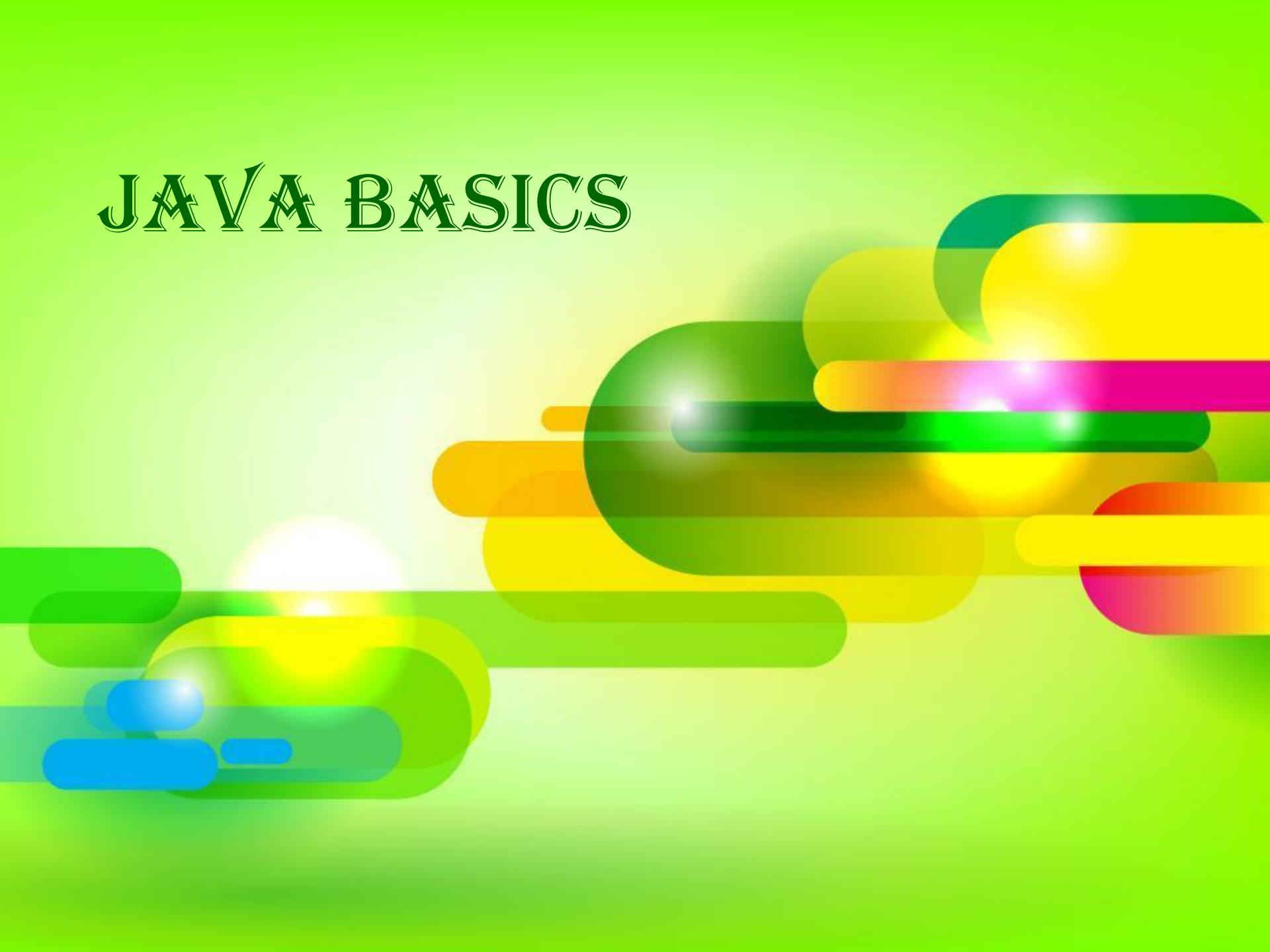


JAVA BASICS



JDK (JAVA DEVELOPMENT KIT)

- ❖ JDK is a software development environment used for developing Java applications.
- ❖ 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.
- ❖ JDK consists of the core Java API and some utilities for building, testing and documenting Java programs.

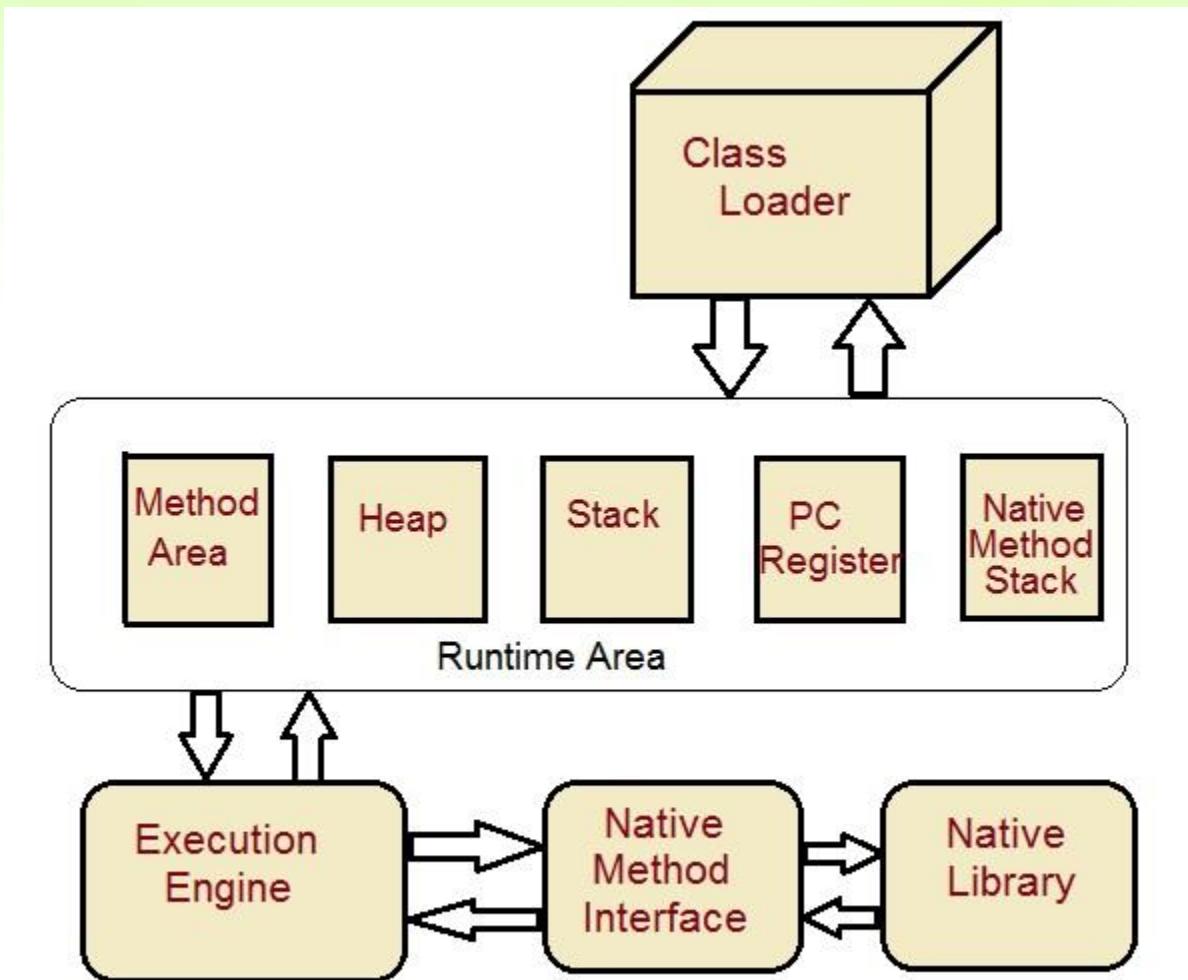
JRE (JAVA RUNTIME ENVIRONMENT)

- ❖ JRE contains everything required to run Java application which has already been compiled.
- ❖ It doesn't contain the code library required to develop Java application.
- ❖ It provides the libraries, the **Java Virtual Machine**, and other components to run applications written in the Java programming language.
- ❖ The JRE does not contain tools and utilities such as compilers or debuggers for developing applications.

JVM (JAVA VIRTUAL MACHINE)

- ❖ The Java Virtual machine (JVM) is the virtual machine that runs the Java bytecodes.
- ❖ JVM only works with bytecode. Hence you need to compile your Java application(.java) so that it can be converted to bytecode format (also known as the .class file).
- ❖ Bytecode then will be used by JVM to run application.
- ❖ There are specific implementations of the JVM for different systems (Windows, Linux, MacOS etc.).

JVM ARCHITECTURE



JDK VS. JRE VS. JVM

- ❖ JRE = JVM + Required Library to run Application.
- ❖ JDK = JRE + Required Library to develop Java Application.
- ❖ The **JDK** is a **superset** of the **JRE**, and contains everything that is in the JRE, plus tools such as the compilers and debuggers necessary for developing applications.

HOW TO SET PATH

- ❖ To set the PATH variable permanently, add the full path of the jdk1.8.0\bin directory to the PATH variable. Typically, this full path looks something like C:\Program Files\Java\jdk1.8.0\bin. Set the PATH variable as follows on Microsoft Windows:
 - ❖ Click Start, then Control Panel, then System.
 - ❖ Click Advanced, then Environment Variables.
 - ❖ Add the location of the bin folder of the JDK installation to the PATH variable in System Variables. The following is a typical value for the PATH variable:
❖ C:\WINDOWS\system32;C:\WINDOWS;C:\Program Files\Java\jdk1.8.0\bin

WHY DO WE SET PATH IN JAVA?

- ❖ To execute java console based programs in windows environment we have to use java and javac commands.
- ❖ Since, java and javac commands are unknown for windows till we do not specify explicitly where those executable resides.
- ❖ This is the reason while setting the path we specify path of bin folder(bin contains all the binary executable).

JAVA5 - FEATURES

- ❖ Generics
- ❖ Annotations
- ❖ Autoboxing/unboxing
- ❖ Enumerations
- ❖ Varargs
- ❖ Enhanced for each loop
- ❖ Static imports
- ❖ New concurrency utilities in `java.util.concurrent`
- ❖ Scanner class for parsing data from various input streams and buffers.

JAVA6 - FEATURES

- ❖ Scripting Language Support
- ❖ Performance improvements
- ❖ JAX-WS
- ❖ JDBC 4.0
- ❖ Java Compiler API
- ❖ JAXB 2.0 and StAX parser
- ❖ Pluggable annotations
- ❖ New GC algorithms

JAVA 7 - FEATURES

- ❖ JVM support for dynamic languages
- ❖ Compressed 64-bit pointers
- ❖ Strings in switch
- ❖ Automatic resource management in try-statement
- ❖ The diamond operator
- ❖ Simplified varargs method declaration
- ❖ Binary integer literals
- ❖ Underscores in numeric literals
- ❖ Improved exception handling
- ❖ ForkJoin Framework
- ❖ NIO 2.0 having support for multiple file systems, file metadata and symbolic links
- ❖ WatchService

JAVA 7 - FEATURES

- ❖ Timsort is used to sort collections and arrays of objects instead of merge sort
- ❖ APIs for the graphics features
- ❖ Support for new network protocols, including SCTP and Sockets Direct Protocol

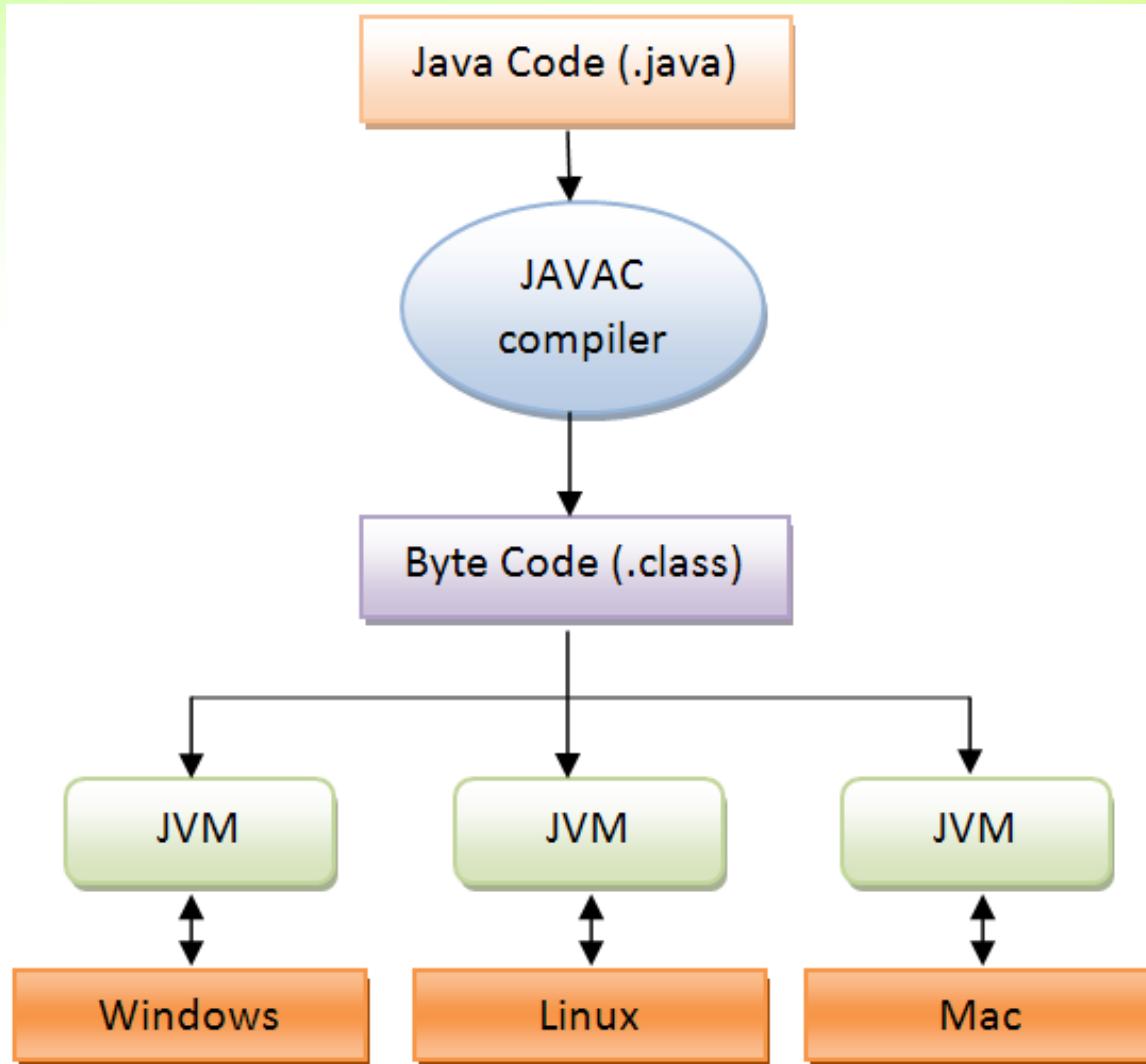
JAVA8 - FEATURES

- ❖ Lambda expression support in APIs
- ❖ Functional interface and default methods
- ❖ Optionals
- ❖ Nashorn – JavaScript runtime which allows developers to embed JavaScript code within applications
- ❖ Annotation on Java Types
- ❖ Unsigned Integer Arithmetic
- ❖ Repeating annotations
- ❖ New Date and Time API
- ❖ Statically-linked JNI libraries
- ❖ Launch JavaFX applications from jar files
- ❖ Remove the permanent generation from GC

JAVA FEATURES

- ❖ Object Oriented
- ❖ Platform Independent
- ❖ Simple
- ❖ Secure
- ❖ Architecture neutral
- ❖ Portable
- ❖ Robust
- ❖ Multithreaded
- ❖ Interpreted
- ❖ High Performance

BYTE CODE



❖ Java class is written in unicode characters.

