Core Java Language Fundamentals

Xoriant

Contents

Module	Topic
Module 1	Introduction to Java
Module 2	Classes & objects
Module 3	OOP concepts in Java
Module 4	Inheritance & Polymorphism
Module 5	Interfaces
Module 6	Packages
Module 7	Wrapper classes
Module 8	String, StringBuffer & StringBuilder
Module 9	Reflection
Module 10	Annotations
Module 11	Enums
Module 12	Nested classes

Introduction to Java

Java History

- In 1991, a small group of Sun Microsystem engineers called the 'Green Team' was formed. 'Green Team' was led by James Gosling.

- 'Green Team' initiated a revolutionary task to develop a language for controlling consumer devices such as set-top boxes, televisions etc.

- Later on it was found that this language is suited for internet programming.

- Earlier name of this language was 'Greentalk', then changed to 'Oak' & finally in 1995, it was named as 'Java'.

- 'Java' is an island of Indonesia where first coffee was produced (called java coffee).

Java Version History

JDK Alpha and Beta (1995)

JDK 1.0 (23rd Jan, 1996)

JDK 1.1 (19th Feb, 1997)

J2SE 1.2 (8th Dec, 1998)

J2SE 1.3 (8th May, 2000)

J2SE 1.4 (6th Feb, 2002)

J2SE 5.0 (30th Sep, 2004)

Java SE 6 (11th Dec, 2006)

Java SE 7 (28th July, 2011)

Java SE 8 (18th March, 2014)

Java Features

- Simple
- Object-Oriented
- Platform independent
- Secure
- Robust
- Architecture neutral
- Portable
- Dynamic
- Multithreaded
- Distributed

Java Features continue…

- *Simple*

  Java syntax is based upon C++. However, java has removed several complex topics from C++ like pointers, virtual base class, friend functions, structures, operator overloading, private inheritance etc.

- *Object-Oriented*

  Java is an object oriented language. It supports basic pillars of OOP like class, object, data abstraction, encapsulation, inheritance & polymorphism.

- *Platform independent*

  - ✓ Compiling java source file generates .class file, also called as byte code.
  - ✓ The byte code is a platform independent code. It means that the same byte code can run on any platform like windows, linux, mac etc.
  - ✓ Java provides JVM (Java Virtual Machine) which is an abstract computing machine that enables a computer to run a Java program.
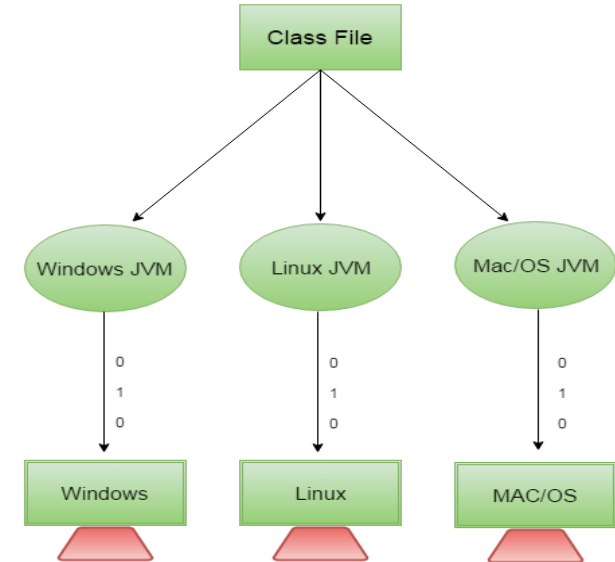
- *Secure*
  - ✓ Java is more secure than C or C++ because Java does not support explicit pointer.
  - ✓ When we run a java program, it is loaded by classloader then the byte code verified by 'byte code verifier' & finally 'security manager' checks the security of the program.

- *Robust*

  Robust language is a language where the program withstands in worst scenario. Java is robust because it uses strong memory management. Java developer can only allocate memory but memory deallocation is taken care by java using 'garbage collector (GC)'.

- *Architecture neutral*

  Java provides fix size of every primitive. For example integer takes 4 bytes irrespective of operating system or underlying hardware architecture.

- *Portable*

  Java is portable because we can run byte code on any machine & any operating system. Also, memory consumption is same on all platforms since primitive data size is platform independent.

8

Java Features continue…

- *Dynamic*

  In Java, all methods are by default virtual & it makes java a dynamic language. Java dynamically allocates memory for a method only when it is called.

- *Multithreaded*

  Java provides us rich APIs to work on multithreaded application. Using java APIs, we can create a new thread, control its priority, apply synchronization etc.

- *Distributed*

  Distributed application provides us communication between two java processes. We can create a distributed application in Java using RMI (Remote Method Invocation).

JVM (Java Virtual Machine)

JVM (Java Virtual Machine)
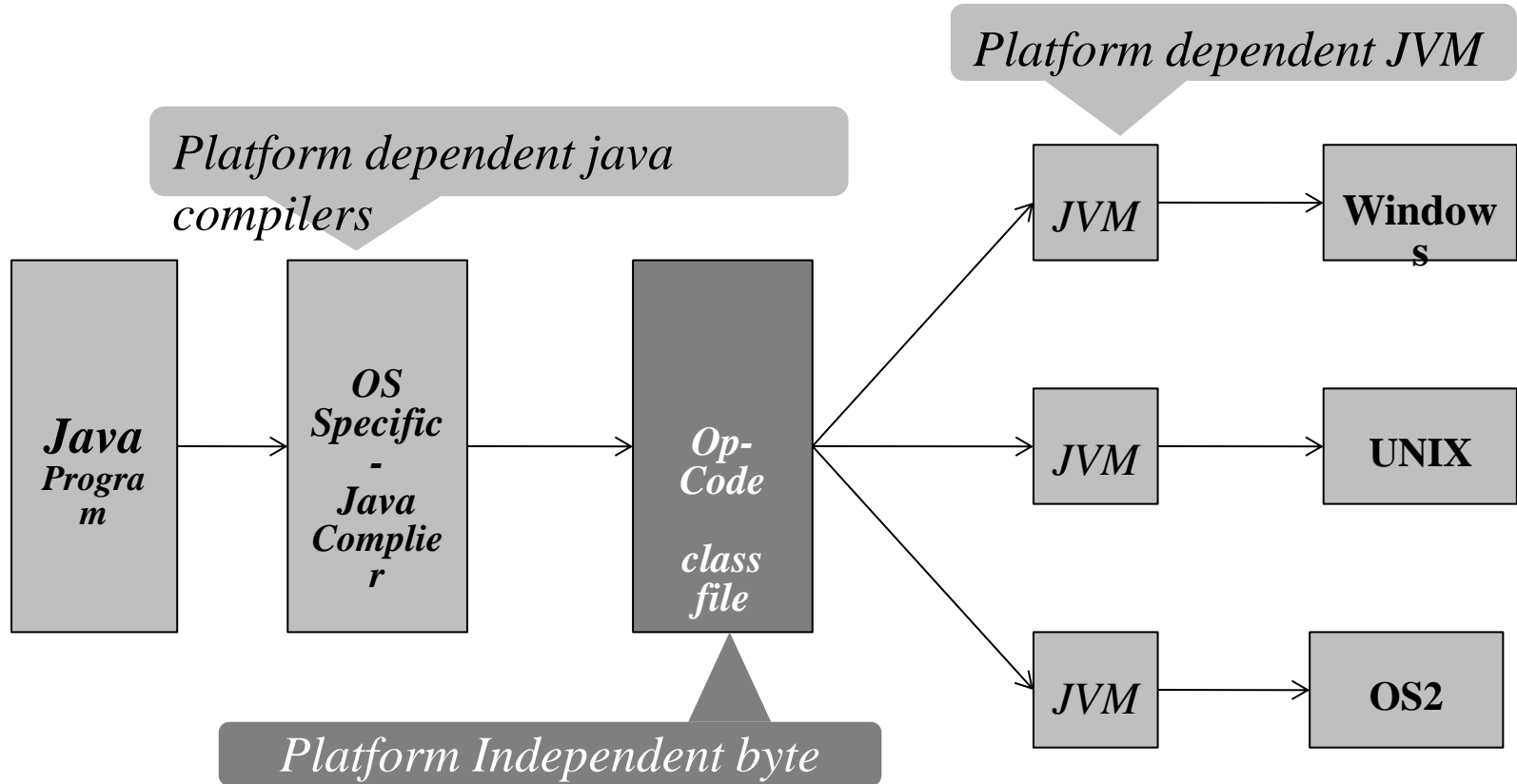
- JVM is an abstract computing machine that enables a computer to run a Java program.

- JVM is a specification. You can find different implementations of JVM i.e. Oracle's Hotspot, BEA System's JRockit etc.

- JVM implementation is platform dependent.

- JVM provides Java its cross-platform capabilities.

