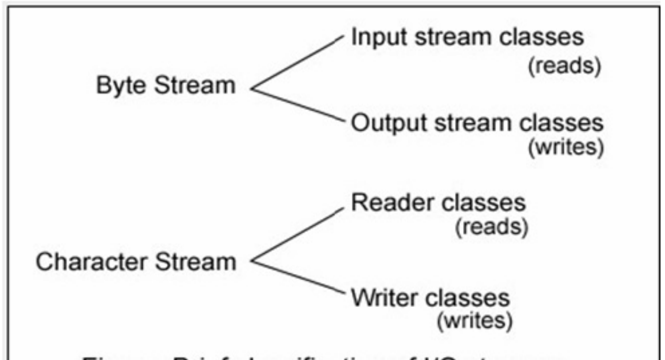


Figure: Brief classification of I/O streams

File Class

```
import java.io.File;
public class FileDetails {
	public static void main(String[] args) {
	File file = new File("E:\\input.txt");
System.out.println("File exists: "+file.exists());
System.out.println("File Path: " + file.getAbsolutePath());
System.out.println("File readable: " + file.canRead());
System.out.println("File writable: " +file.canWrite());
System.out.println("File executable: " +file.canExecute());
System.out.println("File length: " +file.length());
	}
}
```

Output

```
File exists: true
File Path: E:\input.txt
File readable: true
File writable: true
File executable: true
File length: 22
```

B) Reading/Writing using a Byte Stream

Reading a file using byte stream

```
import java.io.FileInputStream;
import java.io.IOException;
public class ReadFileByte {
public static void main(String[] args) throws
IOException {
FileInputStream fis = new
FileInputStream("E:\\input.txt");
int x=0;
while((x=fis.read())!=-1){
System.out.print((char)x);
            }
      }
}
```

Output:

```
This is a sample file.
```

Writing a file using byte stream

```
import java.io.FileOutputStream;
import java.io.IOException;
public class WriteFileByte {
public static void main(String[] args) throws
IOException{
FileOutputStream fos = new
FileOutputStream("E:\\output.txt");
String s = "This is an output message.";
for(int i=0;i<s.length();i++)
fos.write(s.charAt(i));
      }
}
```

Output: *(Will be written in the file)*

```
This is an output message.
```

C) Reading /Writing using Character Stream

Reading a file using byte stream

```
import java.io.FileReader;
import java.io.IOException;
public class ReadFileChar {
public static void main(String[] args) throws
IOException {
FileReader fr = new FileReader("E:\\input.txt");
int x=0;
while((x=fr.read())!=-1){
System.out.print((char)x);
            }
      }
}
```

Output:

```
This is a sample file.
```

Writing a file using character stream

```
import java.io.FileWriter;
import java.io.IOException;
public class WriteFileChar {
public static void main(String[] args) throws
IOException {
FileWriter fw = new FileWriter("E:\\output.txt");
fw.write("This is an output file.");
fw.close();
      }
}
```

Output:

```
This is an output file.
```

D) Serialization

Serialization in java is a mechanism of writing the state (members) of an object into a byte stream, usually files. Re Converting an object from previously written file is called as De Serialization. ObjectInputStream and ObjectOutputStream classes are used. The class having the serialization property must implement Serializable interface.

- ✓ *ObjectInputStream* class has a writeObject(Object o) method to write the object state to the file.
- ✓ *ObjectOutputStream* class has a readObject() method to read the object from the file.

Serialize:

```
import java.io.Serializable;
public class Employee implements Serializable{
        int id;
        String name;
        float salary;
        public Employee() {    }
        public Employee(int id, String name,
float salary) {
                this.id = id;
                this.name = name;
                this.salary = salary;
        }
        public void print(){
        System.out.println("Id: " + id);
        System.out.println("Name: " + name);
        System.out.println("Salary: " + salary);
        }
}
```

```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
public class Serialize {
public static void main(String[] args) throws Exception
{
Employee e = new Employee(1,"John",12345.2f);
FileOutputStream fos = new FileOutputStream
("/home/administrator/Documents/serial");
ObjectOutputStream oos = new
ObjectOutputStream(fos);
System.out.println("Writing object's state in the file");
                oos.writeObject(e);
                fos.close();
                oos.close();
        }
}
```