❖ Java compiler converts these unicode characters into Byte code.

❖ Java Byte code can only be understandable by JVM

IDE - ECLIPSE

Release	Main Release	Platform Version	Projects
Oxygen	June 2017		
Neon	22 June 2016	4.6	
Mars	24 June 2015	4.5	Mars Projects
Luna	24 June 2014	4.4	Luna Projects
Kepler	26 June 2013	4.3	Kepler Projects
Juno	27 June 2012	4.2	Juno Projects
Indigo	22 June 2011	3.7	Indigo Projects

JAVA DATA TYPES

- ❖ In java, data types are classified into two categories :
 - ❖ **Primitive Data type** – byte, short, int, long, float, double, char, boolean.
 - ❖ **Non-Primitive Data type**

PRIMITIVE DATA TYPES

- ❖ The primitive types are also commonly referred to as simple types which 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.
- ❖ **Boolean:** This group includes **boolean**, which is a special type for representing true/false values.

INTEGERS

- ❖ **byte** : It is 1 byte(8-bits) integer data type. Value range from -128 to 127. Default value zero. example: byte b=10;
- ❖ **short** : It is 2 bytes(16-bits) integer data type. Value range from -32768 to 32767. Default value zero. example: short s=11;
- ❖ **int** : It is 4 bytes(32-bits) integer data type. Value range from -2147483648 to 2147483647. Default value zero. example: int i=10;
- ❖ **long** : It is 8 bytes(64-bits) integer data type. Value range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Default value zero. example: long l=100012;

FLOATING POINT NUMBER

❖ **float** : It is 4 bytes(32-bits) float data type. Default value 0.0f. example:
float ff=10.3f;

❖ **double** : It is 8 bytes(64-bits) float data type. Default value 0.0d.
example: double db=11.123;

❖ Boolean

❖ The boolean data type has only two possible values: true and false.
Example:

```
boolean flag = true;  
booleanval = false;
```

CHARACTERS

- ❖ **char** : It is 2 bytes(16-bits) unsigned unicode character.
- ❖ It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive). There are no negative chars.

```
char ch1 = 88; // code for X  
char ch2 = 'Y';
```

NON-PRIMITIVE(REFERENCE)DATA TYPE

- ❖ A reference data type is used to refer to an object. A reference variable is declared to be of specific type and that type can never be changed.

TYPE CASTING

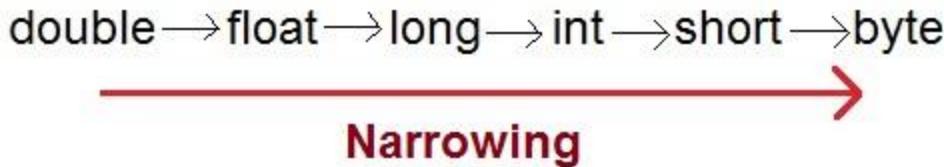
- ❖ Assigning a value of one type to a variable of another type is known as **Type Casting**.

```
int x = 10;  
byte y = (byte)x;
```

- ❖ In Java, type casting is classified into two types:
- ❖ Widening casting or upcasting(implicit)



- ❖ Narrowing casting or downcasting (explicit)



TYPE CASTING

- ❖ Automatic Type casting take place when,
 - ❖ the two types are compatible
 - ❖ the target type is larger than the source type

```
int i = 100;  
long l = i; //no explicit type casting required  
float f = l //no explicit type casting required
```

- ❖ When you are assigning a larger type value to a variable of smaller type, then you need to perform **explicit type casting**.

```
double d = 100.04;  
long l = (long)d; //explicit type casting  
required  
int i = (int)l; //explicit type casting  
required
```

MAIN METHOD

```
public static void main (String[] args)
```

- ❖ The main method must be **public** so it can be found by the JVM when the class is loaded.
- ❖ Similarly, it must be **static** so that it can be called after loading the class, without having to create an instance of it.
- ❖ All methods must have a **return type**, which in this case is void.
- ❖ The list of **String arguments** is there to allow to pass parameters when executing a Java program from the command line.

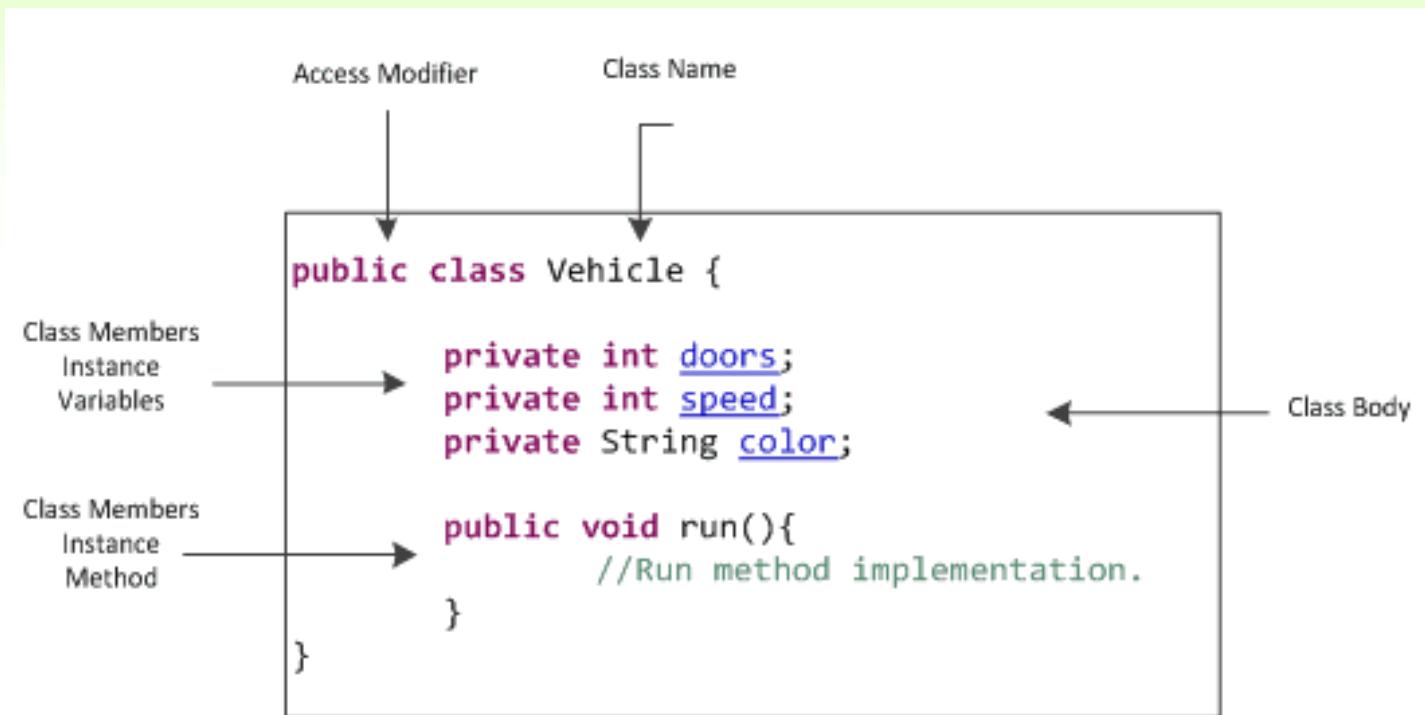
CLASS

- ❖ A class can be defined as a template/blueprint that describes the behavior/state that the object of its type supports.
- ❖ A class defines new data type. Once defined this new type can be used to create object of that type.
- ❖ Object is an instance of class.
- ❖ A class is declared by '**class**' keyword. A class contains both **data** and the **code** that operates on that data.
- ❖ The data or variables defined within a class are called **instance variables** and the code that operates on this data is known as **methods**.
- ❖ The instance variables and methods are known as **class members**.

RULES FOR JAVA CLASS

- ❖ A class can have only public or default(no modifier) access specifier.
- ❖ It can be either abstract, final or concrete (normal class).
- ❖ It must have the class keyword, and class must be followed by a legal identifier.
- ❖ It may optionally extend one parent class. By default, it will extend java.lang.Object.
- ❖ It may optionally implement any number of comma-separated interfaces.
- ❖ The class's variables and methods are declared within a set of curly braces {}.
- ❖ Each .java source file may contain only one public class. A source file may contain any number of default visible classes.
- ❖ Finally, the source file name must match the public class name and it must have a .java suffix.

CLASS



CLASS EXAMPLE

- ❖ Lets create a **Student** class

```
class Student  
{  
    String name;  
    int rollno;  
    int age;  
}
```

- ❖ Now create a reference variable

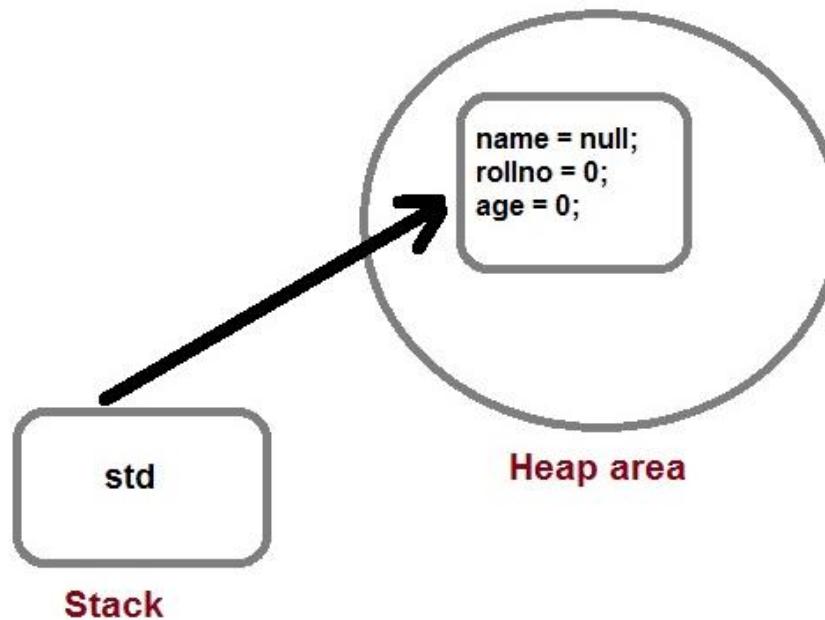
```
Student std
```

- ❖ Now create an object and assign it to reference variable

```
std= new Student();
```

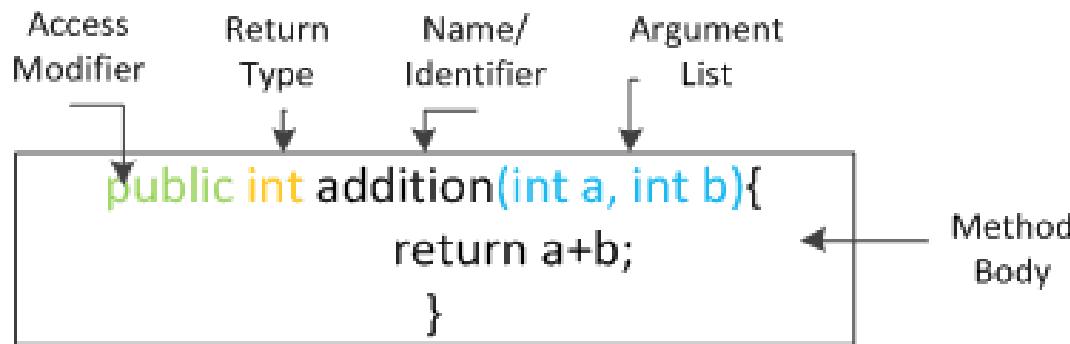
CLASS EXAMPLE

- ❖ After the above statement std is instance/object of Student class.
- ❖ Here the new keyword creates an actual physical copy of the object and assign it to the **std** variable.
- ❖ It will have physical existence and get memory in heap area.
- ❖ The new operator dynamically allocates memory for an object



METHODS

- ❖ A method is a program module that contains a series of statements that carry out a task.
- ❖ To execute a method, you invoke or call it from another method.



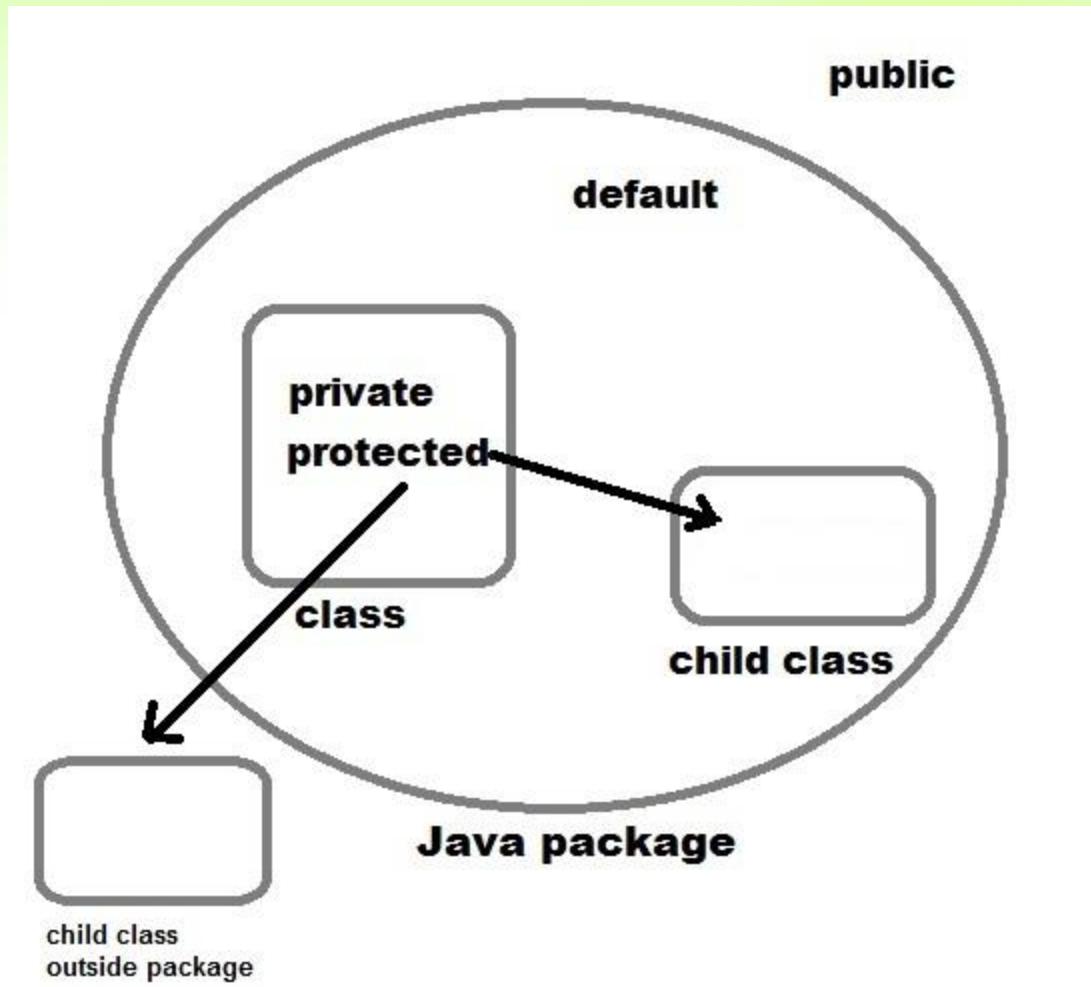
ACCESS MODIFIERS

- ❖ Each object has members (members can be variable and methods) which can be declared to have specific access.
- ❖ In Java, modifiers are categorized into two types,
 - ❖ Access control modifier
 - ❖ Non Access Modifier

ACCESS CONTROL MODIFIER

- ❖ Java language has four access modifier to control access levels for classes, variable methods and constructor.
- ❖ **Default** : Default has scope only inside the same package
- ❖ **Public** : Public has scope that is visible everywhere
- ❖ **Protected** : Protected has scope within the package and all sub classes
- ❖ **Private** : Private has scope only within the classes

ACCESS CONTROL MODIFIER



ACCESS CONTROL MODIFIER

Modifier	Class	Constructor	Method	Data/variables
Public	Yes	Yes	Yes	Yes
Protected		Yes	Yes	Yes
Default	Yes	Yes	Yes	Yes
Private		Yes	Yes	Yes

QUESTIONS

- ❖ Can we overload the main() method?
 - ❖ Yes, we can overload a main() method. A class can have any number of main() methods. But, one of those must be in the form “public static void main(String[] args)” in order to start the execution.
- ❖ What is an Instance variable?
 - ❖ Instance variables are variables within a class but outside any method. These variables are instantiated when the class is loaded.
- ❖ What is Java Virtual Machine and how it is considered in context of Java’s platform independent feature?
 - ❖ When Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.
- ❖ Can we have protected class?
 - ❖ No, we cannot have class or interface with protected modifier. Only fields, methods and constructors can be protected.