- JVM consists of JIT (Just-in-time) compiler & interpreter. JIT compiler compiles byte code into the instructions understood to underlying platform where as interpreter interprets them one by one.
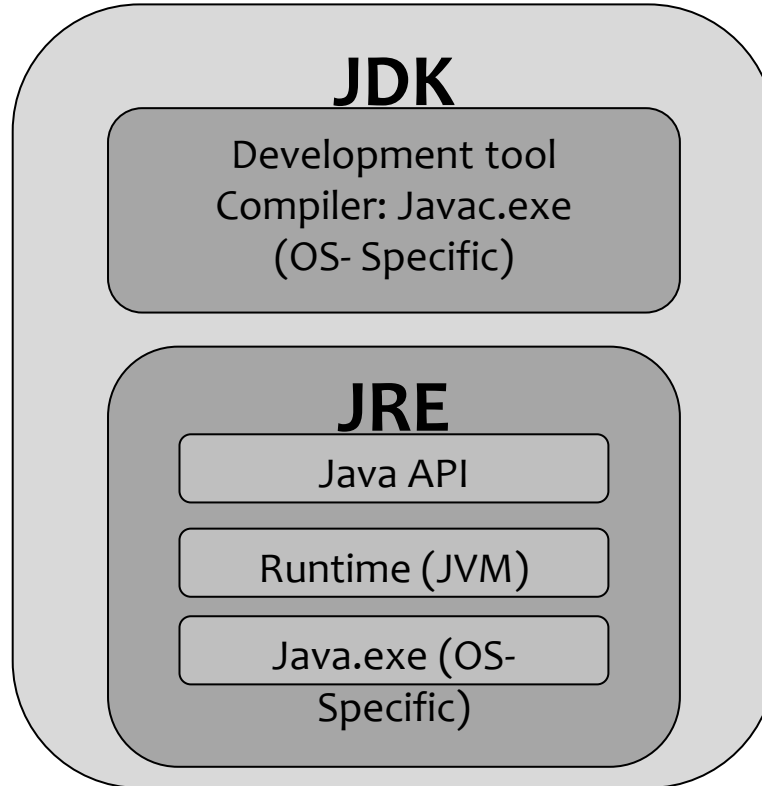
IDE (Integrated Development Environment)

Popular IDEs
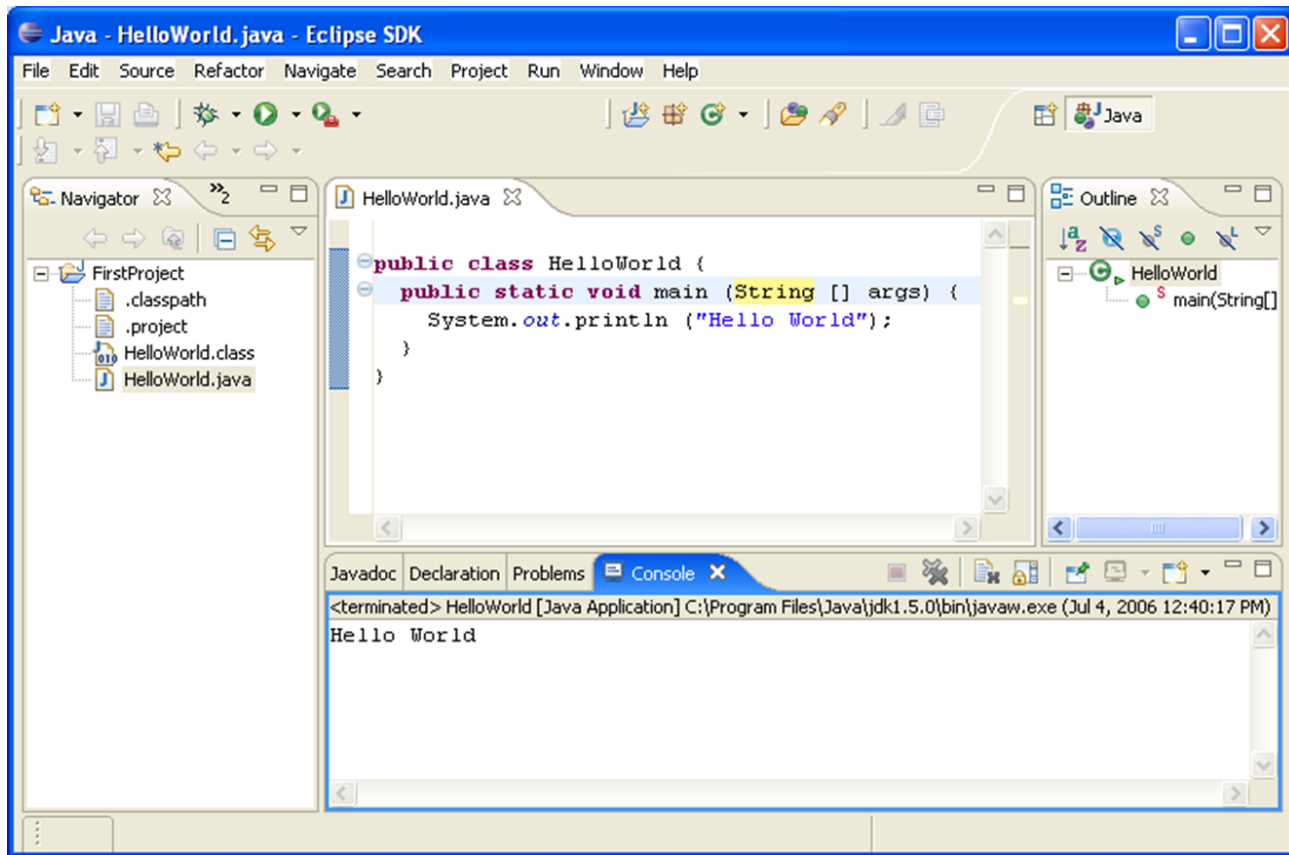
- NetBeans
- Eclipse
- IntelliJ
- jDeveloper

```java
public class HelloWorld {

        public static void main(String args[]) {

                System.out.println("Hello World!!");

        }

}
```

Note:

The class in which we write main() method, that class name should be given to the name of the file. For example: HelloWorld.java

A java source file can have maximum only one public class.

Compiling & running a program

- Set the environment variable PATH to Java's bin directory.
- To compile, at the command prompt, type:

   javac –d . HelloWorld.java
- If there are no errors, there should be a file called **HelloWorld.class** in your working directory.
- To run the program, at the command prompt, type:

   java HelloWorld

Naming conventions in Java

Classes:

First character of every word should be capital while other characters should be small. Need not use an Underscore. Ex : class HelloWorld.

Variables or Methods:

First character of every word except first word should be capital while other characters should be small. Need not use Underscore. Ex: method:- void showData( ){ } variable:- int empNo = 100;

Packages:

All characters should be small. Ex: package threading;

Constants:

All characters should be capital. Ex : float PI = 3.142;

Java Data Types

Primitive data types	User defined data types
byte	class
short	enum
int	Array
long	
float	
double	
char	
boolean	

Primitive Data Types

byte	8-bit integer (signed) Size (-27 to 27-1)
short	16-bit integer (signed) Size (-215 to 215 -1)
int	32-bit integer (signed) Size (-231 to 231 – 1)
Long	64-bit integer (signed) Size (-263 to 263 – 1)
Float	32-bit floating-point. Precision to 7-8th digit
Double	64-bit floating-point. Precision to16-18th digit
boolean	either true or false
char	16-bit Unicode

Comments in Java

```java
public class Temperature {
    /**  The program is written by…
    @ Author Ivan Ericsson
    */
    public static void main (String [ ] args) {
        //Variable declarations:

        /* double centigrade;
         double fahrenheit;
        */
        double centigrade;
        double fahrenheit;
        centigrade = 33.33;
        fahrenheit = (centigrade * 9 / 5) + 32;
        System.out.println(centigrade+" Centigrade = "               +fahrenheit + "
Fahrenheit.");
    }
}
```

documentation comment

Single line comment

Multiple line comment

Arithmetic Operators

+   addition

-       subtraction

*   multiplication

/   division

%   remainder

Unary minus (-) for negation

Unary plus (+).

Increment and Decrement Operators

```java
public class IncOp {
    public static void main(String[] args) {
        int i = 1;
        System.out.println (++i + " " + i++ + " " + i);
    }
}
```

What will the output of this program?

Assignment Operators

Let initial value of j be 15,

i = j;

i += 5;

i = i + 5;

i -= 5;

i *= 5;

i /= 5;

i %= 5;

What would be the final value for 'i'?

==   Comparison Operator

!=   Not Equal To Operator

< ,>   Greater than and Less than operators

<= , >=   Greater than or equal to / Less than or equal to

```java
public class Test {
    public static void main (String [ ] args) {
        if( getConnection() && openFile()){
            System.out.println("true ");
        }
        else{
            System.out.println("false ");
        }
    }
    public static boolean getConnection() {
        System.out.println("connecting...");
        return false;
    }
    public static boolean openFile(){
        System.out.println("opening file");
        return true;
    }
}
```

```
if (expression){
    statement1;
}
else {
    statement2;
}
```

```java
class Larger {
    public static void main (String [ ] args) {
        int a = 10;
        int b = 15;
        if (a > b){
            System.out.println ("A is larger.");
        }
        else{
            System.out.println ("B is larger.");
        }
    }
}
```

```java
class LargestOutOfThree {
    public static void main(String[] args) {
        int a = 25, b = 15, c = 10, largest;
        if (a > b) {
            if (a > c)
                largest = a;
            else
                largest = c;
        } else {
            if (b > c)
                largest = b;
            else
                largest = c;
        }
        System.out.println("Largest : " + largest);
    }
}
```

Xoriant Solution Pvt. Ltd.

```java
public class DayOfWeek {
    public static void main(String[] args) {
        int dayOfWeek = Integer.parseInt(args[0]);
        if (dayOfWeek == 0)
            System.out.println("Sunday");
        else if (dayOfWeek == 1)
            System.out.println("Monday");
        else if (dayOfWeek == 2)
            System.out.println("Tuesday");
        else if (dayOfWeek == 3)
            System.out.println("Wednesday");
        else if (dayOfWeek == 4)
            System.out.println("Thursday");
        else if (dayOfWeek == 5)
            System.out.println("Friday");
        else if (dayOfWeek == 6)
            System.out.println("Saturday");
    }
}
```

Xoriant Solution Pvt. Ltd.

```java
switch (dayOfWeek) {
    case 0: System.out.println ("Sunday");
    case 1: System.out.println ("Monday");
    case 2: System.out.println ("Tuesday");
    case 3: System.out.println ("Wednesday");
    case 4: System.out.println ("Thursday");
    case 5: System.out.println ("Friday");
    case 6: System.out.println ("Saturday");

    default : System.out.println ("ERROR!");
}
```

The switch Statement

```java
switch (dayOfWeek) {
  case 0: System.out.println ("Sunday"); break;
  case 1: System.out.println ("Monday");break;
  case 2: System.out.println ("Tuesday");break;
  case 3: System.out.println ("Wednesday");break;
  case 4: System.out.println ("Thursday");break;
  case 5: System.out.println ("Friday");break;
  case 6: System.out.println ("Saturday");break;

  default : System.out.println ("ERROR!");
}
```