Deserialize:

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
public class DeSerialize {
public static void main(String[] args) throws Exception {
FileInputStream fis = new FileInputStream
("/home/administrator/Documents/serial");
ObjectInputStream ois = new ObjectInputStream(fis);
System.out.println("Reading object's state from the file");
Employee e = (Employee)ois.readObject();
                e.print();
                fis.close();
                ois.close();
        }
}
```

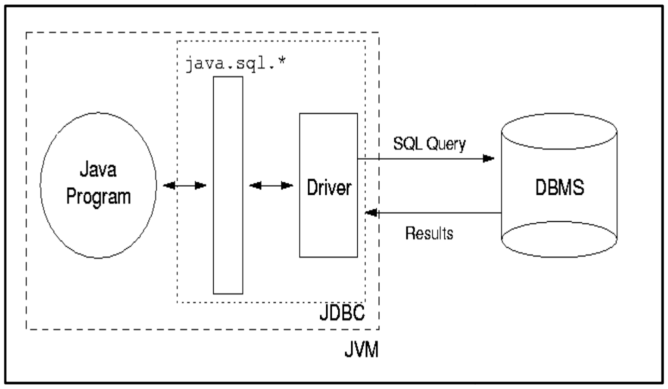
Output

```
Reading object's state from the file
Id: 1
Name: John
Salary: 12345.2
```

Chapter 9: JDBC

Java Database Connectivity (JDBC) is an application program interface (API) specification for connecting programs written in Java to the data in popular databases. The application program interface lets you encode access request statements in Structured Query Language (SQL) that are then passed to the program that manages the database. It returns the results through a similar interface. JDBC is very similar to the SQL Access Group's Open Database Connectivity (ODBC) and, with a small "bridge" program, you can use the JDBC interface to access databases through the ODBC interface.

A) JDBC Architecture



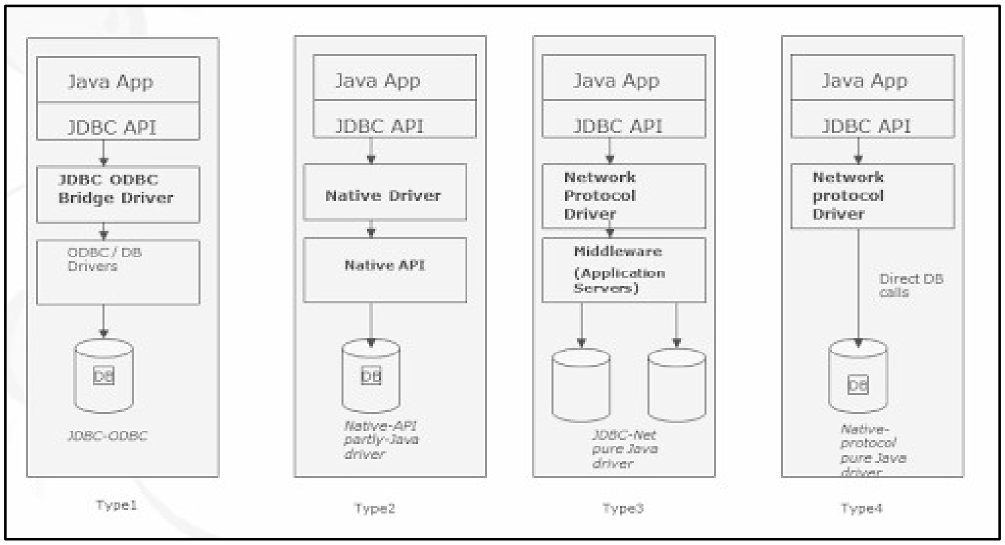
The Java program uses the JDBC API to embed logic with the SQL queries. With the help of an appropriate driver, the queries are passed to the Database Engine which in turn returns the affected row count or selected column data back to Java program. The program must have a presentation logic which displays the query output to the user.

B) JDBC Drivers

Why are Drivers needed ?

JDBC doesn't focus only on one kind of Database Vendor. Thus, every Database like Oracle, MS-SQL, MySql have some differences like data types etc. Drivers are needed to make the program support the particular Database.

There are four types of Drivers out of which, most suitable driver should be used. We are going to use Type 1 driver in our eg. This driver is freely available on the internet as a .jar file and can be added into the build path of our application.



Type 1: JDBC ODBC Bridge Driver

In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Slowest type of driver. ODBC conversion overhead increases time needed for the operation.

Type 2: JDBC-Native API

In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.

Type 3: JDBC-Net pure Java

In a Type 3 driver, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server.

Type 4: 100% Pure Java

In a Type 4 driver, a pure Java-based driver communicates directly with the vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.

C) Steps involved in JDBC

1. **Load the driver**

Class.forName ("com.mysql.jdbc.Driver");

System.out.println("Driver is loaded");

2. **Establish a connection.**

Connection con= DriverManager.getConnection ("jdbc:mysql://localhost/test","root","");

System.out.println("Connection is created");

3. **Create a statement.**

Statement st = con.createStatement();

4. **Execute the query.**

String q = "insert into employee values(1,'Lio',11656.9)";

int i = st.executeUpdate(q);

5. **Display the output.**

System.out.println("Rows affected : "+i);

D) CRUD operations on a table

We are using **MySql** as our database. We have created a database schema ***test*** using:

create database test;

use test;

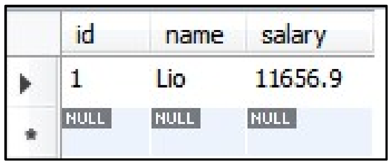
Create a table employee having *id,name* and *salary* using:

create table employee(id integer,name varchar(20),salary float);

i. Inserting a row