METHOD OVERLOADING

- ❖ If two or more method in a class have same name but different parameters, it is known as method overloading.
- ❖ Overloading always occur in the same class(unlike method overriding).
- ❖ Method overloading is one of the ways through which java supports polymorphism.
- ❖ Method overloading can be done by **changing number of arguments** or by **changing the data type of arguments**.

WHAT IS METHOD SIGNATURE

- ❖ A method signature is the method name and the number and type of its parameters.
- ❖ Return types and thrown exceptions are not considered to be a part of the method signature.

QUESTION - 1

```
class Area
{
    void find(int i,int j){
        System.out.println("method with int arguments");
    }

    void find(long i,long j){
        System.out.println("method with long arguments");
    }

    public static void main (String[] args)
    {
        Area ar = new Area();
        ar.find(2, 3);

    }
}
```

❖ Output:

QUESTION - 2

```
class Area
{
    void find(int i,int j){
        System.out.println("method with int arguments");
    }

    void find(long i,long j){
        System.out.println("method with long arguments");
    }

    public static void main (String[] args)
    {
        Area ar = new Area();
        ar.find(21, 31);

    }
}
```

❖Output:

QUESTION - 3

```
class Area
{
    void find(long i,long j){
        System.out.println("method with long arguments");
    }

    public static void main (String[] args)
    {
        Area ar = new Area();
        ar.find(2, 3);

    }
}
```

❖Output:

QUESTION - 4

```
class Area
{
    void find(int i,int j){
        System.out.println("method with int arguments");
    }

    public static void main (String[] args)
    {
        Area ar = new Area();
        ar.find(2L, 3L);

    }
}
```

❖Output:

QUESTION - 5

```
class Area
{
    void find(int i,int j){
        System.out.println("method with int arguments");
    }

    void find(long i,long j){
        System.out.println("method with long arguments");
    }

    public static void main (String[] args)
    {
        Area ar = new Area();
        ar.find(2, 3L);

    }
}
```

❖ Output:

QUESTION - 6

```
class Area
{
    void find(double i, double j){
        System.out.println("method with double arguments");
    }

    public static void main (String[] args)
    {
        Area ar = new Area();
        ar.find(2, 3);
    }
}
```

❖Output:

QUESTION - 7

```
class Area
{
    void find(float i, float j){
        System.out.println("method with float arguments");
    }

    void find(double i, double j){
        System.out.println("method with double arguments");
    }

    public static void main (String[] args)
    {
        Area ar = new Area();
        ar.find(2, 3);

    }
}
```

❖Output:

QUESTIONS

- ❖ Q – 8. Can we overload static methods in java?
- ❖ Q – 9. Can we use a field or a method declared without access modifiers outside the package.?
- ❖ Q – 10. Can you create a sub class to the following class?

```
class A
{
    private A()
    {
        //First Constructor
    }

    private A(int i)
    {
        //Second Constructor
    }
}
```

QUESTIONS

- ❖ Q – 11. Why we can't instantiate Class-A in the below code outside the package even though it has public constructor?

```
package pack1;

class A
{
    public A()
    {
        //public constructor
    }
}

package pack2;

import pack1.*;

class B
{
    A a = new A();           //Compile Time Error
}
```

QUESTIONS

- ❖ Q – 12. Can we declare static methods as private?
- ❖ Q – 13. What is the use of final keyword in java?
- ❖ Q – 14. Is it possible to declare final variables without initialization?
- ❖ Q – 15. Where all we can initialize a final non-static global variable if it is not initialized at the time of declaration?
- ❖ Q – 16. Where all we can initialize a final static global variable if it is not initialized at the time of declaration?
- ❖ Q – 17. Can we declare constructors as final?
- ❖ Q – 18. Can we create object for final class?

NON ACCESS MODIFIER

- ❖ Non-access modifiers do not change the accessibility of variables and methods, but they do provide them special properties. Non-access modifiers are of 5 types,

- ❖ Final
- ❖ Static
- ❖ Transient
- ❖ Synchronized
- ❖ Volatile

FINAL

- ❖ This modifier applicable to **class, method, and variables.**
- ❖ This modifier tells the compiler not to change the value of a variable once assigned.
- ❖ If applied to class, it cannot be sub-classed. If applied to a method, the method cannot be overridden in sub-class.
- ❖ A final method can be inherited/used in the subclass, but it cannot be overriden.

```
final int a = 5;  
  
class StudyJava{  
    final void learn(){System.out.println("learning  
something new");  
}  
}
```

FINAL

- ❖ A reference variable declared final can never be reassigned to refer to an different object.
- ❖ However, the data within the object can be changed. So, the state of the object can be changed but not the reference.

STATIC

- ❖ Static modifier is applicable to:
 - ❖ Method
 - ❖ Variable
 - ❖ Class nested within another class
 - ❖ Initialization block
- ❖ Static modifier is not applicable to:
 - ❖ Class (Not Nested)
 - ❖ Constructor
 - ❖ Interfaces
 - ❖ Method Local Inner Class
 - ❖ Inner Class methods
 - ❖ Instance Variables
 - ❖ Local Variables

STATIC

- ❖ Static Modifiers are used to create **class variable and class methods** which can be accessed **without instance of a class**.
- ❖ Static variables are defined as a class member that can be accessed without any object of that class.
- ❖ Static variable has only one single storage.
- ❖ All the object of the class having static variable will have the same instance of static variable.
- ❖ Static variables are initialized only once.

STATIC VARIABLE VS INSTANCE VARIABLE

Static variable	Instance Variable
Represent common property	Represent unique property
Accessed using class name	Accessed using object
get memory only once	get new memory each time a new object is created

STATIC METHOD

- ❖ A method can also be declared as static.
- ❖ Static methods do not need instance of its class for being accessed.
- ❖ main() method is the most common example of static method.
- ❖ main() method is declared as static because it is called before any object of the class is created.

STATIC METHOD

```
Vehicle.java X
1 package sct;
2
3 public class Vehicle {
4
5     private int doors;
6     private int speed;
7     private String color;
8
9     public static void run(){
10         //Static Run method implementation.
11     }
12
13     public void stop (){
14         //Implementation of Stop method
15     }
16 }
17
18 class Maruti {
19     public void TestVehicleClass(){
20         //To Access run() method we dont need object of Vehicle class
21         Vehicle.run();
22         // To Access stop() method we need object of Vehicle class, else compilation fails.
23         new Vehicle().stop();|
24     }
25 }
```

STATIC BLOCK

- ❖ Java supports a special block, called **static block** (also called static clause) which can be used for static initializations of a class.
- ❖ This code inside static block is executed only once: the first time you make an object of that class or the first time you access a static member of that class (even if you never make an object of that class).
- ❖ Also, static blocks are executed before constructors.

WHAT IS THE USE OF STATIC BLOCK

- ❖ Static block is mostly used for changing the default values of static variables. This block gets executed when the class is loaded in the memory.
- ❖ A class can have multiple Static blocks, which will execute in the same sequence in which they have been written into the program.

NON ACCESS MODIFIERS

- ❖ **Transient modifier**
- ❖ When an instance variable is declared as transient, then its value doesn't persist when an object is serialized

- ❖ **Synchronized modifier**
- ❖ When a method is synchronized it can be accessed by only one thread at a time.

- ❖ **Volatile modifier**
- ❖ The volatile modifier is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory.
- ❖ Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory. Volatile can only be applied to instance variables, which are of type object or private.

CONSTRUCTORS

- ❖ A constructor is a special method that is used to initialize an object.
- ❖ It is treated as a special member function because its name is the same as the class name.
- ❖ Java constructors are invoked when their objects are created
- ❖ Every class has a constructor.
- ❖ If we don't explicitly declare a constructor for any java class the compiler builds a default constructor for that class.
- ❖ A constructor does not have any return type.
- ❖ Constructor in Java can not be abstract, static, final or synchronized. These modifiers are not allowed for constructor.

TYPES OF CONSTRUCTORS

- ❖ **Default constructor** (no-arg constructor) - A constructor having no parameter is known as default constructor. It is also known as no-arg constructor.
- ❖ **Parameterized constructor** - A constructor having argument list is known as parameterized constructor.

CONSTRUCTOR OVERLOADING

- ❖ Like methods, a constructor can also be overloaded.
- ❖ Overloaded constructors are differentiated on the basis of their type of parameters or number of parameters.

```
class paramC{  
    paramC(int a, int b){  
        System.out.print("Parameterized Constructor");  
        System.out.println(" having Two parameters");  
    }  
    paramC(int a, int b, int c){  
        System.out.print("Parameterized Constructor");  
        System.out.println(" having Three parameters");  
    }  
}
```

THIS KEYWORD

- ❖ this keyword is used to refer to current object.
- ❖ this is always a reference to the object on which method was invoked.
- ❖ this can be used to invoke current class constructor.
- ❖ this can be passed as an argument to another method.

```
class Box
{
    Double width, height, depth;
    Box (double w, double h, double d)
    {
        this.width = w;
        this.height = h;
        this.depth = d;
    }
}
```

CALL OVERLOADED CONSTRUCTOR

```
public class ThisExample {  
  
    ThisExample() {  
        this("Java");  
        System.out.println("Inside  
Constructor without parameter");  
    }  
    ThisExample(String str) {  
        System.out  
            .println("Inside Constructor  
with String parameter as " + str);  
    }  
}
```

- ❖ this keyword can only be the first statement in Constructor.
- ❖ A constructor can have either this or super keyword but not both.

CALL METHOD OF THE CLASS

```
public void getName()
{
    System.out.println("StudyJava");
}

public void display()
{
    this.getName();
    System.out.println();
}
```

THIS AS METHOD PARAMETER

```
void method() {  
    method1(this);  
}  
void method1(JBT1 t) {  
    System.out.println(t.i);  
}
```

CONSTRUCTOR CHAINING

```
new Temp(8, 10); // invokes parameterized constructor 3
```

```
Temp(int x, int y)
{
    //invokes parameterized constructor 2
    this(5);
    System.out.println(x * y);
}
```

```
→Temp(int x)
{
    //invokes default constructor
    this();
    System.out.println(x);
}
```

```
→Temp()
{
    System.out.println("default");
}
```

ARRAYS

- ❖ An array is a collection of similar data types.
- ❖ Array is a container object that hold values of homogenous type.
- ❖ It is also known as static data structure because size of an array must be specified at the time of its declaration.
- ❖ An array can be either primitive or reference type. It gets memory in heap area. Index of array starts from zero to size-1.
- ❖ It occupies a contiguous memory location.

ARRAY DECLARATION

```
datatype[ ] identifier;  
or  
datatype identifier[ ];
```

```
int[ ] arr;  
char[ ] arr;  
short[ ] arr;  
long[ ] arr;  
int[ ][ ] arr; // two dimensional array.
```

INITIALIZATION OF ARRAY

- ❖ **new** operator is used to initialize an array.

```
int[ ] arr = new int[10]; //this creates an empty  
array named arr of integer type whose size is 10.  
or
```

```
int[ ] arr = {10,20,30,40,50}; //this creates an  
array named arr whose elements are given.
```

ACCESSING ARRAY ELEMENTS

- ❖ Array index starts from 0. To access nth element of an array. Syntax

```
arrayname[n-1];
```

- ❖ To find the length of an array, we can use the following syntax:
array_name.length.