The while Loop

```java
public class WhileLoop {
    public static void main(String[] args) {
        int i = 1;
        while (i <= 19) {
            System.out.println (i);
            i += 2;
        }
    }
}
```

```java
public class DoWhileLoop {

    public static void main(String[] args) {
        int i = 0;
        do {
        System.out.println (i);
                i += 2;
        } while (i <= 19);
    }

}
```

```java
public class ForLoop {

    public static void main(String[] args) {
        int i;
        for (i = 1; i<=10; i++){
            System.out.println (i);
        }
    }
}
```

break and continue Statements

```java
public class TestBreakContinue {

    public static void main(String[] args) {
        for (int i = 1; i <= 70; i++) {
            if (i % 7 == 0)
                continue;
            System.out.print(" " + i);

            if (i % 40 == 0) {
            System.out.println("Terminating Loop");
            break;
            }
        }
    }
}
```

Labeled break and continue Statements

```java
public class LabeledBreak {

    public static void main(String[] args) {
    outer:
        for (int i = 1; i <= 10; i++) {
            for (int j = 1; j <= 5; j++) {
                System.out.print("\t" + (i * j));
                if ((i * j) == 18) {
                    break outer;
                }
            }
            System.out.println();
        }
    }
}
```
Xoriant Solution Pvt. Ltd.

Classes & objects

What is a class?

➢ A class is nothing but a blueprint or a template for creating different objects which defines its properties and behaviors.

➢ A **class** can contain fields and methods to describe the behavior of an object.

Sample class

```
public class Date {
        private int day;
        private int month;          Attributes
        private int year;
        public Date() {
                this.day = 25;
                this.month = 7;     Default constructor
                this.year = 2010;
        }
        public Date(int day, int month, int year) {
                this.day = day;
                this.month = month;     Parameterized constructor
                this.year = year;
        }
        public int getDay() {
                return this.day;    Method
        }
}
```

```
public class DateTest {

    public static void main(String args[]) {

        Date date1 = new Date(); //Calling default constructor

        Date date2 = new Date(17, 4, 1995); //Calling parameterized constructor

    }

}
```

In order to create java object, we call default or parameterized constructor on the class.

In java, objects are always created on heap segment of the process. For example:

Date dt = new Date(27, 4, 2011);
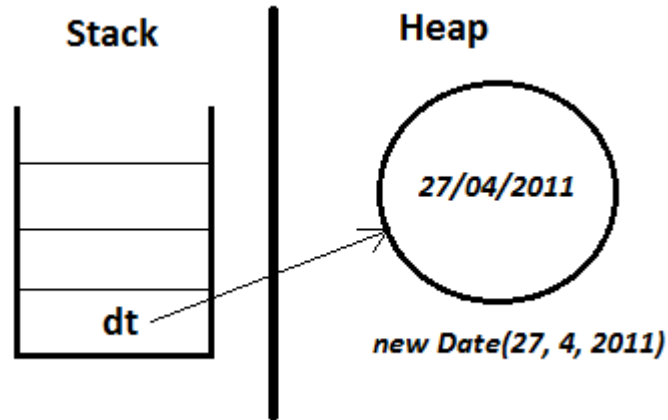
```java
public class VarArgs {
    static void test(int ... v) {
        System.out.println("Number of arguments :
"+v.Length+" Contains : ");
        for(int x : v){
            System.out.println(x+"  ");
        }
    }

    public static void main(String[] args){
        test(10);
        test(1,435,78);
        test();
    }
}
```

'static' keyword

'static' keyword in java can be used in 4 ways:

- ✓ Static as variable

- ✓ Static as method

- ✓ Static as class

- ✓ Static as block

'static' variables

```
public class Date {
        private int day;
        private int month;
        private int year;
        private static MAX_DAYS_IN_MONTH = 31;
}
```
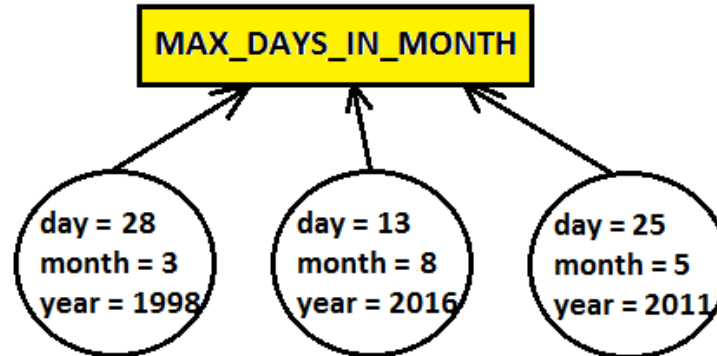
Instance variables/object variables/non-static variables

static variable/class variable

'static' variables continue…

➢ 'static' variables belong to the class & not to a specific instance.

➢ 'static' variables of a class always has single copy, irrespective of number of objects.

➢ The value of static variable can be changed.

➢ 'static' variables should be accessed using class name instance of object reference. For example: Date.MAX_DAYS_IN_MONTH = 31;

'static' methods

➢ 'static' methods are used to manipulate static variables of a class.

➢ 'static' method can be called without creating any object of the class.

➢ 'static' method should be accessed using class name instance of object reference. For example: Date.setMaxDaysInMonth(31);

➢ 'static' methods cannot access instance variables of a class because it does not have access to 'this' pointer.

'static' class

```
class OuterClass {
    static class NestedClass {

    }
}
```

➢ 'static' keyword can be applied only on inner classes & not on outer.

➢ Like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.

➢ Static nested classes are accessed using the enclosing class name:
  OuterClass.NestedClass nestedObject = new OuterClass.NestedClass();

47

```
class Test {

    private static int ary[] = new int[50]; //Array declaration

    static {

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

            ary[i] = i * 3;

        }

    }

}
```
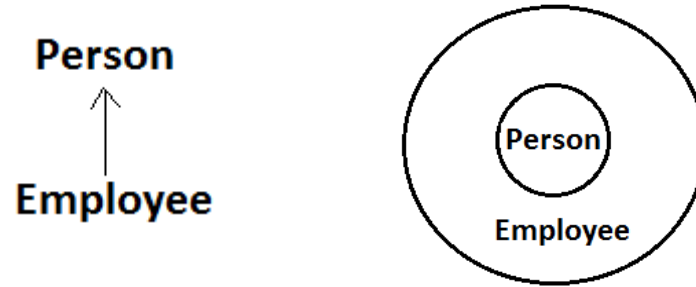
static block

'static' block continue...

➢ Static block is initialized implicitly by JVM when JVM loads the class into memory.

➢ Thus, the entry point in java is not a main() method but static block.

➢ Static block is initialized once & only once.

➢ Static block cannot be called explicitly.

➢ A class can have multiple static blocks. However, JVM combines them together into a single block & initializes as per their order in source code.

➢ Static block can access only static variables & not instance variables because it does not have access to *'this'* pointer.

➢ Developers use static block to execute the code which you wish to execute once & only once.

Singletone class

➢ Singletone class is special class that can be instantiated only once.

➢ In order to make a class singletone, the most important thing is stopping users to create its object using *'new'* keyword. This is achieved using private constructor. Please refer the code below:

```
class SampleSingletone {

        private static SampleSingletone ref = null;

        private SampleSingletone() { }

        public static SampleSingletone getInstance() {

                if(ref == null)

                        ref = new SampleSingletone();

                return ref;

        }

}
```

is-a relationship

Person

↑

Employee

Person

Employee

- ➢ Is-a relationship in java is represented in terms of inheritance.
- ➢ If class employee inherits class Person then we say that 'every employee *is a* person'.
- ➢ In is-a relationship, super class object is embedded inside sub class object. It means that you cannot imagine existence of employee object unless have person inside.
- ➢ You can also see is-a relationship as tight coupling between two objects.

```
class Person {
        Date dob;
}
```
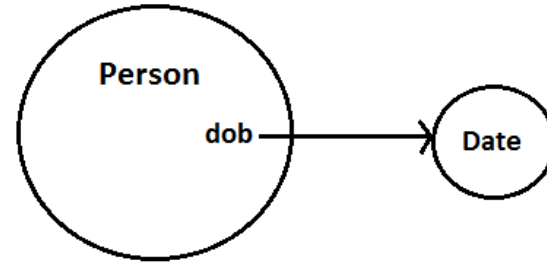

- ➢ has-a relationship in java is represented in terms object having an attribute of another object reference. If class Person has an attribute of class Date then we say that 'every person *has a* date of birth'.

- ➢ In has-a relationship, one object only holds reference of another object i.e. object outside object. It means that you can imagine existence of person object without having date associated.

- ➢ You can also see is-a relationship as loose coupling between two objects.