```
import java.sql.*;
public class Insert {
public static void main(String[] args) {
try {
Class.forName ("com.mysql.jdbc.Driver");
System.out.println("Driver is loaded");
Connection
con=DriverManager.getConnection("jdbc:mysql:/
/localhost/test","root","root");
System.out.println("Connection is created");
String    q    =    "insert    into    employee
values(1,'Lio',11656.9)";

Statement st = con.createStatement();
int i = st.executeUpdate(q);
System.out.println("Rows affected : "+i);
}
catch (Exception e) {
e.printStackTrace();
                }
        }
}
```

Output

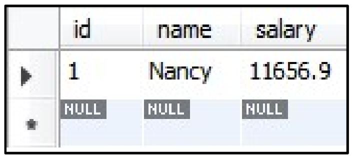


	id	name	salary
▶	1	Lio	11656.9
*	NULL	NULL	NULL

ii. Updating a row

```
import java.sql.*;
public class Update {
public static void main(String[] args) {
try {
Class.forName ("com.mysql.jdbc.Driver");
System.out.println("Driver is loaded");
Connection
con=DriverManager.getConnection("jdbc:mysql:/
/localhost/test","root","root");
System.out.println("Connection is created");
String  q  =  "update  employee  set  name='Nancy'
where id=1";
Statement st = con.createStatement();
int i = st.executeUpdate(q);
System.out.println("Rows affected : "+i);
}
catch (Exception e) {
e.printStackTrace();
        }
        }
}
```

Output

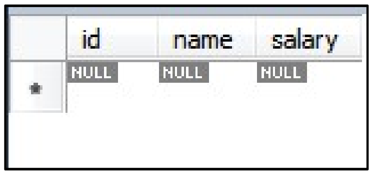


	id	name	salary
▶	1	Nancy	11656.9
*	NULL	NULL	NULL

iii. Deleting a row

```
import java.sql.*;
public class Delete {
public static void main(String[] args) {
try {
Class.forName ("com.mysql.jdbc.Driver");
System.out.println("Driver is loaded");
Connection
con=DriverManager.getConnection("jdbc:mysql:/
/localhost/test","root","root");
System.out.println("Connection is created");
String q = "delete from employee where id=1";
Statement st = con.createStatement();
int i = st.executeUpdate(q);
System.out.println("Rows affected : "+i);
                    }
                 catch (Exception e) {