FOR EACH LOOP

- ❖ J2SE 5 introduces special type of for loop called **foreach** loop to access elements of array.
- ❖ Using **foreach** loop you can access complete array sequentially without using index of array.

```
int[] arr = {10, 20, 30, 40};  
for(int x : arr)  
{
```

COPYING ARRAY

- ❖ Copying An Array Using for Loop.

```
int[] a = {12, 21, 0, 5, 7};    //Declaring and  
initializing an array of ints  
  
        int[] b = new int[a.length];  
//Declaring and instantiating another array of ints  
with same length  
  
        for (int i = 0; i < a.length; i++)  
        {  
            b[i] = a[i];  
        }
```

COPYING ARRAY

- ❖ Copying An Array Using **copyOf()** Method of `java.util.Array` Class

```
int[] a = {12, 21, 0, 5, 7}; //Declaring and  
initializing an array of ints
```

```
//creating a copy of array 'a' using copyOf()  
method of java.util.Arrays class
```

```
int[] b = Arrays.copyOf(a, a.length);
```

COPYING ARRAY

- ❖ Copying An Array Using **clone()** Method.
- ❖ All arrays will have **clone()** method inherited from **java.lang.Object** class. Using this method, you can copy an array.

```
int[] a = {12, 21, 0, 5, 7};    //Declaring and  
initializing an array of ints  
  
                                //creating a copy of array 'a' using clone()  
method  
  
int[] b = a.clone();
```

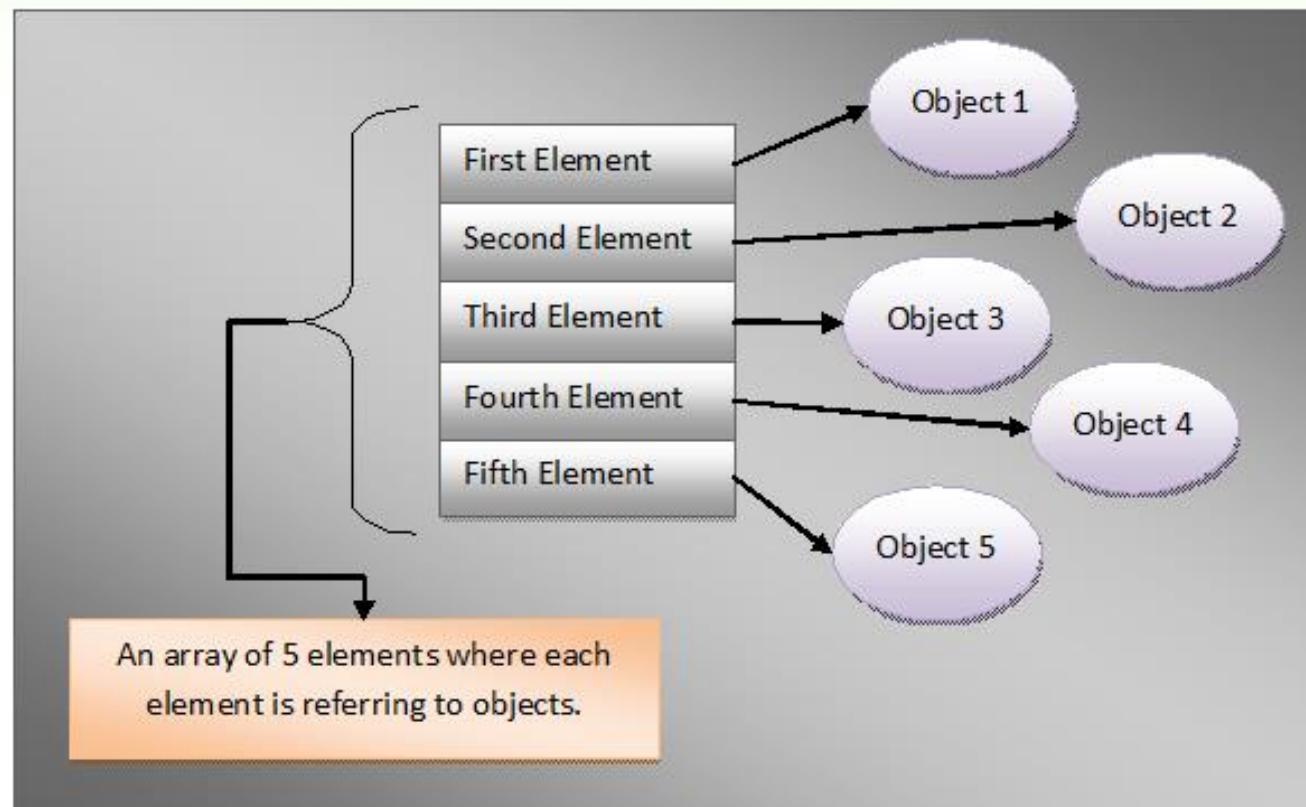
COPYING ARRAY

- ❖ Copying An Array Using **arraycopy()** Method Of **System Class**.
- ❖ Using **arraycopy()** method, you can copy a part of an array into another array.

```
int[] a = {12, 21, 0, 5, 7};    //Declaring and  
initializing an array of ints  
  
                                //Creating an array object of same length as  
array 'a'  
  
int[] b = new int[a.length];  
  
                                //creating a copy of array 'a' using  
arraycopy() method of System class  
  
System.arraycopy(a, 0, b, 0, a.length);
```

ARRAYS OF OBJECTS

- ❖ Array can hold the references to any type of objects.
- ❖ It is important to note that array can contain only references to the objects, not the objects itself.



ARRAYS AS PARAMETERS

- ❖ Arrays can be passed to method as arguments and methods can return an array.
- ❖ Arrays are **Passed-By-Reference**. That means, When an array is passed to a method, reference of an array object is passed not the copy of the object.
- ❖ So, Any changes made to object in the method will be reflected in the actual object.

MULTI-DIMENSIONAL ARRAY

- ❖ A multi-dimensional array is very much similar to a single dimensional array.
- ❖ It can have multiple rows and multiple columns unlike single dimensional array

```
datatype[ ][ ] identifier;  
int[ ][ ] arr = new int[10][10];
```

JAGGED ARRAY

- ❖ Jagged arrays in java are arrays containing arrays of different length.
- ❖ Jagged arrays are also multidimensional arrays. Jagged arrays in java sometimes are also called as **ragged arrays**.

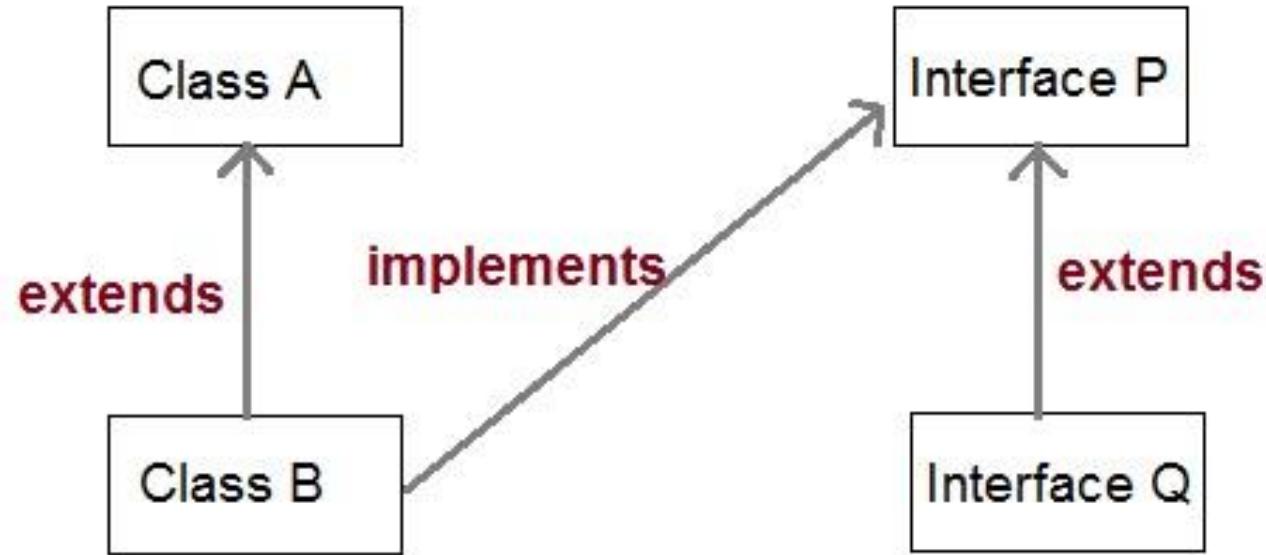
```
int[ ][ ] arr = new int[3][ ];      //there will be 10  
arrays whose size is variable  
arr[0] = new int[3];  
arr[1] = new int[4];  
arr[2] = new int[5];
```

INHERITANCE (IS-A)

- ❖ **Inheritance** is one of the key features of Object Oriented Programming.
- ❖ Inheritance provided mechanism that allowed a class to inherit property of another class.
- ❖ When a Class extends another class it inherits all non-private members including fields and methods.
- ❖ Inheritance in Java can be best understood in terms of Parent and Child relationship, also known as **Super class(Parent)** and **Sub class(child)** in Java language.

INHERITANCE (IS-A)

- ❖ Inheritance defines is-a relationship between a Super class and its Sub class.
- ❖ **extends** and **implements** keywords are used to describe inheritance in Java.



INHERITANCE

```
class Vehicle.  
{  
    .....  
}  
class Car extends Vehicle  
{  
    ..... //extends the property of vehicle class.  
}
```

- ❖ Now based on above example. In OOPs term we can say that,
- ❖ Vehicle is super class of Car.
- ❖ Car is sub class of Vehicle.
- ❖ Car **IS-A** Vehicle.

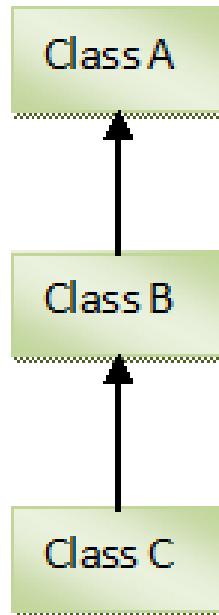
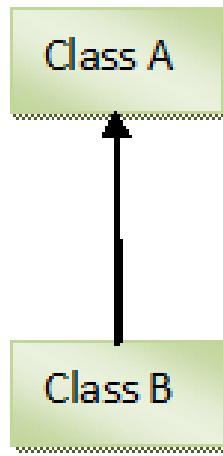
PURPOSE OF INHERITANCE

- ❖ It promotes the code **reusability** i.e. the same methods and variables which are defined in a parent/super/base class can be used in the child/sub/derived class.
- ❖ It promotes polymorphism by allowing method overriding.
- ❖ Main **disadvantage** of using inheritance is that the two classes (parent and child class) **gets tightly coupled**.
- ❖ This means that if we change code of parent class, it will affect to all the child classes which is inheriting/deriving the parent class, and hence, it **cannot be independent** of each other.

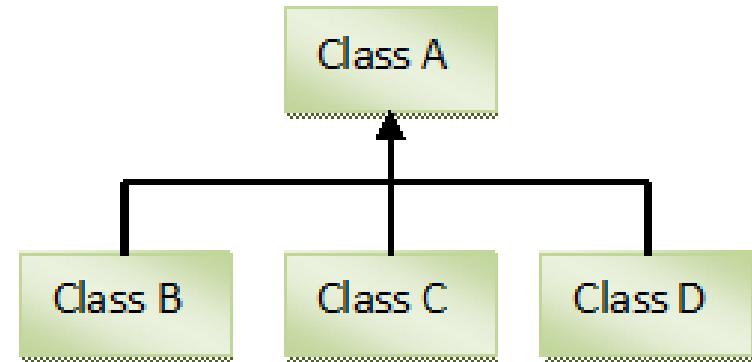
INHERITANCE - EXAMPLE

```
class Parent
{
    public void p1()
    {
        System.out.println("Parent method");
    }
}
public class Child extends Parent {
    public void c1()
    {
        System.out.println("Child method");
    }
    public static void main(String[] args)
    {
        Child cobj = new Child();
        cobj.c1();    //method of Child class
        cobj.p1();    //method of Parent class
    }
}
```

TYPES OF INHERITANCE IN JAVA



Single Inheritance

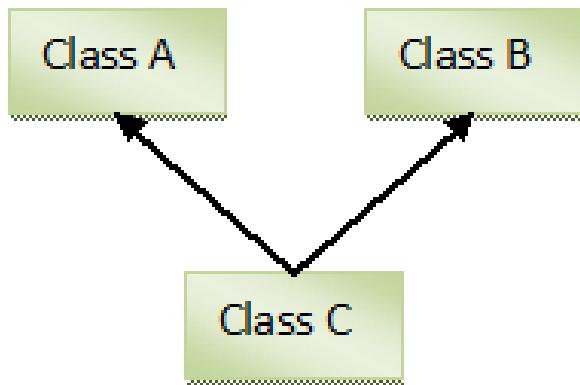


Hierarchical Inheritance

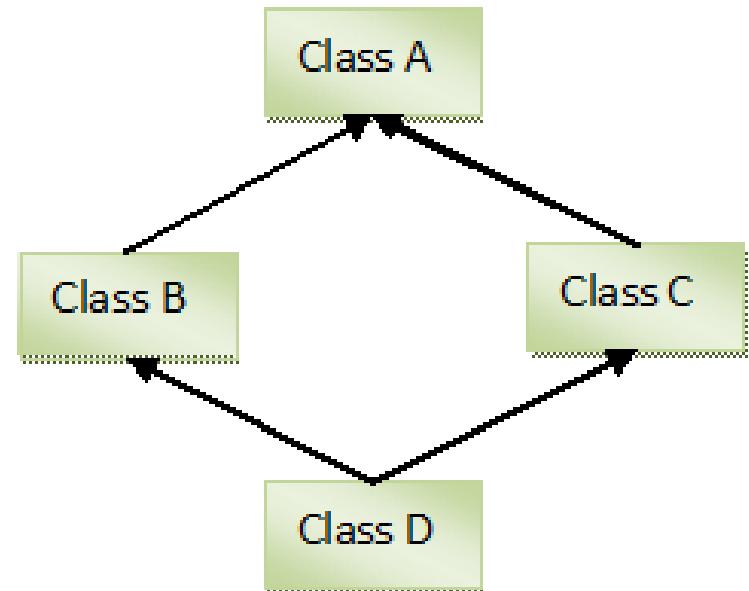
Multilevel Inheritance

INHERITANCE NOT SUPPORTED IN JAVA

Multiple inheritance is not supported in java.



Multiple Inheritance



Hybrid Inheritance

SUPER KEYWORD

- ❖ In Java, **super** keyword is used to refer to immediate parent class of a child class.
- ❖ In other words super keyword is used by a subclass whenever it need to refer to its immediate super class.
- ❖ Usage of java super Keyword
 - ❖ super can be used to refer immediate parent class instance variable.
 - ❖ super can be used to invoke immediate parent class method.
 - ❖ super() can be used to invoke immediate parent class constructor.

SUPER KEYWORD

```
class Parent
{
    String name;
}
class Child extends Parent {

    String name;

    void detail()
    {
        super.name = "Parent";
        name = "Child";
    }
}
```

METHOD OVERRIDING

- ❖ When a class extends its super class, all or some members of super class are inherited to sub class.
- ❖ When a inherited super class method is modified in the sub class, then we call it as **method overriding** .
- ❖ Through method overriding, we can modify super class method according to requirements of sub class.

RULES OF METHOD OVERRIDING

- ❖ **Name of the overridden method** must be same as in the super class. You can't change name of the method in subclass.
- ❖ **Return Type Of Overridden Method :**
 - ❖ The return type of the overridden method must be compatible with super class method.
 - ❖ If super class method has primitive data type as its return type, then overridden method must have same return type in sub class also.
 - ❖ If super class method has derived or user defined data type as its return type, then return type of sub class method must be of same type or its sub class.

RULES OF METHOD OVERRIDING

❖ Visibility Of Overridden method :

- ❖ You can keep same visibility or increase the visibility of overridden method but you can't reduce the visibility of overridden methods in the subclass.
- ❖ For example, default method can be overridden as default or protected or public method but not as private

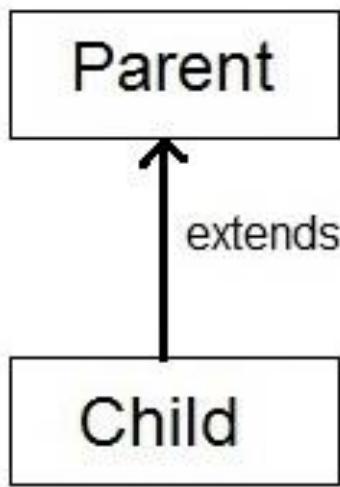
❖ Arguments Of Overridden Methods :

- ❖ For method to be properly overridden , You must not change arguments of method in subclass.
- ❖ If you change the number of arguments or types of arguments of overridden method in the subclass, then method will be overloaded not overridden .

INSTANCEOF OPERATOR

- ❖ In Java, **instanceof** operator is used to check the type of an object at runtime.
- ❖ It is the means by which your program can obtain run-time type information about an object.
- ❖ **instanceof** operator is also important in case of casting object at runtime.
- ❖ **instanceof** operator return **boolean** value, if an object reference is of specified type then it return **true** otherwise **false**.