- ➢ Has-a relationship is also known as 'Composition'. If a class has list of another object type then it is called as 'Aggregation'. For example, Organization object has list of employee objects.
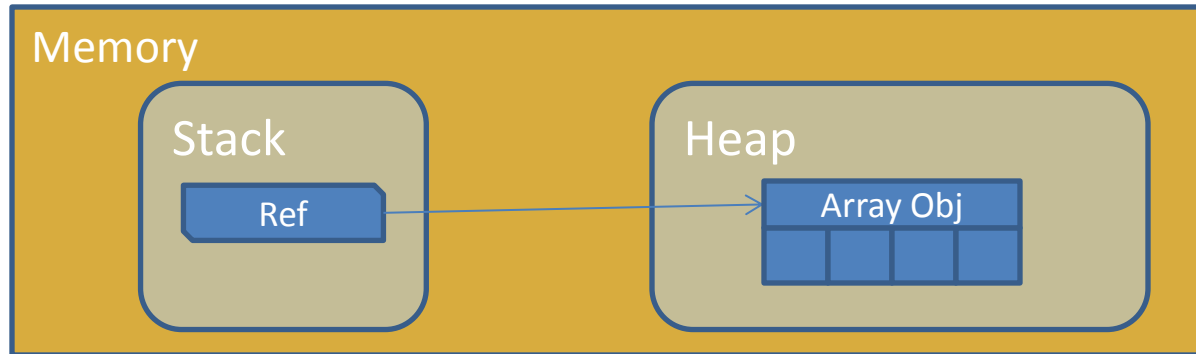
Arrays

Arrays
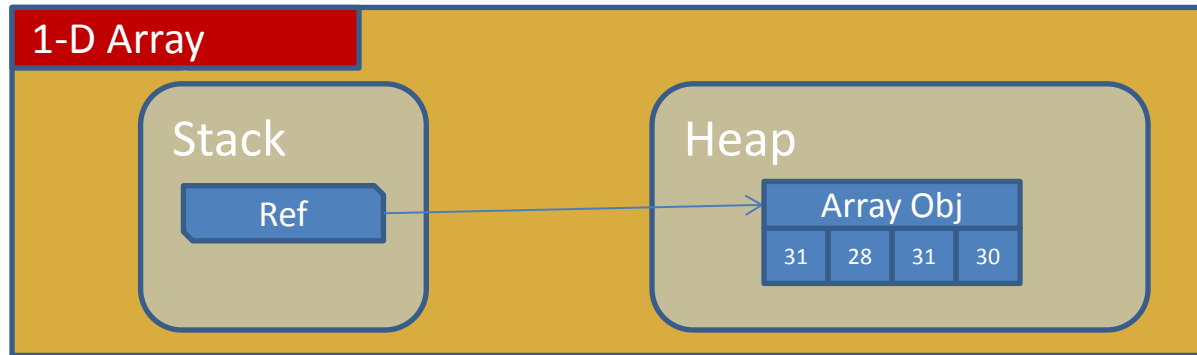
- In java, array is an object.

- Array Declaration

  int [ ] intArray = new int [5];

  int [ ][ ] array2D = new int [4][4];

- Length of an array

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

  System.out.println (intArray [i]);

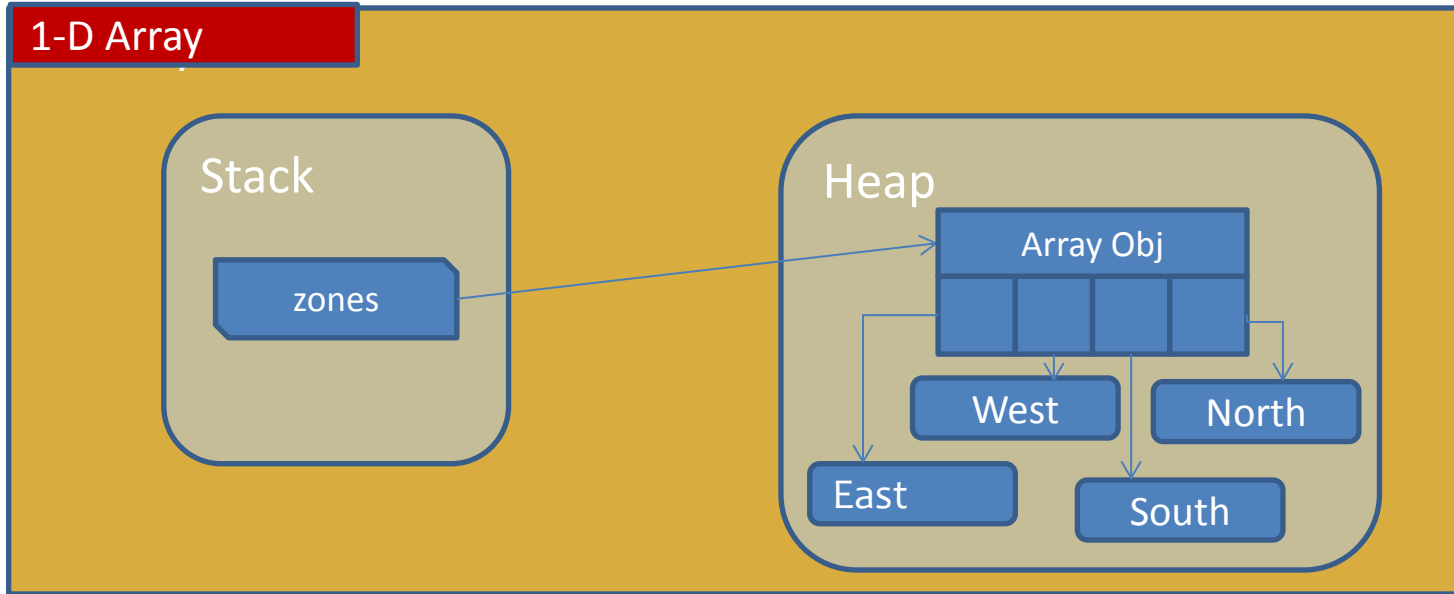# Arrays continue…

- Array Initialization
  - int [ ] daysInMonths = { 31, 28, 31, 30, ... };

- String [ ] zones = { "East", "West", "North", "South" } ;

```
int [ ][ ] pascalsTriangle = {
                        {1},
                        {1, 1},
                        {1, 2, 1},
                        {1, 3, 3, 1},
                        {1, 4, 6, 4, 1}
                      };


int[ ] [ ] array2D = new array2D [ 5 ][ ];  // First subscript is mandatory.
```
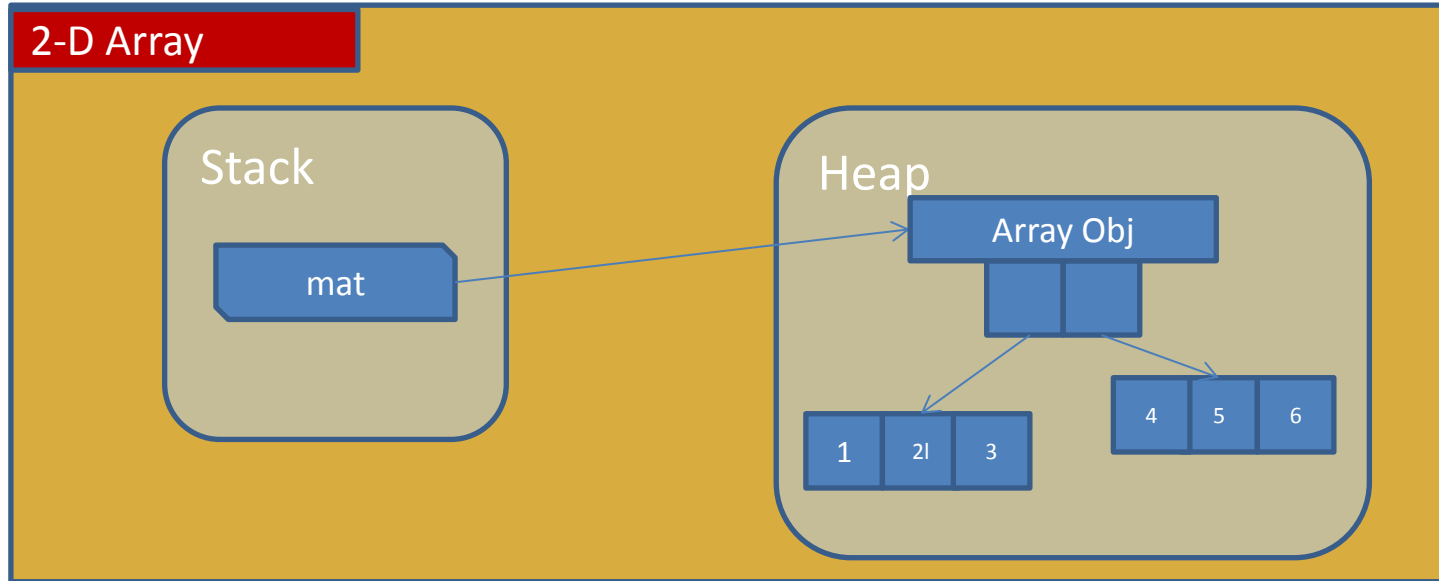
int [ ][ ] mat = {   { 1, 2, 3 },   { 4, 5, 6 }   };

Arrays of Arrays

```java
import java.util.Scanner;

public class TwoDimArray {
    public static void main(String[] args) {
        int [][] marks = new int [4][4];
        Scanner sc = new Scanner(System.in);
        for (int i = 1; i < marks.length ; i ++) {
            System.out.println ("Enter marks of student "+i);
            for (int j = 1; j < marks[i].length ; j++) {
                System.out.println ("Subject "+j);
                marks [i][j] = sc.nextInt();
            }
        }
        for (int i = 1; i < marks.length ; i++) {
            for (int j = 1; j < marks[i].length ; j++)
                System.out.print (marks[i][j]+" ");
        }
    }
}
```

▶ Xoriant Solution Pvt. Ltd.

OOP concepts

Introduction to OOP concepts

➤ In Object oriented programming, developer writes program to create & manipulate an object.

➤ Human being sees this world as a set of objects like fan, tube light, laptop, building, bus, car etc. OOP believes that the same concept we can use while writing program. Hence OOP concepts are based upon natural way of thinking. i.e. the way in which human think about this world, the same thought process is followed while writing program in OOP concepts. Due to this, developers feel object oriented language easier than a procedural language.

➤ Object oriented language is based upon four pillars:
1. Data abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

▶ Xoriant Solution Pvt. Ltd.

Data abstraction

➢ Data abstraction is a process of exposing only useful attributes of a class & keeping other attributes hidden.

➢ For example for a car object, one may expose car model, car name, car price & car color attributes to his friends where as other attributes like car insurance details, car engine number etc. can be kept hidden.

Encapsulation

➢ Encapsulation is a mechanism that binds together code and data in manipulates, and keeps both safe from outside interference and misuse.