                         e.printStackTrace();
                 }
         }
}
```

Output

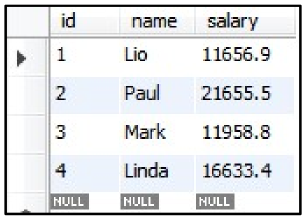


	id	name	salary
*	NULL	NULL	NULL

iv. Selecting rows

```
import java.sql.*;
public class Select {
public static void main(String[] args) {
try {
Class.forName ("com.mysql.jdbc.Driver");
System.out.println("Driver is loaded");
Connection
con=DriverManager.getConnection("jdbc:mysql:/
/localhost/test","root","root");
System.out.println("Connection is created");
String q = "select * from employee";
Statement st = con.createStatement();
ResultSet rs = st.executeQuery(q);
System.out.println("Id\tName\tSalary");
while(rs.next()){
System.out.println(rs.getString(1)+"\t"+rs.getStri
ng(2)+"\t"+rs.getString(3));
                }
}
catch (Exception e) {
e.printStackTrace();
                }
        }
}
```

Output



	id	name	salary
▶	1	Lio	11656.9
	2	Paul	21655.5
	3	Mark	11958.8
	4	Linda	16633.4
	NULL	NULL	NULL

PreparedStatement

If we want to pass dynamic values to a query, statement object is not useful. In such scenarios we use a PreparedStatement.

```
import java.sql.*;
import java.util.Scanner;
public class UserInsert {
public static void main(String[] args) {
try {
Class.forName ("com.mysql.jdbc.Driver");
System.out.println("Driver is loaded");
Connection
con=DriverManager.getConnection("jdbc:mysql:/
/localhost/test","root","root");
System.out.println("Connection is created");
Scanner scan = new Scanner(System.in);
int id=0;
String name=null;
float salary=0.0f;
System.out.println("Enter id:");
id=scan.nextInt();
System.out.println("Enter name:");
name=scan.next();
System.out.println("Enter salary:");
salary=scan.nextFloat();
String q = "insert into employee values(?,?,?)";
PreparedStatement pst =
con.prepareStatement(q);
pst.setInt(1,id);
pst.setString(2,name);
pst.setFloat(3,salary);
int i = pst.executeUpdate();
System.out.println("Rows affected : "+i);
}
catch (Exception e) {
        e.printStackTrace();
                }
        }
}
```

Explanation

String q = "insert into employee values(?,?,?)";

'?' represents a temporary placeholder for the future value.

pst.setInt(1,id); : 1 means the value of id will be substituted at the 1st '?' thats why its set 'Int'

E) Stored Procedures

It is a bad idea to use singular queries to manipulate data. It is better to create procedures containing the queries.

Let us create a procedure which deletes a row of the table whose id is 1.

```
USE `test`;
DROP procedure IF EXISTS `deleteRow`;
DELIMITER $$
USE `test`$$
CREATE PROCEDURE `deleteRow` ()
BEGIN
delete from employee where id = 1;
END$$
DELIMITER ;
```

We use an object of CallableStatement in order to fire the procedure.

```
import java.sql.*;
public class DeleteProcedure {
public static void main(String[] args) {
try {
Class.forName ("com.mysql.jdbc.Driver");
System.out.println("Driver is loaded");
Connection
con=DriverManager.getConnection("jdbc:mysql:
//localhost/test","root","root");
System.out.println("Connection is created");
String q = "call deleteRow()";
CallableStatement st = con.prepareCall(q);
int i = st.executeUpdate(q);
System.out.println("Rows affected : "+i);
}
catch (Exception e) {
e.printStackTrace();
}
        }
}
```

Parameters in procedures

There are three types of parameters:

i. **IN**: The data comes in to the Database
ii. **OUT**: The data goes out of the Database
iii. **INOUT**: The data comes in, gets updated and then goes out of the Database

Parameterized procedure for deleting a row:

```
CREATE DEFINER=`root`@`localhost` PROCEDURE
`deleteRowParameter`(IN id integer)
BEGIN
delete from employee where employee.id=id;
END
```

Java Program:

```
import java.sql.*;
import java.util.Scanner;
public class DeleteParameterProcedure {
public static void main(String[] args) {
try {
Class.forName ("com.mysql.jdbc.Driver");
System.out.println("Driver is loaded");
Connection con=DriverManager.getConnection
("jdbc:mysql://localhost/test","root","root");
System.out.println("Connection is created");
String q = "call deleteRowParameter(?)";
CallableStatement cst = con.prepareCall(q);
System.out.println("Enter id to delete the row");
Scanner scan = new Scanner(System.in);
cst.setInt(1,scan.nextInt());
int i = cst.executeUpdate();
System.out.println("Rows affected : "+i);
}
catch (Exception e) {
        e.printStackTrace();
                }
        }
}
```

Parameterized procedure for updating a row:

```
CREATE DEFINER=`root`@`localhost` PROCEDURE
`updateRowParameter`(IN id integer,INOUT
name varchar(20),INOUT salary float)
BEGIN
update employee set
employee.id=id,employee.name=name,employee
.salary=salary;
END
```

Java Program:

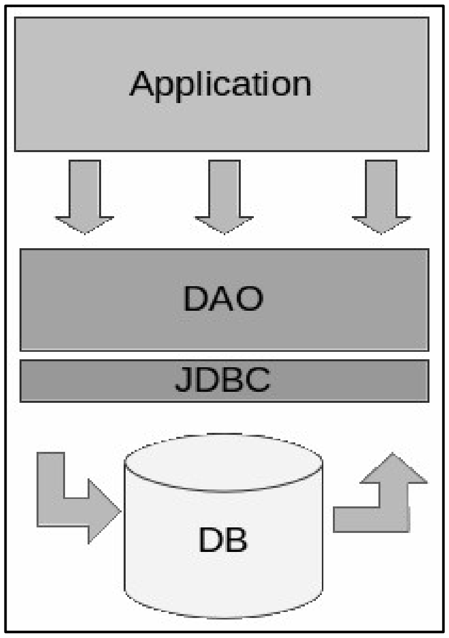
```
import java.sql.*;

import java.util.Scanner;
public class UpdateRowParameter {
public static void main(String[] args) {
try {
Class.forName ("com.mysql.jdbc.Driver");
System.out.println("Driver is loaded");
Connection
con=DriverManager.getConnection("jdbc:mysql:/
/localhost/test","root","root");
System.out.println("Connection is created");
Scanner scan = new Scanner(System.in);
String q = "call updateRowParameter(?,?,?)";
CallableStatement cst = con.prepareCall(q);
System.out.println("Enter id to update the
row");
cst.setInt(1,scan.nextInt());
System.out.println("Enter new name and
salary");
cst.setString(2,scan.next());
cst.setFloat(3,scan.nextFloat());
cst.registerOutParameter(2, Types.VARCHAR);
cst.registerOutParameter(3, Types.FLOAT);
int i = cst.executeUpdate();
System.out.println("Rows affected : "+i);
System.out.println("New Name:
"+cst.getString(2));
System.out.println("New Salary:
"+cst.getFloat(3));
}
catch (Exception e) {
e.printStackTrace();
                }
        }
}
```

Output

```
Driver is loaded

Connection is created

Enter id to update the row

1

Enter new name and salary

johnny

132456

Rows affected : 1

New Name: johnny

New Salary: 132456.0
```

F) DAO Design Pattern

It is tedious and illogical to load the driver and create a connection every time for each query. Instead create a separate class for doing the same; create classes which operate on queries using the connector class. This pattern is called as DAO or Data Access Object. It is also called as Facade pattern



- ✓ DAO facilitates a naive programmer to understand the underlying complex operations in a simple manner.
- ✓ It also allows reusability via logics stored in the methods.
- ✓ Faster performance as driver and connections are created as per optimum requirements.
- ✓ Concepts of singleton can be easily applied on drivers and connections.