INSTANCEOF OPERATOR



Parent p = new Child();
Upcasting

~~Child c = new Parent();~~
Compile time error

Child c = (Child) new Parent();
Downcasting but throws
ClassCastException at runtime.

ABSTRACT CLASS

- ❖ If a class contain any abstract method then the class is declared as abstract class.
- ❖ An abstract class is never instantiated. It is used to provide abstraction.
- ❖ Although it does not provide 100% abstraction because it can also have concrete method.

```
abstract class class_name { }
```

ABSTRACT CLASSES

- ❖ Abstract classes are not Interfaces.
- ❖ An abstract class may or may not have an abstract method. But if any class has even a single abstract method, then it must be declared abstract.
- ❖ Abstract classes can have Constructors, Member variables and Normal methods.
- ❖ Abstract classes are never instantiated.
- ❖ When you extend Abstract class with abstract method, you must define the abstract method in the child class, or make the child class abstract.

ABSTRACT METHOD

- ❖ Method that are declared without any body within an abstract class are called abstract method.
- ❖ The method body will be defined by its subclass.
- ❖ Abstract method can never be final and static.
- ❖ Any class that extends an abstract class must implement all the abstract methods declared by the super class.

```
abstract return_type function_name (); // No  
definition
```

ABSTRACTION USING ABSTRACT CLASS

- ❖ Abstraction is an important feature of OOPS.
- ❖ It means hiding complexity.
- ❖ Abstract class is used to provide abstraction.
- ❖ Although it does not provide 100% abstraction because it can also have concrete method.

QUESTION - 1

```
public class ClassA {  
    public void methodOne(int i) {  
    }  
    public void methodTwo(int i) {  
    }  
    public static void methodThree(int i) {  
    }  
    public static void methodFour(int i) {  
    }  
}  
  
public class ClassB extends ClassA {  
    public static void methodOne(int i) {  
    }  
    public void methodTwo(int i) {  
    }  
    public void methodThree(int i) {  
    }  
    public static void methodFour(int i) {  
    }  
}
```

- ❖ Which method overrides a method in the superclass?
- ❖ Which method hides a method in the superclass?
- ❖ What do the other methods do?

QUESTION2

```
class Base {  
    public void foo() { System.out.println("Base"); }  
}  
  
class Derived extends Base {  
    private void foo() { System.out.println("Derived"); }  
}  
  
public class Main {  
    public static void main(String args[]) {  
        Base b = new Derived();  
        b.foo();  
    }  
}
```

❖Output ?

QUESTION - 3

```
class A
{
}

class B extends A
{
}

class C extends B
{
}

public class MainClass
{
    static void overloadedMethod(A a)
    {
        System.out.println("ONE");
    }

    static void overloadedMethod(B b)
    {
        System.out.println("TWO");
    }

    static void overloadedMethod(Object obj)
    {
        System.out.println("THREE");
    }

    public static void main(String[] args)
    {
        C c = new C();
        overloadedMethod(c);
    }
}
```

❖Output ?

QUESTION - 4

```
class GrandParent{}  
class Parent1 extends GrandParent{}  
class child1 extends Parent1{}  
  
public class MainClass {  
  
    static void overLoadedmethod(GrandParent gp){  
        System.out.println("GrandParent");  
    }  
  
    static void overLoadedmethod(Parent1 p){  
        System.out.println("Parent");  
    }  
  
    static void overLoadedmethod(Object obj){  
        System.out.println("Object");  
    }  
  
    public static void main(String[] args) {  
        child1 ch = new child1();  
        overLoadedmethod(ch);  
    }  
}
```

❖Output ?

QUESTION - 5

```
class GrandParent{}  
class Parent1 extends GrandParent{}  
class child1 extends Parent1{}  
  
public class MainClass {  
  
    static void overLoadedmethod(GrandParent gp){  
        System.out.println("GrandParent");  
    }  
    static void overLoadedmethod(Object obj){  
        System.out.println("Object");  
    }  
    public static void main(String[] args) {  
        child1 ch = new child1();  
        overLoadedmethod(ch);  
    }  
}
```

❖Output ?

QUESTION - 6

```
class GrandParent{}  
class Parent1 extends GrandParent{}  
class child1 extends Parent1{}  
  
public class MainClass {  
  
    static void overLoadedmethod(GrandParent gp){  
        System.out.println("GrandParent");  
    }  
    static void overLoadedmethod(Parent1 p){  
        System.out.println("Parent");  
    }  
    static void overLoadedmethod(child1 ch){  
        System.out.println("Child");  
    }  
    static void overLoadedmethod(Object obj){  
        System.out.println("Object");  
    }  
    public static void main(String[] args) {  
        Parent1 ch = new child1();  
        overLoadedmethod(ch);  
    }  
}
```

❖Output ?

QUESTIONS

- ❖ Q4. What is inheritance?
- ❖ Q5. Can we pass an object of a subclass to a method expecting an object of the super class?
- ❖ Q6. How to call a method of a subclass, if you are holding an object of the subclass in a reference variable of type superclass?
- ❖ Q7. Difference Between this() and super() ?
- ❖ Q8. Is it possible to override non-static methods as static?

INTERFACE

- ❖ Interface is a **pure abstract** class.
- ❖ They are syntactically similar to classes, but you cannot create instance of an Interface and their methods are declared without any body.
- ❖ Interface is used to achieve complete abstraction in Java.
- ❖ When you create an interface it defines what a class can do without saying anything about how the class will do it.
- ❖ Abstract classes may contain both abstract methods as well as concrete methods.
- ❖ But interfaces must contain **only abstract methods**. Concrete methods are not allowed in interfaces.

INTERFACE

- ❖ Interfaces are declared with keyword ‘**interface**’ and interfaces are implemented by the class using ‘**implements**’ keyword.

```
interface InterfaceClass
{
    //Some Abstract methods
}

class AnyClass implements InterfaceClass
{
    //Use 'implements' while implementing Interfaces
    //Don't use 'extends'
}
```

INTERFACE

- ❖ By default, Every field of an interface is public, static and final. You can't use any other modifiers other than these three for a field of an interface.
- ❖ You can't change the value of a field once they are initialized. Because they are static and final. Therefore, sometimes fields are called as **Constants**.
- ❖ By default, All **methods** of an interface are **public** and **abstract**.
- ❖ Like classes, for every interface **.class** file will be generated after compilation.
- ❖ While implementing any interface methods inside a class, that method must be declared as **public**.

INTERFACE

- ❖ SIB – Static Initialization Block and IIB – Instance Initialization Block are not allowed in interfaces.

```
interface InterfaceClassOne
{
    static
    {
        //compile time error
        //SIB's are not allowed in interfaces
    }

    {
        //Here also compile time error.
        //IIB's are not allowed in interfaces
    }

    void methodOne(); //abstract method
}
```

INTERFACE

- ❖ As we all know that, any class in java can not extend more than one class. But class can implement more than one interfaces. This is how multiple inheritance is implemented in java.

```
class AnyClass implements InterfaceClassOne, InterfaceClassTwo
{
    public void methodOne()
    {
        //method of first interface is implemented
    }

    //method of Second interface must also be implemented.
    //Otherwise, you have to declare this class as abstract.

    public void methodTwo()
    {
        //Now, method of Second interface is also implemented.
        //No need to declare this class as abstract
    }
}
```

INTERFACE

```
interface Moveable
```

```
    int AVERAGE-SPEED=40;  
    void move();
```

what you declare

```
interface Moveable
```

```
    public static final int AVERAGE-SPEED=40;  
    public abstract void move();
```

what the compiler
sees

IMPLEMENTING INTERFACES

- ❖ A class can choose to implement only some of the methods of its interfaces. The class then be declared as **abstract**.
- ❖ The interface methods cannot be declared *static* because they comprise the *contract* fulfilled by the objects of the class implementing the interface.
- ❖ Interface methods are always implemented as instance methods.

EXTENDING INTERFACES

- ❖ An interface can extend other interfaces using extend clause. Unlike extending classes, an interface can extend several interfaces.
- ❖ The interfaces extended by an interface (directly or indirectly) are called *superinterfaces*.
- ❖ A **subinterface** inherits all methods from its **superinterfaces**, as their method declarations are all implicitly public.

DOES AN INTERFACE EXTEND OBJECT CLASS IN JAVA.?

- ❖ If an interface does not extend Object class, then why we can call methods of Object class on interface variable like below.

```
interface A
{
}

class InterfaceAndObjectClass
{
    public static void main(String[] args)
    {
        A a = null;

        a.equals(null);

        a.hashCode();

        a.toString();
    }
}
```

DOES AN INTERFACE EXTEND OBJECT CLASS IN JAVA.?

- ❖ If an interface does not extend Object class, then why the methods of Object class are visible in interface.?

```
interface A
{
    @Override
    public boolean equals(Object obj);

    @Override
    public int hashCode();

    @Override
    public String toString();
}
```

DOES AN INTERFACE EXTEND OBJECT CLASS IN JAVA.?

- ❖ This is because, for every public method in Object class, there is an implicit abstract and public method declared in every interface which does not have direct super interfaces.
- ❖ This is the standard Java Language Specification

INTERFACES

- ❖ Interfaces allow you to use classes in different hierarchies, polymorphically. For example, say you have the following interface:

```
public interface Movable {  
    void move();  
}
```

- ❖ Any number of classes, across class hierarchies could implement Movable in their own specific way, yet still be used by some caller in a uniform way. So if you have the following two classes:

```
public class Car extends Vehicle implements Movable {  
    public void move() {  
        //implement move, vroom, vroom!  
    }  
}
```

INTERFACES

```
public class Horse extends Animal implements Movable
{
    public void move() {
        //implement move, neigh!
    }
}
```

❖ From the perspective of the caller, it's just a Movable

- ❖ Movable movable = ...;
- ❖ movable.move();

MARKER INTERFACES

- ❖ Marker Interface in java is an interface with no fields or methods within it. It is used to convey to the JVM that the class implementing an interface of this category will have some special behavior.
- ❖ Hence, an empty interface in java is called a marker interface. Marker interface is also called tag interface. In java we have the following major marker interfaces as under:
 - ❖ Serializable interface
 - ❖ Cloneable interface
 - ❖ Remote interface
 - ❖ ThreadSafe interface

JAVA8 - DEFAULT KEYWORD

- ❖ From Java 8 onwards, the **default** keyword is also used to specify that a method in an interface provides the default implementation of a method.

```
public interface Interface1 {  
    void method1(String str);  
  
    default void log(String str){  
        System.out.println("I1 logging::"+str);  
    }  
}
```

INTERFACE – DEFAULT METHOD

- ❖ Java interface default methods will help us in extending interfaces without having the fear of breaking implementation classes.
- ❖ Modifying one interface in JDK framework breaks all classes that extends the interface which means that adding any new method could break millions of lines of code.
- ❖ Java interface default methods has bridge down the differences between interfaces and abstract classes.
- ❖ A default method cannot override a method from `java.lang.Object`.
- ❖ Java interface default methods are also referred to as Defender Methods or Virtual extension methods.

MULTIPLE INHERITANCE AMBIGUITY

- ❖ Since java class can implement multiple interfaces and each interface can define default method with same method signature, therefore, the inherited methods can conflict with each other.

```
public interface InterfaceA {  
    default void defaultMethod(){  
        System.out.println("Interface A default method");  
    }  
}  
public interface InterfaceB {  
    default void defaultMethod(){  
        System.out.println("Interface B default method");  
    }  
}  
public class Impl implements InterfaceA, InterfaceB {  
}
```

MULTIPLE INHERITANCE AMBIGUITY

- ❖ The above code will fail to compile with the following error,
- ❖ java: class Impl inherits unrelated defaults for defaultMethod() from types InterfaceA and InterfaceB
- ❖ In order to fix this class, we need to provide default method implementation:

```
public class Impl implements InterfaceA, InterfaceB {  
    public void defaultMethod(){  
    }  
}
```

MULTIPLE INHERITANCE AMBIGUITY

- ❖ Further, if we want to invoke default implementation provided by any of super interface rather than our own implementation, we can do so as follows,

```
public class Impl implements InterfaceA, InterfaceB {  
    public void defaultMethod(){  
        // existing code here..  
        InterfaceA.super.defaultMethod();  
    }  
}
```

INTERFACE – DEFAULT METHOD

- ❖ When we extend an interface that contains a default method, we can perform following,
- ❖ Not override the default method and will inherit the default method.
- ❖ Override the default method similar to other methods we override in subclass..
- ❖ Redeclare default method as abstract, which force subclass to override it.

INTERFACE – STATIC METHOD

- ❖ Java interface static method is similar to default method except that we can't override them in the implementation classes.
- ❖ Static method belongs only to Interface class, so you can only invoke static method on Interface class, not on class implementing this Interface.
- ❖ Both class and interface can have static methods with same names, and neither overrides other.
- ❖ A static interface method cannot know about the **this** variable, but a default implementation can.
- ❖ Normally, static method in interface is used as Helper methods while default method are used as a default implementation for classes that implements that interface.

QUESTIONS

- ❖ Q1. What is an Interface in Java?
- ❖ Q2. Can we create non static variables in an interface?
- ❖ Q3. .What will happen if we are not implementing all the methods of an interface in class which implements an interface?
- ❖ Can we declare an Interface with “abstract” keyword?
- ❖ Can we use “abstract” keyword with constructor, Instance Initialization Block and Static Initialization Block?
- ❖ We can’t instantiate an abstract class. Then why constructors are allowed in abstract class?

QUESTIONS

❖ Output ?

```
abstract class AbstractClass
{
    abstract void abstractMethod()
    {
        System.out.println("First Method");
    }
}
```

QUESTIONS

- ❖ Identify error?

```
abstract class AbstractClass
{
    private abstract int abstractMethod();
}
```

QUESTIONS

❖ Output?

```
abstract class X
{
    public X()
    {
        System.out.println("ONE");
    }

    abstract void abstractMethod();
}

class Y extends X
{
    public Y()
    {
        System.out.println("TWO");
    }

    @Override
    void abstractMethod()
    {
        System.out.println("THREE");
    }
}

public class MainClass
{
    public static void main(String[] args)
    {
        X x = new Y();

        x.abstractMethod();
    }
}
```

QUESTIONS

❖ Output?

```
interface X
{
    void methodX();
}

class Y implements X
{
    void methodX()
    {
        System.out.println("Method X");
    }
}
```

ASSOCIATION

- ❖ Association establish relationship between two classes through their objects. The relationship can be one to one, One to many, many to one and many to many.
- ❖ Association is a relationship between two separate classes which can be of any type say one to one, one to may etc. It joins two entirely separate entities.

AGGREGATION

- ❖ Aggregation is a special form of association which is a unidirectional one way relationship between classes (or entities).
- ❖ It represents a Has-A relationship.
- ❖ for e.g. Wallet and Money classes. Wallet has Money but money doesn't need to have Wallet necessarily so its a one directional relationship.
- ❖ In this relationship both the entries can survive if other one ends.
- ❖ If Wallet class is not present, it does not mean that the Money class cannot exist.

WHY WE NEED AGGREGATION?