➢ In encapsulation, the class becomes the 'capsule' or container for the data and operations.

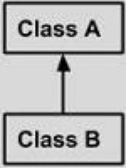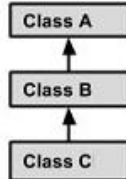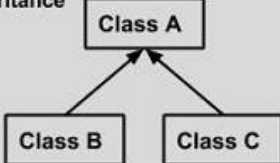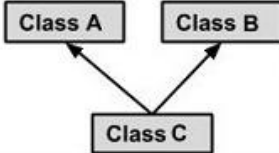➢ In short, a developer can achieve data abstraction using encapsulation.

Inheritance

➢ Using inheritance, one object can acquire all properties and behaviors of another object also known as 'parent object'.

➢ Inheritance provides code reusability.

▶ Xoriant Solution Pvt. Ltd.

Polymorphism

➢ The word *'poly'* means forms & *'morphism'* means forms. Thus, polymorphism is *'one name many forms'*.

➢ Polymorphism example can be a 'shape'. Shape can be circle, triangle, rectangle etc. Thus, shape is one name but having multiple forms.

➢ In object-oriented programming, polymorphism refers to a programming language's ability to process objects differently depending on their data type or class.

▶ Xoriant Solution Pvt. Ltd.

Inheritance relationship

Employee — Super class/ Base class

Manager — Sub class/ Derived class

Polymorphism types

Polymorphism types

Compile time / static polymorphism / early binding / false polymorphism

Runtime / dynamic polymorphism / late binding / true polymorphism

Method overloading / Operator overloading
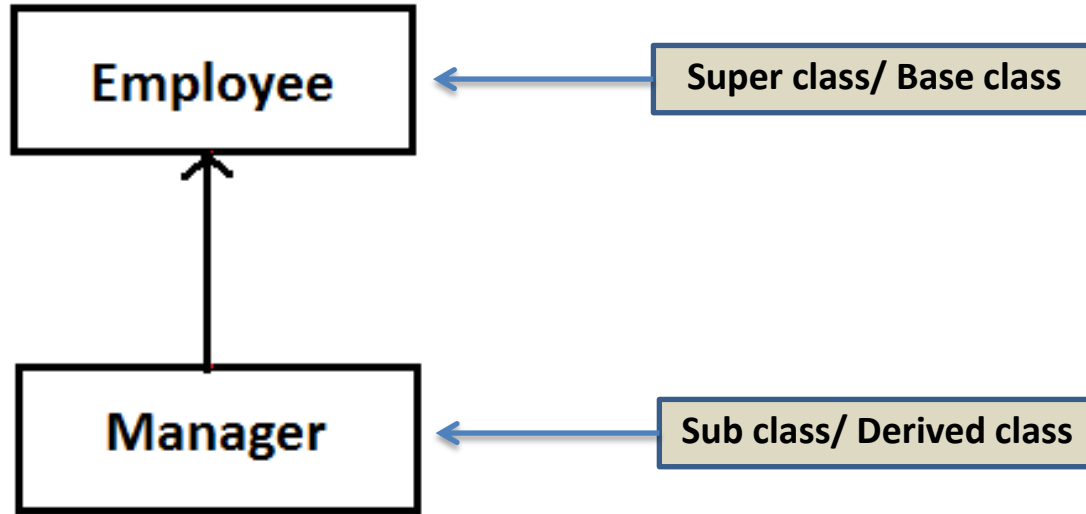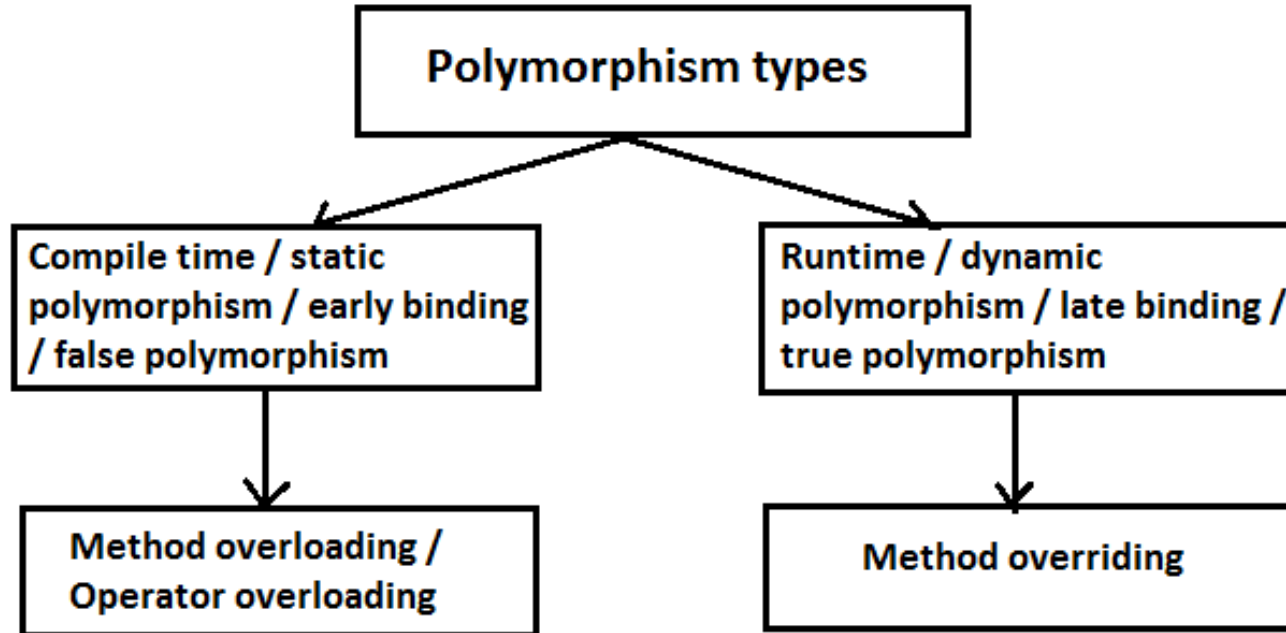
Method overriding

Method overloading

```
class Shape {

    public int getArea(int radius) {

        //calculates circle area

    }

    public int getArea(int length, int breadth) {

        //calculates rectangle area

    }

}
```

getArea() method is overloaded

Method overloading continue...

➤ We can achieve compile time polymorphism using method overloading.

➤ In method overloading, we write a class having multiple methods with the same name but different arguments.

➤ Method overloading is NOT dependent upon method's return type.

➤ Compiler changes name of every method, also called as 'name mangling'. Thus, methods having same name at source code level, might have different names in byte code level. This means when we run java program, JVM never considers overloaded methods with same name & hence method overloading is also called as 'false polymorphism'.

➤ Java does not support operator overloading. However, java has internally overloaded the plus (+) operator.

Method overriding

```
class Point {

    public void draw() {

        //draws a point

    }

}

class Circle extends Point {

    public void draw() {

        //draws a circle

    }

}
```

draw() method is overridden

Method overriding continue…

➢ In method overriding, the method signature (method name & arguments) in super class & sub class is exactly same.

➢ Overridden method's return type can be exactly same in sub class or return type can be sub class of overridden method's return type in super class. For example, if draw() method in class Point returns 'Object' then it can return 'Integer' in the class Circle.

➢ Method overriding is a true polymorphism because after method name mangling, the mangled name is same for super & sub class.

➢ When we call overridden method using sub class object, JVM always calls overridden method belongs to sub class only.

- ➢ 'final' keyword can be used for a resource whose value is constant & can never be changed.

- ➢ Developer can use 'final' keyword at three places:

    - ✓ 'final' variable

    - ✓ 'final' method &

    - ✓ 'final' class

- ✓ 'final' variable cannot be changed once assigned

- ✓ 'final' method cannot be overridden.

- ✓ 'final' classes cannot be inherited.

Object

Date   Account   Employee
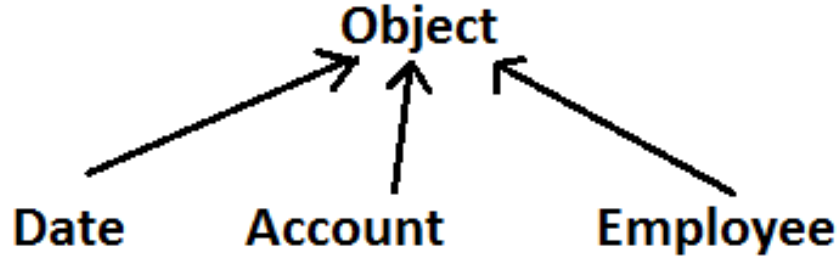
➤ Every java class directly or indirectly inherits a class called 'Object'. Thus, 'Object' is a super most or cosmic super class in java.

➤ Naturally, methods defined inside 'Object' class can be called on any java object.

'Object' class methods

➢ hashCode()

Every java object is assigned a hash code. You can find it out by calling obj.hashCode() method.

➢ equals()

The equals() method compares two objects whether they are same or not. For example:

String s1 = "Hello";

String s2 = "Hello";

if (s1.equals(s2)==true) { //s1.equals(s2) will be TRUE

//logic

}

➢ clone()

Cloning is a process of creating an exact copy of an object. You can clone java object by calling obj.clone() method.

➢ **toString()**

The toString() method returns the string representation of any java object. The toString() defined in class Object, returns a hash code of that object. If you wish to define different string representation of you object then you should override toString() method in your class. For example:

```
class Date {
    public String toString() {
        return day + "/" + month + "/" + year;
    }
}
```

➢ **finalize()**

The finalize() method is called implicitly by JVM when an object is being garbage collected.

- 'super' keyword can be used to call a constructor, method or a field of immediate super class. For example:

```
class DateTime extends Date {
    private int hours;
    private int minutes;
    private int seconds;
    public DateTime(int day, int month, int year, int hours, int minutes, int seconds) {
            super(day, month, year); //Calling super class constructor
            this.hours = hours;
            this.minutes = minutes;
            this.seconds = seconds;
    }
    public String toString() {
            return super.toString() + " " + hours + ":" + minutes + ":" + seconds; //Calling super class method
    }
}
```

Garbage collector

➢ Java developer can only allocate memory but deallocation is taken care by JVM itself using garbage collector.