- ❖ To maintain code re-usability.
- ❖ Suppose there are two other classes College and Staff along with above two classes Student and Address.
- ❖ In order to maintain Student's address, College Address and Staff's address we don't need to use the same code again and again. We just have to use the reference of Address class while defining each of these classes like:
 - Student Has-A Address (Has-a relationship between student and address)
 - College Has-A Address (Has-a relationship between college and address)
 - Staff Has-A Address (Has-a relationship between staff and address)

COMPOSITION

- ❖ Composition is a restricted form of Aggregation in which two entities (or you can say classes) are highly dependent on each other.
- ❖ For e.g. Human and Heart. A human needs heart to live and a heart needs a Human body to survive.
- ❖ In other words when the classes (entities) are dependent on each other and their life span are same (if one dies then another one too) then its a composition.
- ❖ Heart class has no sense if Human class is not present.

RELATIONSHIP

- ❖ Manager is an employee of XYZ limited corporation.
- ❖ Manager uses a swipe card to enter XYZ premises.
- ❖ Manager has workers who work under him.
- ❖ Manager has the responsibility of ensuring that the project is successful.
- ❖ Manager's salary will be judged based on project success.

EXCEPTION HANDLING

- ❖ An exception (or exceptional event) is a problem that arises during the execution of a program.
- ❖ When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

EXCEPTION HANDLING

- ❖ An exception can occur for many different reasons. Following are some scenarios where an exception occurs.
- ❖ A user has entered an invalid data.
- ❖ A file that needs to be opened cannot be found.
- ❖ A network connection has been lost in the middle of communications or the JVM has run out of memory.
- ❖ Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

EXCEPTION HANDLING

- ❖ An exception can occur for many different reasons. Following are some scenarios where an exception occurs.
- ❖ A user has entered an invalid data.
- ❖ A file that needs to be opened cannot be found.
- ❖ A network connection has been lost in the middle of communications or the JVM has run out of memory.
- ❖ Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

EXCEPTION HANDLING

- ❖ Based on these, we have three categories of Exceptions:
 - ❖ **Checked exceptions** – A checked exception is an exception that occurs at the compile time, these are also called as compile time exceptions.
 - ❖ **Unchecked exceptions** – An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API.
 - ❖ **Errors** – These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

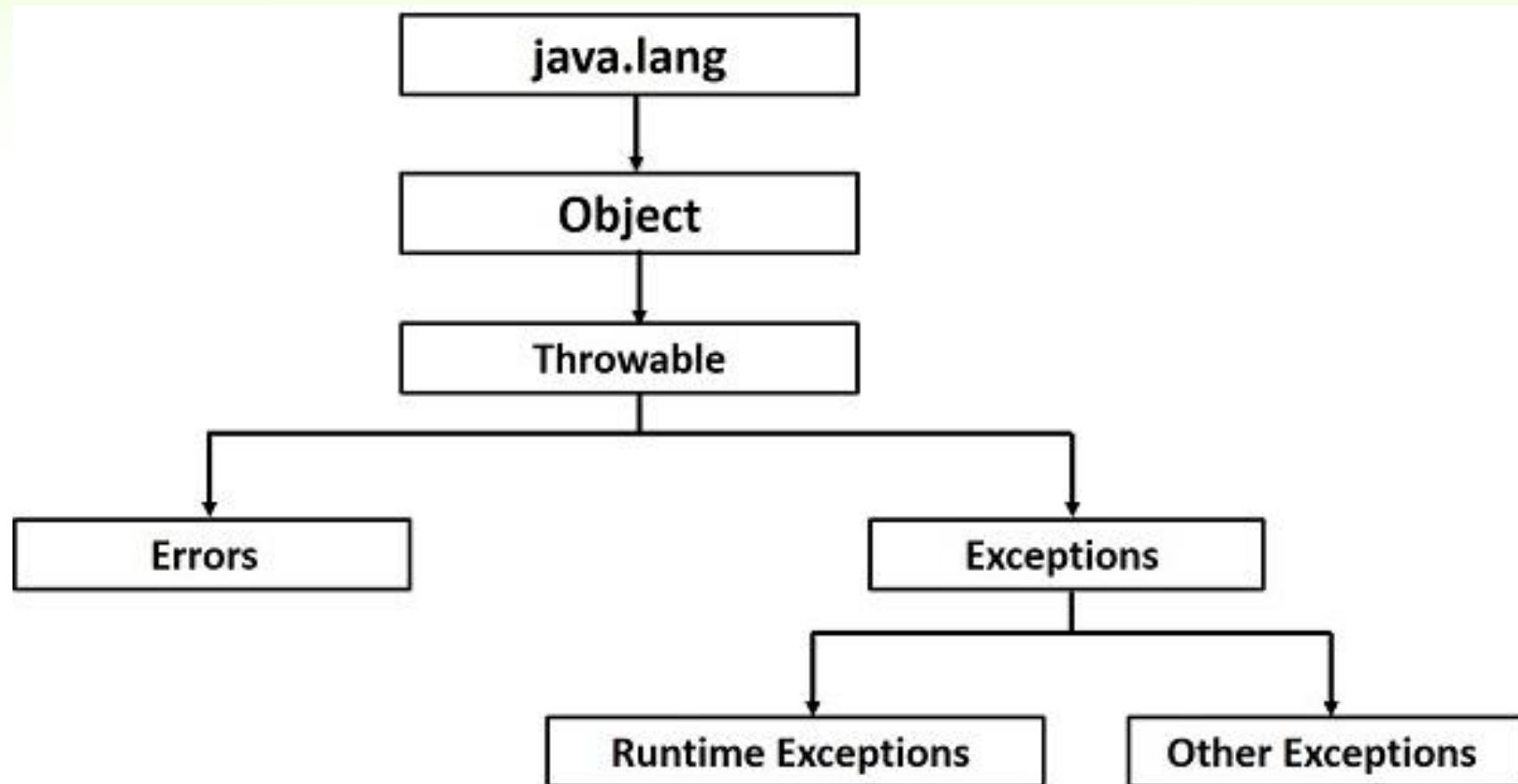
EXCEPTION HIERARCHY

- ❖ All exception classes are subtypes of the **java.lang.Exception** class.
- ❖ The exception class is a subclass of the **Throwable** class.
- ❖ Other than the exception class there is another subclass called **Error** which is derived from the **Throwable** class.

- ❖ Errors are abnormal conditions that happen in case of severe failures, these are not handled by the Java programs. Errors are generated to indicate errors generated by the runtime environment. Example: JVM is out of memory. Normally, programs cannot recover from errors.

EXCEPTION HIERARCHY

- ❖ The Exception class has two main subclasses: **IOException** class and **RuntimeException** Class.



EXCEPTION METHODS

- ❖ **public String getMessage()** – Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
- ❖ **public String toString()** – Returns the name of the class concatenated with the result of getMessage().
- ❖ **public void printStackTrace()** – Prints the result of toString() along with the stack trace to System.err, the error output stream.

CATCHING EXCEPTIONS

- ❖ A method catches an exception using a combination of the **try and catch keywords**.
- ❖ A try/catch block is placed around the code that might generate an exception.
- ❖ Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like

```
try {  
    // Protected code  
    catch(ExceptionName e1) {  
        // Catch block  
    }  
}
```

CATCHING EXCEPTIONS

- ❖ The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it.
- ❖ Every try block should be immediately followed either by a catch block or finally block.
- ❖ A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked.

CATCHING MULTIPLE TYPE EXCEPTIONS

- ❖ Since Java 7, you can handle more than one exception using a single catch block, this feature simplifies the code.

```
catch (IOException|FileNotFoundException ex) {  
    logger.log(ex);
```

CATCHING EXCEPTIONS

- ❖ The code which is prone to exceptions is placed in the try block. When an exception occurs, that exception occurred is handled by catch block associated with it.
- ❖ Every try block should be immediately followed either by a catch block or finally block.
- ❖ A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked.

THROW/THROWS KEYWORD

- ❖ If a method does not handle a **checked exception**, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.
- ❖ You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword.

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

TRY WITH RESOURCES

- ❖ **try-with-resources**, also referred as **automatic resource management**, is a new exception handling mechanism that was introduced in Java 7, which automatically closes the resources used within the try catch block.
- ❖ To use this statement, you simply need to declare the required resources within the parenthesis, and the created resource will be closed automatically at the end of the block.

```
try(FileReader fr = new FileReader("file path")) {  
    // use the resource  
}catch() {  
    // body of catch  
}  
}
```

THE FINALLY BLOCK

- ❖ The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception.
- ❖ Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

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

NESTED TRY CATCH BLOCKS

- ❖ In Java, try-catch blocks can be nested. i.e one try block can contain another try-catch block. The syntax for nesting try blocks is,

```
try      //Outer try block
{
    try      //Inner try block
    {
        //Some Statements
    }
    catch (Exception ex)      //Inner catch block
    {
    }
}
catch(Exception ex)      //Outer catch block
{
}
}
```

NESTED TRY CATCH BLOCKS

- ❖ Nested try blocks are useful when different statements of try block throw different types of exceptions.
- ❖ If the exception thrown by the inner try block can not be caught by its catch block, then this exception is propagated to outer try blocks. Any one of the outer catch block should handle this exception otherwise program will terminate abruptly.

RETURN VALUE

- ❖ If finally block returns a value then try and catch blocks may or may not return a value.
- ❖ If finally block does not return a value then both try and catch blocks must return a value.
- ❖ If try-catch-finally blocks are returning a value according to above rules, then you should not keep any statements after finally block. Because they become unreachable and in Java, Unreachable code gives compile time error.
- ❖ finally block overrides any return values from try and catch blocks.
- ❖ finally block will be always executed even though try and catch blocks are returning the control.

THROWING AN EXCEPTION

- ❖ Throwable class is super class for all types of errors and exceptions.
- ❖ An object to this Throwable class or it's sub classes can be created in two ways.
- ❖ First one is using an argument of catch block. In this way, Throwable object or object to it's sub classes is implicitly created and thrown by java run time system.
- ❖ Second one is using **new operator**. In this way, Throwable object or object to it's sub classes is explicitly created and thrown by the code.

METHOD OVERRIDING WITH THROWS

- ❖ Rules need to follow when overriding a method with throws clause.
- ❖ If super class method is not throwing any exceptions, then it can be overridden with any unchecked type of exceptions, but can not be overridden with checked type of exceptions.
- ❖ If a super class method is throwing unchecked exception, then it can be overridden in the sub class with same exception or any other unchecked exceptions but can not be overridden with checked exceptions.
- ❖ If super class method is throwing checked type of exception, then it can be overridden with same exception or with it's sub class exceptions i.e you can decrease the scope of the exception, but can not be overridden with it's super class exceptions i.e you can not increase the scope of the exception.

USER DEFINED EXCEPTIONS

- ❖ We can define our own exception classes as per our requirements. These exceptions are called **user defined exceptions** in java OR **Customized exceptions**.
- ❖ User defined exceptions must extend any one of the classes in the hierarchy of exceptions.
- ❖ If the user defined class is extending Exception class, then checked exception is created.
- ❖ If the user defined Exception class is extending RunTimeException Class, then unchecked exception is created.

USER DEFINED EXCEPTIONS

- ❖ We can define our own exception classes as per our requirements. These exceptions are called **user defined exceptions** in java OR **Customized exceptions**.
- ❖ User defined exceptions must extend any one of the classes in the hierarchy of exceptions.
- ❖ If the user defined class is extending Exception class, then checked exception is created.
- ❖ If the user defined Exception class is extending RunTimeException Class, then unchecked exception is created.

STRINGS

- ❖ String represents sequence of characters enclosed within the double quotes. “abc”, “JAVA”, “123”, “A” are some examples of strings.
 - ❖ In many languages, strings are treated as character arrays.
 - ❖ But In java, strings are treated as objects.
 - ❖ String class is encapsulated under **java.lang** package.
 - ❖ To create and manipulate the strings, Java provides three classes.
 - 1) `java.lang.String`
 - 2) `java.lang.StringBuffer`
 - 3) `java.lang.StringBuilder`

STRINGS

- ❖ All these three classes are members **of java.lang** package and they are final classes. That means you can't create subclasses to these three classes.
- ❖ **java.lang.String** objects are **immutable** in java. That is, once you create String objects, you can't modify them. Whenever you try to modify the existing String object, a new String object is created with modifications. Existing object is not at all altered.
- ❖ Where as **java.lang.StringBuffer** and **java.lang.StringBuilder** objects are mutable. That means, you can perform modifications to existing objects.

STRINGS

- ❖ Only **String** and **StringBuffer** objects are **thread safe**.
StringBuilder objects are not thread safe. So whenever you want immutable and thread safe string objects, use `java.lang.String` class and whenever you want mutable as well as thread safe string objects then use `java.lang.StringBuffer` class.
- ❖ In all three classes, **toString()** method is overridden. Whenever you use reference variables of these three types, they will return contents of the objects not physical address of the objects.
- ❖ **hashCode()** and **equals()** methods are overridden only in **java.lang.String** class but not in `java.lang.StringBuffer` and `java.lang.StringBuilder` classes.

STRINGS

- ❖ There is no **reverse()** and **delete()** methods in String class. But, StringBuffer and StringBuilder have reverse() and delete() methods.
- ❖ In case of String class, you can create the objects without new operator. But in case of StringBuffer and StringBuilder class, you have to use new operator to create the objects.

STRING CONSTRUCTOR

- ❖ If you want to create an empty string object, then use no-arg constructor of String class.

```
String s = new String();
```

- ❖ Below constructor takes character array as an argument.

```
char[] chars = {'J', 'A', 'V', 'A'};      //Character  
Array  
String s = new String(chars);
```

- ❖ Below constructor takes string as an argument.

```
String s = new String("JAVA");
```

STRING CONSTRUCTOR

- ❖ This constructor takes **StringBuffer** type as an argument.

```
StringBuffer strBuff = new StringBuffer("abc");
String s = new String(strBuff);
```

```
StringBuilder strBldr = new StringBuilder("abc");
String s = new String(strBldr);
```

STRING LITERALS

- ❖ In Java, all string literals like “java”, “abc”, “123” are treated as objects of `java.lang.String` class. That means, all methods of `String` class are also applicable to string literals.
- ❖ You can also create the objects of `String` class without using new operator. This can be done by assigning a string literal to reference variable of type `java.lang.String` class.

```
String s1 = "abc";
```

```
String s2 = "abc"+"def";
```

```
String s3 = "123"+"A"+"B";
```