➢ Garbage collector is a low priority thread which starts automatically when we run any java program.

➢ Garbage collector's job is to continuously scan heap memory & destroy those objects who are having reference count zero.

➢ Developer cannot force garbage collector on JVM. However, we can request JVM to run garbage collector by calling *System.gc()* method.

➢ If you want JVM to notify your object before getting destroyed then the object needs to override *finalize()* method.

Abstract classes

Point

Line

draw() method overridden

Circle

Rectangle

draw() method overridden

Cylinder

Cube
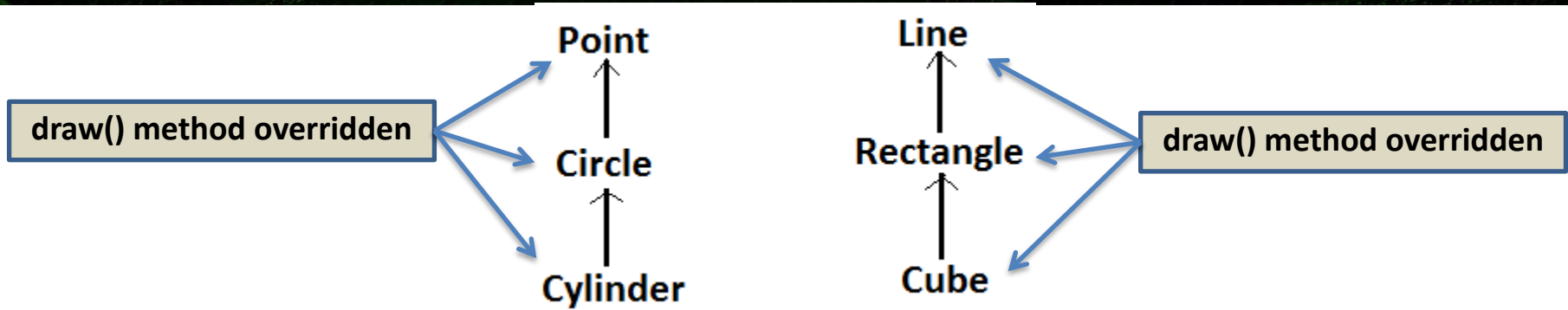
- Suppose there are two multilevel hierarchy of classes as shown above. All classes have overridden the draw() method. Now, we want to call draw() method on all objects then we will write the following code:

```
Point p[] = new Point[3];              Line ln[] = new Line[3];
p[0] = new Point();                    ln[0] = new Line();
p[1] = new Circle();                   ln[1] = new Rectangle();
p[2] = new Cylinder();                 ln[2] = new Cube();
for(int i=0; i<p.length; i++)          for(int i=0; i<ln.length; i++)
    p[i].draw();                           ln[i].draw();
```
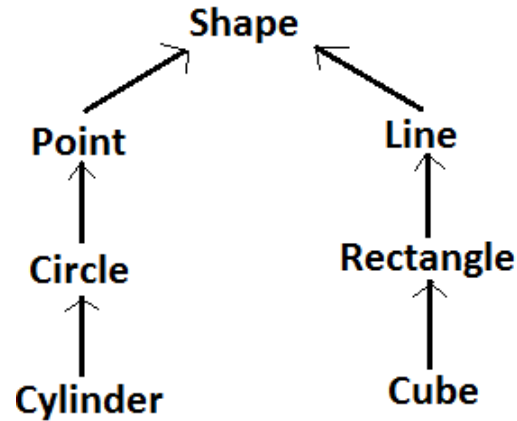
> In the earlier program, we had created two arrays (Point & Line) of three elements each. Instead of this, can we create a single array of six elements? Yes. This is possible by enforcing a common base class on top of Point & Line known as Shape. Now the program will be written as follows:

Shape s[] = new Shape[6];

s[0] = new Point();

s[5] = new Cube();

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

   s[i].draw();

➤ Now let us focus on the definition of class Shape. In order to compile our previous code, we must add draw() method inside Shape class. Here the question arises is 'Can Shape class define draw() method?'. Truly speaking NO because unless you tell which shape you want to draw, we cannot implement draw() method.

➤ Thus any method you are forced to write inside a class which you cannot define then such a method is called as abstract method.

➤ If class has one or more methods abstracts then you must declare the class as abstract.

abstract class Shape {

    abstract void draw(); *//Abstract methods do not have body.*

    }

Facts about abstract classes

- If class has one or more methods abstracts then you must declare the class as abstract.

- Abstract classes may have non abstract methods.

- Abstract classes cannot be instantiated.

- If a class extends any abstract class then the sub class must either override all abstract methods from super class or declare itself as abstract class.

- Abstract methods do not have body.

- Abstract methods cannot be private.

- Abstract methods cannot be final.

- If a class does not have any abstract class still developer can declare it as abstract if wishes.

Interfaces

What is an interface?

➢ Interface is a special type of abstract class.

➢ Interface is a fully abstraction of a class. It means interface can not have method definition possible.

➢ In interfaces, all methods are by default *'public abstract'* & all variables are by default *'public static final'*.

➢ Interface won't lead to fragile base class problem.

➢ If a class inherits an interface then it has to override all its abstract methods or declare itself as an abstract class.

➢ A class can inherit many interfaces.

Interface definition

Java provides us a keyword called 'interface' to define a user defined interface. For

example:

```
public interface Printable {
    void print();
}
public class Date implements Printable {
    public void print() { //prints date }
}
```

We have added a method print(). However, we did not declare it *'public abstract'* because

it is implicitly understood once *'interface'* keyword is used.

A class uses 'implements' keyword in order to inherit an interface.

Understanding interfaces
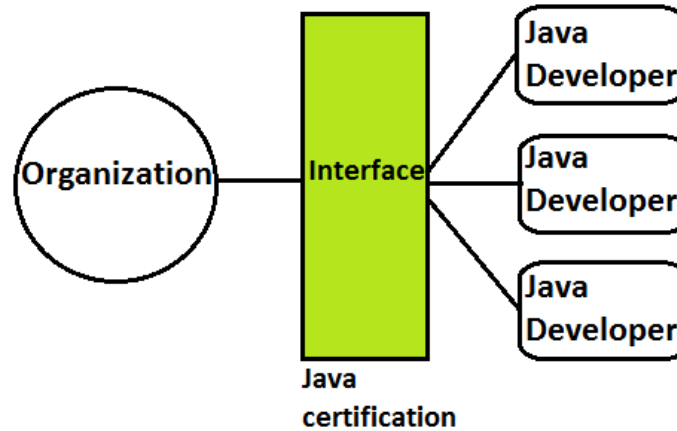
➢ Interfaces are service based i.e. whenever you want to expose any service, you have to use interfaces. This is because interface sign a contract between service provider & service user. For example if an organization wants to recruit java developers who are 'java certified' then java certification acts as an interface between java developer & the organization.

➢ Interfaces are generally designed by third party. For example in above diagram, 'java certification' is provided by 'Oracle Inc.'

➢ Interfaces gather irrelevant objects together. For example, *'Printable'* interface can be implemented by Circle, Employee, Date, Account objects. Note that all these are irrelevant objects. However, you can gather them together by making a common array of Printable type as follows:

> ***Printable p[] = new Printable[4];***
>
> p[0] = new Circle();
>
> p[1] = new Employee();
>
> p[2] = new Date();
>
> p[3] = new Account();
>
> for(int i=0; i<p.length; i++)
>
> > ***p[i].print();***

➢ Marker interface is an interface having no method inside. It means that a sub class implementing a marker interface need not override any method. For example:

> ➢ interface Cloneable { }

➢ Marker interfaces are permission granting interfaces. It means they grant permission to perform certain operation. For example, if you wish to clone any java object, you simply call object.clone() method. However, java does not allow to clone any java object unless object gives permission. So, here the object must implement 'Cloneable' interface in order to grant permission to JVM.

➢ There are several marker interfaces provided by java APIs like Cloneable, Serializable, SingleThreadModel, Remote etc.

Quiz

Suppose we have three classes A, B, C & three interfaces Intf1, Intf2, Intf3. Now choose the correct statement:

➢ class B extends A

  Yes

➢ class C extends A, B

  No

➢ class B extends Intf1

  No

➢ class B extends A implements Intf1

  Yes

➢ class B implements Intf1, Intf2

  Yes

Quiz continue…

➢ interface Intf1 extends A

  No

➢ interface Intf1 implements A

  No

➢ interface Intf2 implements Intf1

  No

➢ interface Intf2 extends Intf1

  Yes

➢ interface Intf3 extends Intf1, Intf2

  Yes

Packages

What is a package?

➢ Package is a group of similar types of classes, interfaces and sub-packages.

➢ Package is used to categorize the classes and interfaces so that they can be easily maintained.

➢ Package provides access protection.

➢ Package removes naming conflict.

➢ Package in java can be categorized in two form, built-in package and user-defined package.

```
package mypack;
import java.util.*;
public class Main{
 public static void main(String args[]){
   System.out.println("Welcome to package mypack");
   }
}
```

➢ Package statement must be a first statement in the source file.

➢ According to naming convention, package name should be all lowercase letters.

➢ If you do not use the package statement, the classes and interfaces of a source file become a member of an unnamed default package.

➢ Finally, compile the source file as follows:

javac –d . Main.java

Java in-built packages

java.lang (default)

System
Thread
Exception
String
StringBuffer

java.io

Reader
Writer
InputStream
OutputStream
PrintWriter

java. net

Socket
ServerSocket
URLEncoder
InetAddress
SocketImpl

java.util

HashMap
Set
Vector
List
ArrayList

java.awt

Applet
Frame
Button
TextField
Checkbox

Access Specifiers

Access	Private	Default	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same package Subclass	No	Yes	Yes	Yes
Same package Non-Subclass	No	Yes	Yes	Yes
Different package Subclass	No	No	Yes (But only through Inheritance)	Yes
Different package Non-Subclass	Yes	Yes	Yes	Yes

Xoriant Solution Pvt. Ltd.

Wrapper classes

Wrapper class

➢ Wrapper class wraps a primitive datatype & provides the mechanism *to convert primitive into object and object into primitive*.

➢ Java provides wrapper class for every primitive data type. For example:

Primitive datatype	Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Wrapper class hierarchy
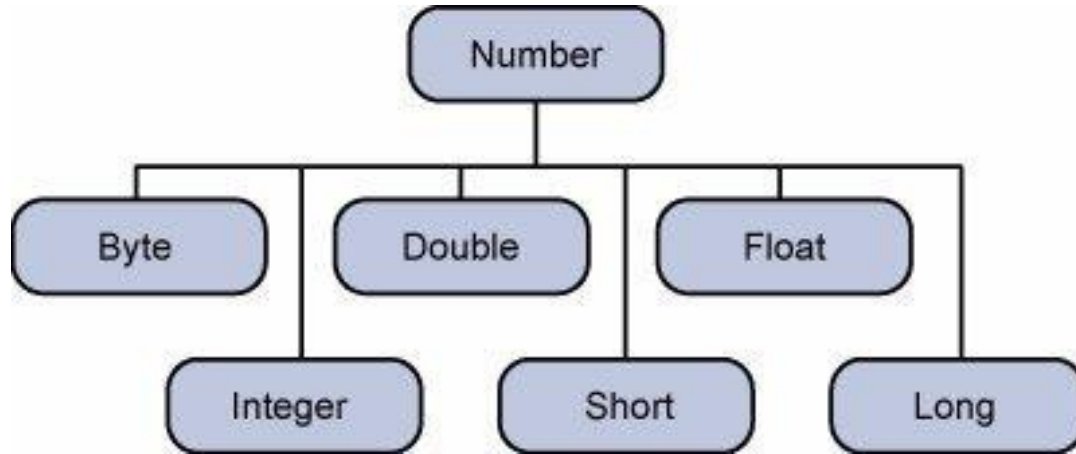
➢ Most of the wrapper classes extend the class java.lang.Number.
➢ Wrapper classes are final classes.
➢ Wrapper class 'Boolean' & 'Character' directly extend class Object.

Utility methods

Wrapper classes also provide us following utility methods:

➢ Convert string into integer primitive.

  int x = Integer.parseInt("25");

➢ Convert string into integer object.

  Integer obj = Integer.valueOf("25");

➢ Finding greater number.

  int maxNo = Integer.max(12, 25); //25

➢ Finding smaller number.

  int minNo = Integer.min(12, 25); //12

Autoboxing & unboxing

- ➤ Warpper classes frequently need to apply boxing & unboxing operations. For example:

  int x = 20;

  Integer intObj = new Integer(x); //Boxing

  int y = intObj.intValue(); //Unboxing

- ➤ Now, the question arises is, can JVM implicitly take care boxing & unboxing operations? The answer is YES. It is because from JDK 1.5, java has introduced auto-boxing & auto-unboxing features.

- ➤ Autoboxing is a process of converting of primitive data types into its equivalent Wrapper type.

  Integer intObj = 20; //Autoboxing

  int y = intObj; //Auto-unboxing

String, StringBuffer & StringBuilder

'String' in Java
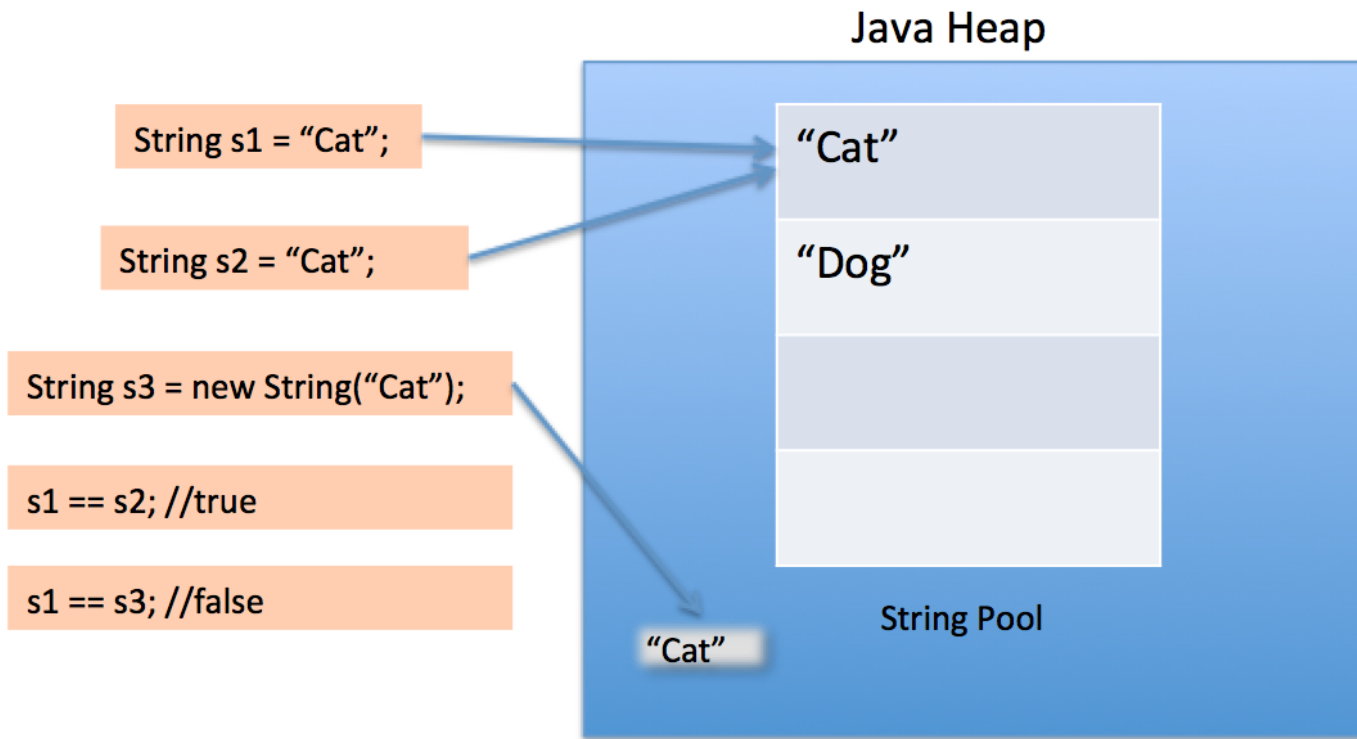
➢ Java provides a class called String to hold string data. Thus, in java string is an object instead of character array.

➢ The class 'String' is final class.

➢ Strings are immutable. It means, once String object is created, that object cannot be changed.

➢ String can be created in 2 ways:

       String str = new String("Hello");

       OR

       String str = "Hello";

String Pool

String methods

length()
    The length() method is used to calculate length of a string.
    int len = "Hello".length(); //returns 5

trim()
    Removes leading and trailing whitespaces of a string.
    String str = "   Hello   ".trim(); //returns "Hello"

substring()
    Returns a string that is a substring of this string.
    String str = "Hello".substring(0, 2); //returns "He"

startsWith()
    Tests if this string starts with the specified prefix.
    boolean b = "Hello".startsWith("he"); //returns false

endsWith()
Tests if this string ends with the specified suffix.
boolean b = "Hello".endsWith("lo"); //returns true

charAt()
Returns the char value at the specified index.
char ch = "Hello".charAt(1); //returns 'e'

equals()
Compares this string with another string.
boolean b = "Hello".equals("hello"); //returns false

concat()
Concatenates the specified string to the end of this string.
String str = "Hello".concat(" World"); //returns "Hello World"

contains()
    Returns true if the string contains the specified sequence of char values.
    boolean b = "Hello".contains("ell"); //returns true

getBytes()
    Converts every character of a string into byte & finally returns array of bytes.
    byte b[] = "ABCD".getBytes(); //returns [65, 66, 67, 68]

indexOf()
    Returns the index within this string of the first occurrence of the specified substring.
    int x = "Hello".indexOf("lo"); //returns 3

replace()
    Replaces each occurance of target string with specified replacement string.
    String str = "I wake up early. I go to office.".replace("I", "We"); //returns "We wake up early. We go to office."

split()

Splits the string around matches of the given regular expression.

String tokens[] = "Hello World".split(" "); //returns "Hello" & "World"

toUpperCase()

Converts all characters of a string into upper case.

String str = "Hello".toUpperCase(); //returns "HELLO"

toLowerCase()

Converts all characters of a string into lower case.

String str = "Hello".toLowerCase(); //returns "hello"

Mutable strings (StringBuffer & StringBuilder)

➢ String is an immutable class. Hence we should use the string only if it is not changing frequently. However, few applications demand frequent string manipulation & hence we should use 'StringBuffer' or 'StringBuilder'.

➢ Thus, StringBuffer and StringBuilder classes are used when there is a necessity to make a lot of modifications to Strings of characters.

```
StringBuilder sb = new StringBuilder("Hello");
sb.append(" World");
System.out.println(sb); //prints "Hello World"
```

➢ The default capacity of StringBuffer & StringBuilder is 16. However it is extensible depending upon your data.

➢ Methods of StringBuffer & StringBuilder are similar to the class String itself.

StringBuffer vs StringBuilder

StringBuffer	StringBuilder
StringBuffer is synchronized i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is non-synchronized i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
StringBuffer is less efficient than StringBuilder.	StringBuilder is more efficient than StringBuffer.
In multithreaded application, we should use StringBuffer in order to overcome data corruption.	In single threaded application, we should use StringBuilder.

Reflection

What is Reflection?

- ➢ Java Reflection is used to inspect classes, interfaces, fields and methods at runtime.