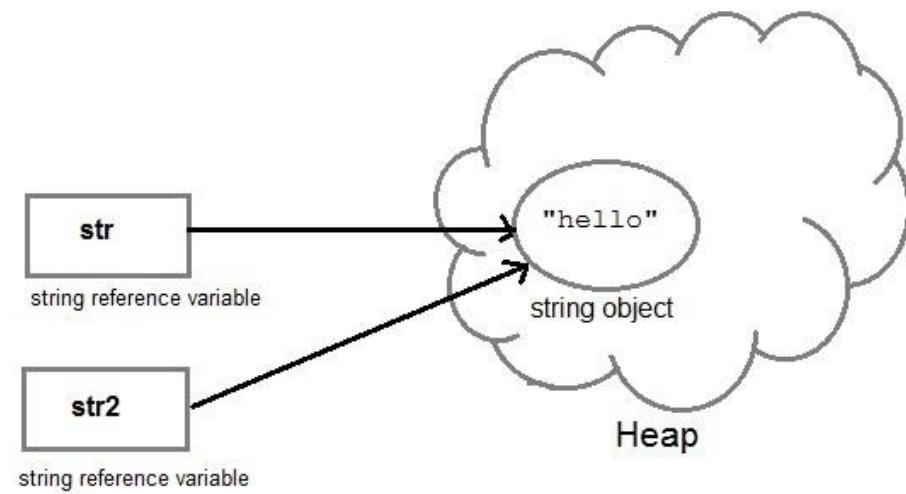
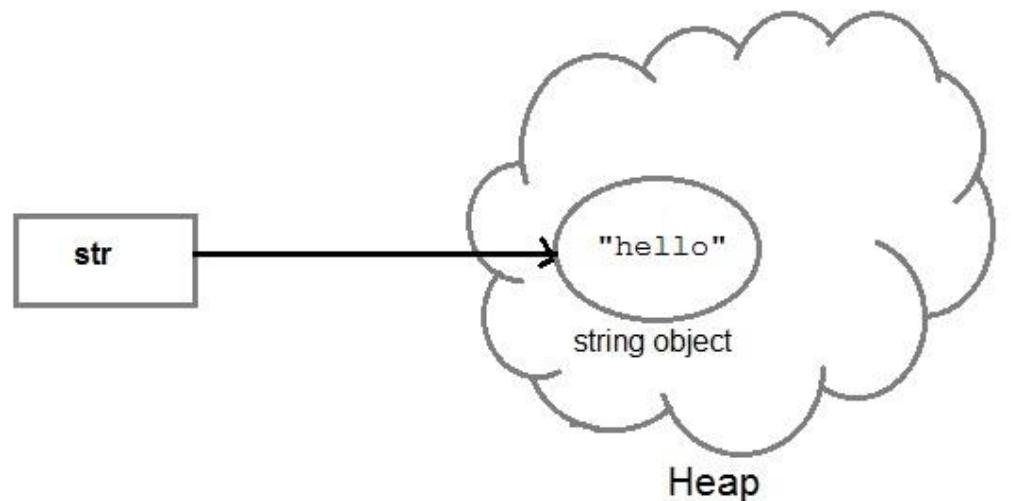
HOW STRINGS ARE STORED

- ❖ JVM divides the allocated memory to a Java program into two parts. one is **Stack** and another one is **heap**.
- ❖ Stack is used for execution purpose and heap is used for storage purpose.
- ❖ In that heap memory, JVM allocates some memory specially meant for string literals. This part of the heap memory is called **String Constant Pool**.
- ❖ Each time you create a String literal, the JVM checks the string pool first. If the string literal already exists in the pool, a reference to the pool instance is returned. If string does not exist in the pool, a new string object is created, and is placed in the pool.

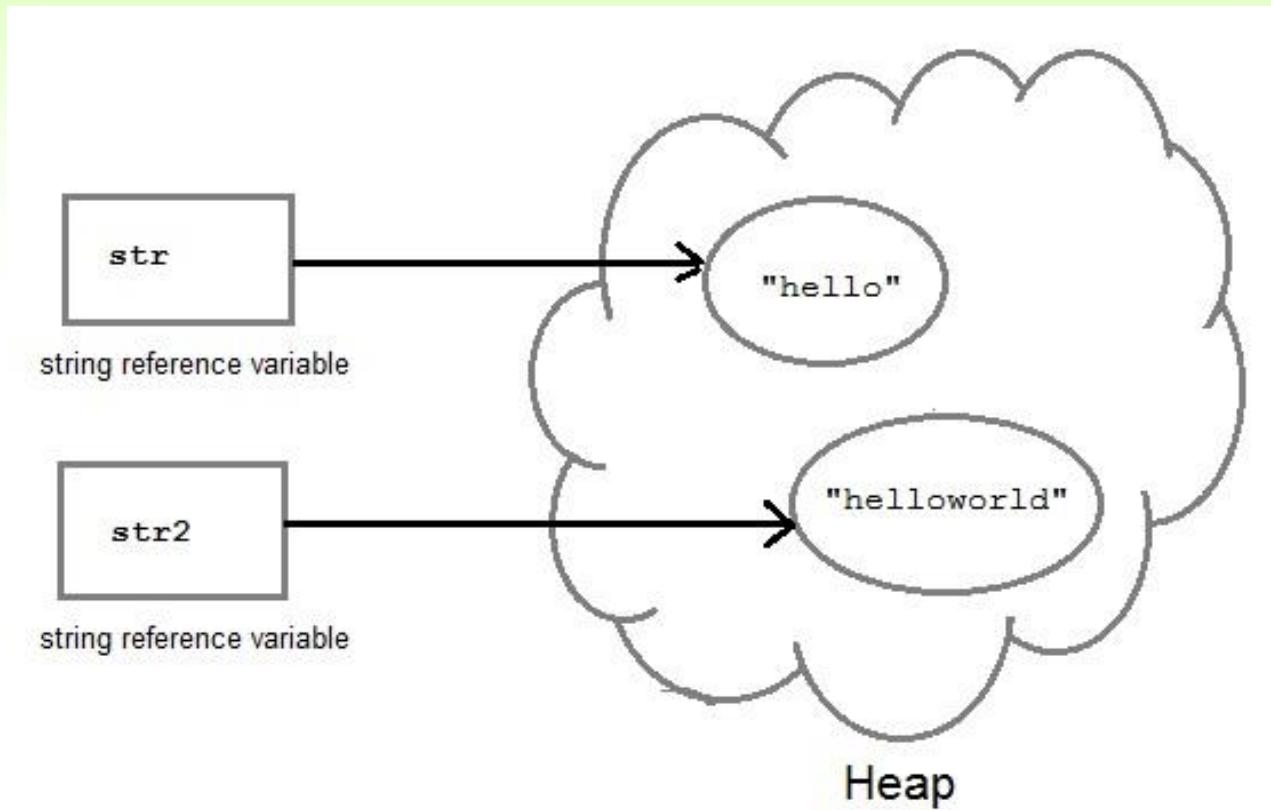
HOW STRINGS ARE STORED

- ❖ Whenever you create a string object using **string literal**, that object is stored in the **string constant pool** and whenever you create a string object using **new keyword**, such object is stored in the **heap memory**.
- ❖ pool space is allocated to an object depending upon it's content. There will be no two objects in the pool having the same content.
- ❖ But, when you create string objects using new keyword, a new object is created whether the content is same or not.

HOW STRINGS ARE STORED



HOW STRINGS ARE STORED



`==` VS. `EQUALS`

- ❖ “`==`” operator compares the two objects on their physical address. That means if two references are pointing to same object in the memory, then comparing those two references using “`==`” operator will return true.
- ❖ For example, if `s1` and `s2` are two references pointing to same object in the memory, then invoking `s1 == s2` will return true.
- ❖ This type of comparison is called “**Shallow Comparison**”.

== VS. EQUALS

- ❖ In **java.lang.String** class, **equals()** method is overridden to provide the comparison of two string objects based on their contents.
- ❖ That means, any two string objects having same content will be equal according to equals() method.
- ❖ For example, if s1 and s2 are two string objects having the same content, then invoking s1.equals(s2) will return true.

STRING CLASS FUNCTION

- ❖ **charAt()** – function returns the character located at the specified index.
- ❖ **equalsIgnoreCase()** – determines the equality of two Strings, ignoring thier case.
- ❖ **length()** – returns the number of characters in a String.
- ❖ **replace()** – replaces occurrences of character with a specified new character.
- ❖ **substring()** – returns a part of the string.
- ❖ **toLowerCase()**
- ❖ **toUpperCase()**

STRING CLASS FUNCTION

- ❖ **indexOf()** – returns the index of first occurrence of a substring or a character. **indexOf()** method has four forms:
- ❖ **int indexOf(String str):** It returns the index within this string of the first occurrence of the specified substring.
- ❖ **int indexOf(int ch, int fromIndex):** It returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
- ❖ **int indexOf(int ch):** It returns the index within this string of the first occurrence of the specified character.
- ❖ **int indexOf(String str, int fromIndex):** It returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

STRINGBUFFER

- ❖ **StringBuffer** class is used to create a mutable string object i.e its state can be changed after it is created.
- ❖ It represents growable and writable character sequence.
- ❖ As we know that String objects are immutable, so if we do a lot of changes with String objects, we will end up with a lot of memory leak.
- ❖ So **StringBuffer** class is used when we have to make lot of modifications to our string. It is also **thread safe** i.e multiple threads cannot access it simultaneously.

STRINGBUFFER

- ❖ StringBuffer defines 4 constructors. They are,
 - ❖ StringBuffer ()
 - ❖ StringBuffer (int size)
 - ❖ StringBuffer (String str)
 - ❖ StringBuffer (charSequence []ch)
-
- ❖ **StringBuffer()** creates an empty string buffer and reserves room for 16 characters.
 - ❖ **stringBuffer(int size)** creates an empty string and takes an integer argument to set capacity of the buffer.

STRINGBUFFER METHODS

- ❖ **append()** – This method will concatenate the string representation of any type of data to the end of the invoking StringBuffer object.

```
StringBuffer append(String str)
```

```
StringBuffer append(int n)
```

```
StringBuffer append(Object obj)
```

- ❖ **insert()** – This method inserts one string into another. Here are few forms of insert() method.

```
StringBuffer insert(int index, String str)
```

```
StringBuffer insert(int index, int num)
```

```
StringBuffer insert(int index, Object obj)
```

STRINGBUFFER METHODS

- ❖ **reverse()** – This method reverses the characters within a StringBuffer object.

```
StringBuffer str = new StringBuffer("Hello");
str.reverse();
System.out.println(str);
```

- ❖ **replace()** – This method replaces the string from specified start index to the end index.

```
StringBuffer str = new StringBuffer("Hello World");
str.replace( 6, 11, "java");
System.out.println(str);
```

LOOPS

- ❖ **While loop** – Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.

```
while(Boolean_expression) {  
    // Statements  
}
```

- ❖ **For loop** – is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times.

```
for(initialization; Boolean_expression; update) {  
    // Statements  
}
```

LOOPS

- ❖ **Do While loop** –is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

```
do {  
    // Statements  
}while(Boolean_expression);
```

LOOP CONTROL STATEMENTS

- ❖ **Break statement** – Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
- ❖ **Continue statement** – Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

SWITCH STATEMENT

- ❖ A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

```
switch(expression) {  
    case value :  
        // Statements  
        break; // optional  
  
    case value :  
        // Statements  
        break; // optional  
  
    // You can have any number of case statements.  
    default : // Optional  
        // Statements  
}
```

COLLECTIONS

- ❖ Collections are nothing but group of objects stored in well defined manner.
- ❖ Earlier, Arrays are used to represent these group of objects. But, arrays are not re-sizable. size of the arrays are fixed.
- ❖ Size of the arrays can not be changed once they are defined. This causes lots of problem while handling group of objects.
- ❖ To overcome this drawback of arrays, Collection framework or simply collections are introduced in java from JDK 1.2.

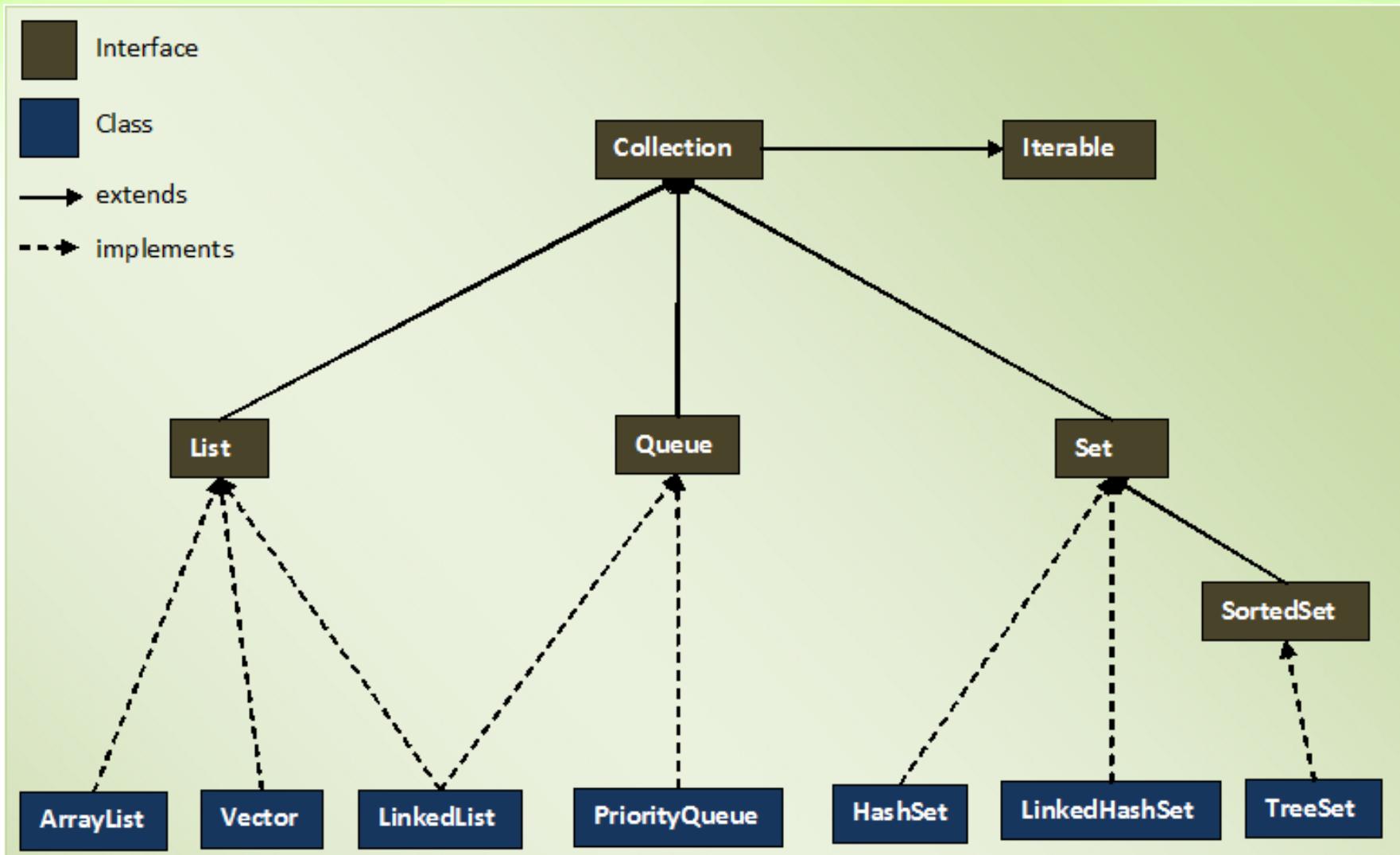
COLLECTION FRAMEWORK

- ❖ Collection Framework in java is a centralized and unified theme to store and manipulate the group of objects.
- ❖ Java Collection Framework provides some pre-defined classes and interfaces to handle the group of objects.
- ❖ Using collection framework, you can store the objects as a list or as a set or as a queue or as a map and perform operations like adding an object or removing an object or sorting the objects without much hard work.