- ➢ Using reflection, it is also possible to instantiate new objects, invoke methods and get/set field values.

- ➢ There are several applications where reflection can be useful. For example:

    1. Debugger tools
    2. Servers
    3. Middleware (example RMI)

Reflecting a class

Here is a sample program to analyze a class String:

import java.lang.reflect.*; //import reflection package

Class c = Class.*forName("java.lang.String"); //loads the class forcefully into memory*

Constructor con[] = c.getDeclaredConstructors(); //returns all constructors of String class

Method m[] = c.getDeclaredMethods(); //returns all methods of String class

Field f[] = c.getDeclaredFields(); //returns all fields of String class

'Class' object

- Java provides a class called 'Class'. It plays vital role in reflecting any java class.
- How to object class 'Class' object?

    Class strClass = String.class;

    OR

    Class strClass = Class.forName("java.lang.String");

- Useful methods of class 'Class':

    int modifiers = strClass.getModifiers();

    Modifier.isAbstract(modifiers);

    Modifier.isPublic(modifiers);

    Modifier.isFinal(modifiers);

    String packageName = strClass.getPackage(); //returns package name

    Class interfaces[] = strClass.getInterfaces(); //returns all implemented interfaces

    Class superClass = strClass.getSuperClass(); //returns base class

Access private resources

Reflection APIs are very powerful & they even allow us to access the value of private fields or call private methods etc. Here is a sample code:

```
class Sample {
    private String messageTxt = "Welcome";
}

Class sClass = Sample.class;
Field messageField = sClass.getDeclaredField("messageTxt");
messageField.setAccessible(true);
System.out.println("messageText = "+ messageField.get(new Sample())); //prints 'Welcome'
```

If you do not want your classes getting reflected at runtime then set security policy to stop reflection API usage.

Java Annotations

What are annotations?

➤ Annotation feature was added from Java 1.5.

➤ Annotation provides metadata for your java program. For example:

      @Entity

      public class Account {

       //fields

      }

➤ Java annotations are typically used for the following purposes:

    1. Compiler instructions

    2. Build-time instructions

    3. Runtime instructions

11

Annotation basics

➢ Annotation elements are similar to attributes of a class. For example:

        @Table(name='ACC_MASTER', primaryKey='accno')

➢ Annotation can be placed at following levels:

1. Class level
2. Interface level
3. Method level
4. Field level
5. Method parameter level &
6. Local variable level

Java built-in annotations

➢ **@Deprecated**

The @Deprecated annotation is used to mark a class, method or field as deprecated, meaning it should no longer be used.

➢ **@Override**

The @Override Java annotation is used above methods that override methods in a superclass.

➢ **@SuppressWarnings**

The @SuppressWarnings annotation makes the compiler suppress warnings for a given method.

Custom annotations

```java
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface MyAnnotation {

    String   value() default "";
    String   name();
    int      age();
    String[] newNames();
}
```

@interface

@interface signals to the Java compiler that this is a Java annotation definition.

Default

Using 'default', we can assign default values to an element.

@Retention

@Retention will allow us to specify whether the annotation will be available at runtime for reflection or not.

@Target

@Target can be used to specify at what level you want to apply the custom annotation. i.e. method level, class level, method parameter level etc.

Enums

```
class FruitName {
        public static final int APPLE = 1;
        public static final int BANANA = 2;
        public static final int MANGO = 3;
        public static final int ORANGE = 4;
}
class Food {
        private int fruit;
}
```

Limitations:

➢ Possible assigning of incorrect values i.e. food.fruit = 7;

➢ No meaningful printing. i.e. food.fruit = 3;

➢ Need prefix class name in order to access the constant i.e. FruitName.APPLE

public enum fruitName { *APPLE, BANANA, MANGO, ORANGE* *};*

➢ Always add curly braces around enum constants because enum is like a class.

➢ Enum constants are implicitly 'static final'.

➢ You cannot change the value of constants once created.

➢ Enum can be used inside switch statement.

➢ Enum is type-safe:

public enum FruitNames { APPLE,BANANA,MANGO,ORANGE};

FruitNames fruit = FruitNames.BANANA;

fruit = 5; //compilation error

➢ Enum is Java are reference type:

Java enum is like a class or interface & hence you can declare constructor, method & variables inside enum.

➢ Enum constant values:

We can specify the value of enum constants at the creation time.

enum FruitNames { APPLE(10), BANANA(5), MANGO(8), ORANGE(3)};

Enum features continued…

➢ You can declare constructor in enum. However it must be private since it is called within enum only.

➢ In order to get the value of enum item, you can write getValue() method inside enum.

➢ Enum can be used with switch cases-

FruitNames fruit = FruitNames.MANGO;

Switch(fruit) {

case APPLE: System.out.println("Apple"); break;

case BANANA: System.out.println("Banana"); break;

}

Enum features continued...

➢ You can compare enum values with '=='

  FruitNames fruit = FruitNames.MANGO;

  If (fruit == FruitNames.MANGO) //TRUE

➢ You can override toString() inside enum as follows:

  public enum FruitNames {

  APPLE(4), BANANA(2), MANGO(7), ORANGE(10);

  public String toString() {

  switch(this) {

  *   case APPLE: System.out.println("Fruit: " + value); break;*

  * }*

  }

Nested classes
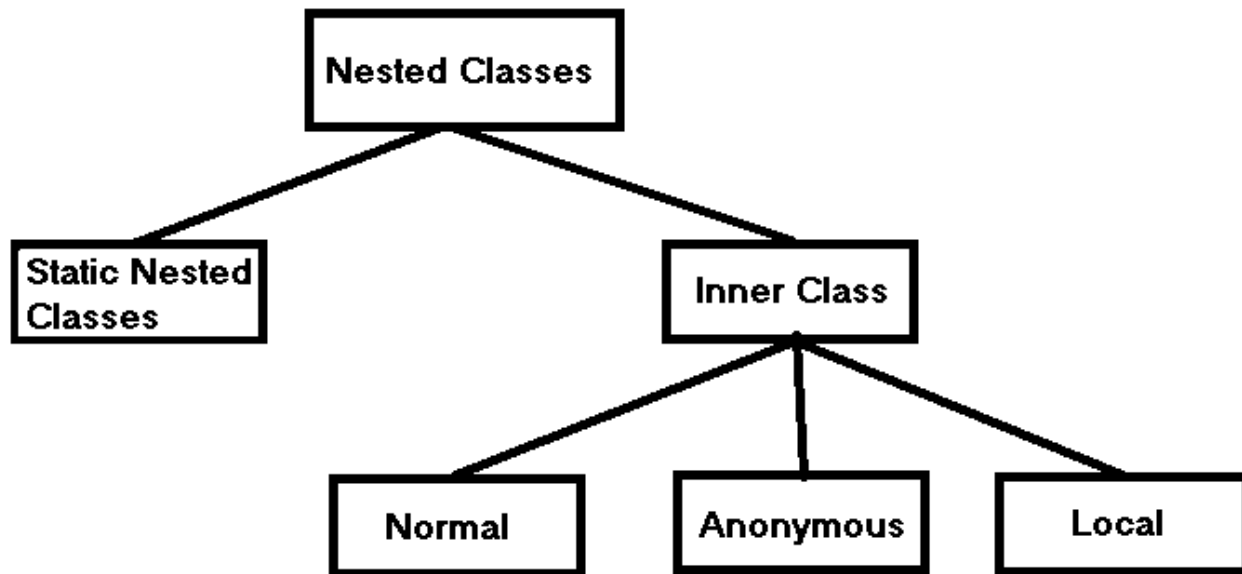
What is Nested Class?

Java allows you to define a class within another class. Such a class is called a nested class.

```
class OuterClass {

     class NestedClass {

     }

}
```

Outer classes can only be declared as public or package private. Where as, Nested class can be declared private, public, protected, or package private.

Why to use Nested classes?

➢ Logically grouping the classes at one place.

➢ Encapsulation advantage

➢ More readable & maintainable code.

```
class OuterClass {

        static class NestedClass {

        }

}
```

Like static class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class: it can use them only through an object reference.

Static nested classes are accessed using the enclosing class name:

OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();

```
class OuterClass {

        class NestedClass {

        }

}
```

Inner class has direct access to the outer class fields & methods. Inner class is instantiated as follows:

OuterClass.InnerClass innerObject = outerObject.new InnerClass();

```
public void performArithOpr(int x, int y) {
        class ArithmeticOpr {      //Local class
                public int add(int a, int b) {
                        return a + b;
                }
        }
}
```

1. Local class has access to the members of its enclosing class & also to local variables.
2. Local classes can be declared inside method, if condition, loops etc.

Anonymous inner classes

```
Printable printableCircle = new Printable() {
    public void print() {
            System.out.println("Circle printed..");
    }
};
```

1. Anonymous class makes your code brief but comprehensive.
2. Anonymous class enables to declare & instantiate a class at the same time.
3. Anonymous class is a local class with no name.
4. Anonymous class is used if you need local class only once.

Thank You!

US – Corporate Headquarters

1248 Reamwood Avenue,
Sunnyvale, CA 94089
Phone: (408) 743 4400

343 Thornall St 720
Edison, NJ 08837
Phone: (732) 395 6900

UK

20 Broadwick Street
Soho, London
W1F 8HT, UK

89 Worship Street
Shoreditch,
London EC2A 2BF, UK
Phone: (44) 2079 938 955

India

Mumbai
4th Floor, Nomura
Powai , Mumbai 400 076

Pune
5th Floor, Amar Paradigm
Baner, Pune 411 045

Kolkata
2B, 12th Floor, Tower 'C'
Rajarhat, Kolkata 700 156

Bangalore
4th Floor, Kabra Excelsior,
80 Feet Main Road,
Koramangala 1st Block,
Bengaluru (Bangalore) 560034

Gurgaon
A/373rd Floor, Sigma Center
Gurgaon, Haryana 122 011s