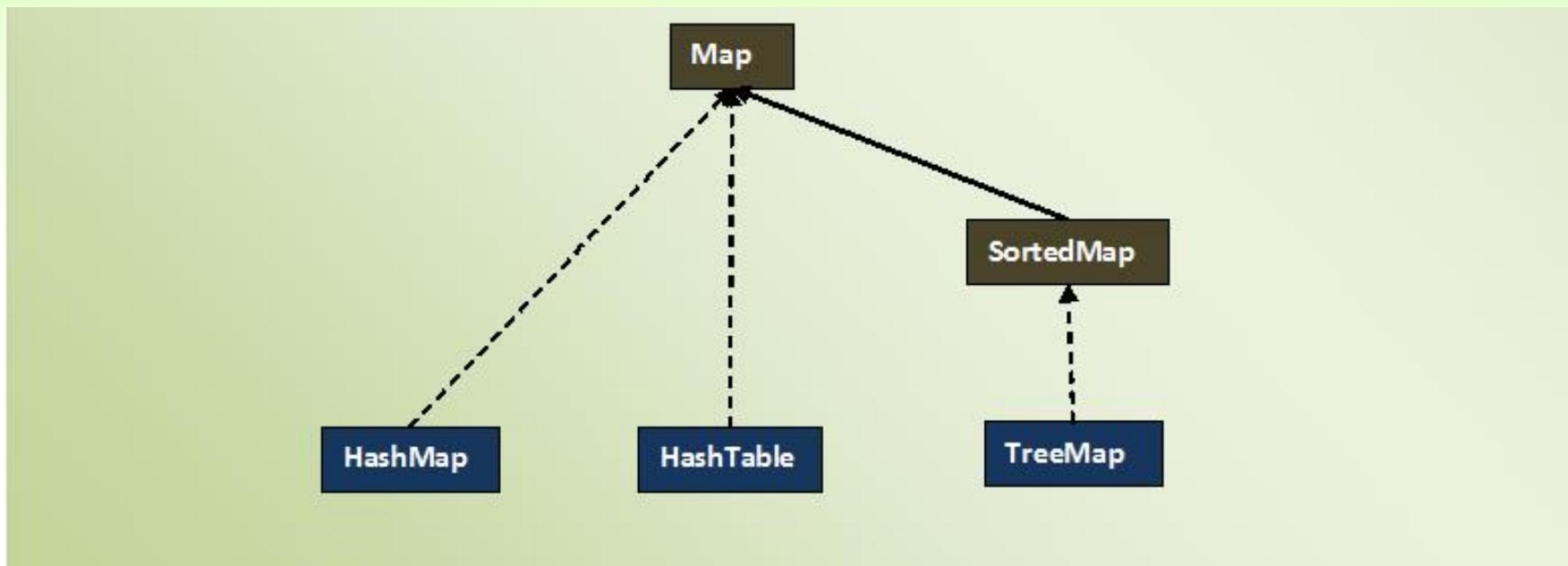
COLLECTION FRAMEWORK HIERARCHY

- ❖ All classes and interfaces related to Collection Framework are placed in **java.util** package. **java.util.Collection** class is at the top of class hierarchy of Collection Framework.

COLLECTION FRAMEWORK HIERARCHY



COLLECTION FRAMEWORK HIERARCHY



COLLECTION FRAMEWORK HIERARCHY

- ❖ The entire collection framework is divided into four interfaces.
- ❖ 1) **List** —> It handles sequential list of objects. **ArrayList**, **Vector** and **LinkedList** classes implement this interface.
- ❖ 2) **Queue** —> It handles special list of objects in which elements are removed only from the head. **LinkedList** and **PriorityQueue** classes implement this interface.
- ❖ 3) **Set** —> It handles list of objects which must contain unique element. This interface is implemented by **HashSet** and **LinkedHashSet** classes and extended by **SortedSet** interface which in turn, is implemented by **TreeSet**.

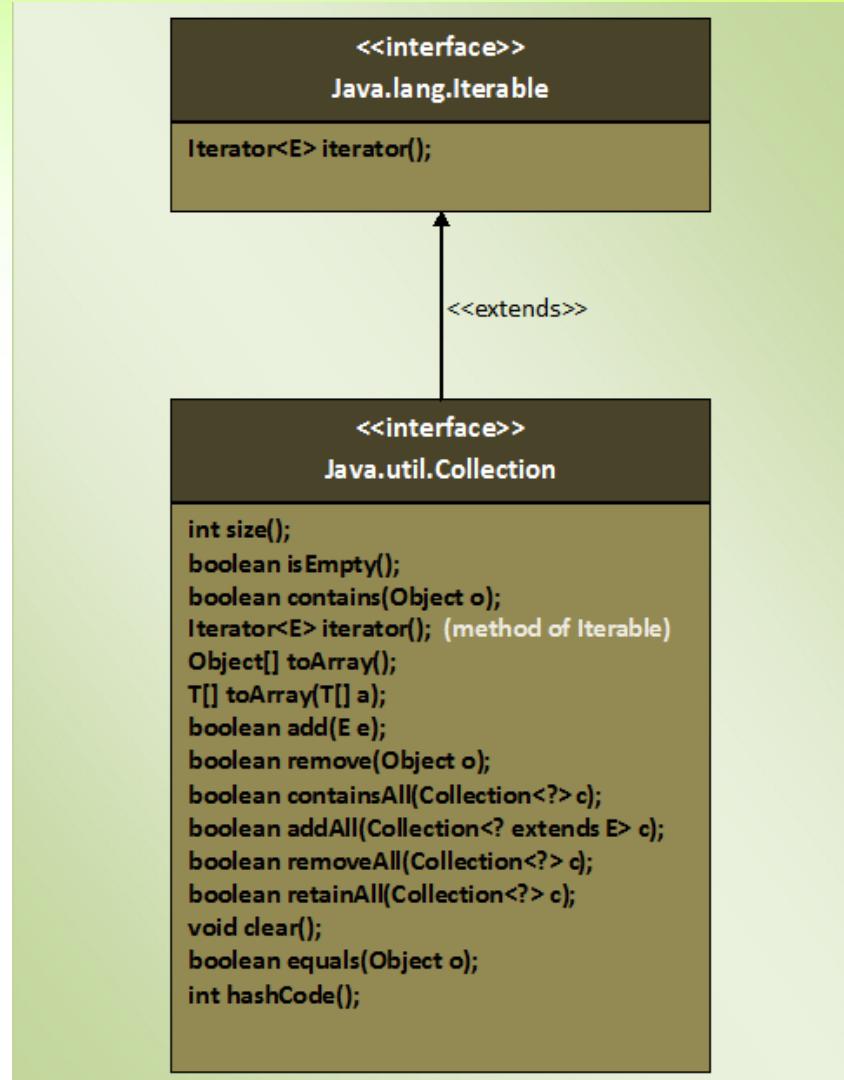
COLLECTION FRAMEWORK HIERARCHY

- ❖ 4) **Map** —> This is the one interface in Collection Framework which is not inherited from Collection interface. It handles group of objects as Key/Value pairs. It is implemented by **HashMap** and **HashTable** classes and extended by **SortedMap** interface which in turn is implemented by **TreeMap**.
- ❖ Three of above interfaces (List, Queue and Set) inherit from Collection interface. Although, Map is included in collection framework it does not inherit from Collection interface.

COLLECTION INTERFACE

- ❖ Collection interface is the root level interface in the collection framework. List, Queue and Set are all sub interfaces of Collection interface.
- ❖ Collection interface extends **Iterable** interface which is a member of **java.lang** package. Iterable interface has only one method called **iterator()**. It returns an Iterator object, using that object you can iterate over the elements of Collection.

COLLECTION INTERFACE



COLLECTION INTERFACE METHODS

boolean add(E obj)	Used to add objects to a collection. Returns true if obj was added to the collection. Returns false if obj is already a member of the collection, or if the collection does not allow duplicates.
boolean addAll(Collection C)	Add all elements of collection C to the invoking collection. Returns true if the element were added. Otherwise, returns false.
boolean remove(Object obj)	To remove an object from collection. Returns true if the element was removed. Otherwise, returns false.
boolean removeAll(Collection C)	Removes all element of collection C from the invoking collection. Returns true if the collection's elements were removed. Otherwise, returns false.

COLLECTION INTERFACE METHODS

boolean contains(Object obj)	To determine whether an object is present in collection or not. Returns true if obj is an element of the invoking collection. Otherwise, returns false.
boolean isEmpty()	Returns true if collection is empty, else returns false.
int size()	Returns number of elements present in collection.
void clear()	Removes total number of elements from the collection.

COLLECTION INTERFACE METHODS

boolean retainAll(Collection c)	Deletes all the elements of invoking collection except the specified collection.
Iterator iterator()	Returns an iterator for the invoking collection.
boolean equals(Object obj)	Returns true if the invoking collection and obj are equal. Otherwise, returns false.
Object[] toArray(Object array[])	Returns an array containing only those collection elements whose type matches of the specified array.
boolean equals(Object o)	Compares the specified object with this collection for equality.
int hashCode()	Returns the hash code value of this collection.

COLLECTION INTERFACE

- ❖ equals() and hashCode() methods in the Collection interface are not the methods of java.lang.Object class.
- ❖ Because, interfaces does not inherit from Object class. Only classes in java are sub classes of Object class.
- ❖ Any classes implementing Collection interface must provide their own version of equals() and hashCode() methods or they can retain default version inherited from Object class.

WHY COLLECTIONS WERE MADE GENERIC ?

- ❖ Generics added type safety to Collection framework.
- ❖ Earlier collections stored Object class references which meant any collection could store any type of object.
- ❖ Hence there were chances of storing incompatible types in a collection, which could result in run time mismatch.
- ❖ Hence Generics was introduced through which you can explicitly state the type of object being stored.

COLLECTIONS AND AUTOBOXING

- ❖ Autoboxing converts primitive types into Wrapper class Objects.
- ❖ As collections doesn't store primitive data types(stores only references), hence Autoboxing facilitates the storing of primitive data types in collection by boxing it into its wrapper type.

EXCEPTIONS IN COLLECTIONS

Exception Name	Description
UnSupportedOperationException	occurs if a Collection cannot be modified
ClassCastException	occurs when one object is incompatible with another
NullPointerException	occurs when you try to store null object in Collection
IllegalArgumentException	thrown if an invalid argument is used
IllegalStateException	thrown if you try to add an element to an already full Collection

LIST INTERFACE

- ❖ List Interface represents an ordered or sequential collection of objects.
- ❖ This interface has some methods which can be used to store and manipulate the ordered collection of objects.
- ❖ The classes which implement the List interface are called as Lists.
ArrayList, **Vector** and **LinkedList** are some examples of lists
- ❖ You have the control over where to insert an element and from where to remove an element in the list.

LIST INTERFACE

- ❖ Elements of the lists are ordered using Zero based index.
- ❖ You can access the elements of lists using an integer index.
- ❖ Elements can be inserted at a specific position using integer index. Any pre-existing elements at or beyond that position are shifted right.
- ❖ Elements can be removed from a specific position. The elements beyond that position are shifted left.
- ❖ A list may contain duplicate elements.
- ❖ A list may contain multiple null elements.

LIST INTERFACE

E get(int index)	Returns element at the specified position.
E set(int index, E element)	Replaces an element at the specified position with the passed element.
void add(int index, E element)	Inserts passed element at a specified index.
E remove(int index)	Removes an element at specified index.
int indexOf(Object o)	It returns an index of first occurrence of passed object.
int lastIndexOf(Object o)	It returns an index of last occurrence of passed object.

LIST INTERFACE

ListIterator<E> listIterator()	It returns a list iterator over the elements of this list.
ListIterator<E> listIterator(int index)	Returns a list iterator over the elements of this list starting from the specified index.
List<E> subList(int fromIndex, int toIndex)	Returns sub list of this list starting from 'fromIndex' to 'toIndex'.

ADVANTAGES OF USING ARRAYLIST OVER ARRAYS

- ❖ Drawbacks of arrays are:
- ❖ Arrays are of fixed length. You can not change the size of the arrays once they are created.
- ❖ You can not accommodate an extra element in an array after they are created.
- ❖ Memory is allocated to an array during it's creation only, much before the actual elements are added to it.

ARRAYLIST CLASS

- ❖ ArrayList, in simple terms, can be defined as re-sizable array.
- ❖ ArrayList is same like normal array but it can grow and shrink dynamically to hold any number of elements.
- ❖ ArrayList is a sequential collection of objects which increases or decreases in size as we add or delete the elements.
- ❖ Default initial capacity of an ArrayList is 10. This capacity increases automatically as we add more elements to arraylist.
- ❖ You can also specify initial capacity of an ArrayList while creating it.
- ❖ ArrayList class implements **List** interface and extends **AbstractList**. It also implements 3 marker interfaces – **RandomAccess**, **Cloneable** and **Serializable**.

MODIFICATION OPERATIONS

- ❖ **boolean add(E e)** : This method appends an element at the end of this List. If the ArrayList is empty then there will be exactly one element after this operation.
- ❖ **void add(int index, E element)** : This method inserts an element at the specified position.
- ❖ **boolean remove(Object o)** : It removes first occurrence of specified element from the list.
- ❖ **E remove(int index)** : This method removes an element from the specified position.
- ❖ **E set(int index, E element)** : This method replaces an element at the specified position with the passed element.

BULK MODIFICATION OPERATIONS

- ❖ **boolean addAll(Collection c)** : This method appends all elements of the passed collection at the end of this list.
- ❖ **boolean addAll(int index, Collection c)** : This method inserts all elements of the passed collection at the specified position in this list.
- ❖ **boolean removeAll(Collection c)** : This method removes all elements of this list which are also elements of the passed collection.
- ❖ **boolean retainAll(Collection c)** : This method retains only those elements in this list which are also elements of the passed collection.
- ❖ **void clear()** : This method removes all elements of the list.

ITERATOR VS. LISTITERATOR

- ❖ **Iterator** and **ListIterator** are two interfaces in Java collection framework which are used to traverse the collections.
- ❖ Although ListIterator extends Iterator, there are some differences in the way they traverse the collections.
- ❖ Using Iterator, you can traverse List, Set and Queue type of objects. But using ListIterator, you can traverse only List objects. In Set and Queue types, there is no method to get the ListIterator object. But, In List types, there is a method called `listIterator()` which returns ListIterator object.

ITERATOR VS. LISTITERATOR

- ❖ Using Iterator, we can traverse the elements only in forward direction. But, using ListIterator you can traverse the elements in both the directions – forward and backward. ListIterator has those methods to support the traversing of elements in both the directions.

ITERATOR METHODS

- ❖ **boolean hasNext()** → Checks whether collection has more elements.
- ❖ **E next()** → Returns the next element in the collection.
- ❖ **void remove()** → Removes the current element in the collection i.e element returned by next().

LIST ITERATOR METHODS

- ❖ boolean hasNext() -> Checks whether the list has more elements when traversing the list in forward direction.
- ❖ boolean hasPrevious() -> Checks whether list has more elements when traversing the list in backward direction.
- ❖ E next() -> Returns the next element in the list and moves the cursor forward.
- ❖ E previous() -> Returns the previous element in the list and moves the cursor backward.
- ❖ int nextIndex() -> Returns index of the next element in the list.
- ❖ int previousIndex() -> Returns index of the previous element in the list.

LIST ITERATOR METHODS

- ❖ void remove() -> Removes the current element in the collection i.e element returned by next() or previous().
- ❖ void set(E e) -> Replaces the current element i.e element returned by next() or previous() with the specified element.
- ❖ void add(E e) -> Inserts the specified element in the list.

ITERATING AN ARRAYLIST IN JAVA

- ❖ Iteration Using Normal for loop.
- ❖ Iteration Using Iterator Object.
- ❖ Iteration Using ListIterator Object.
- ❖ Iteration Using Enhanced for loop.

LINKEDLIST CLASS

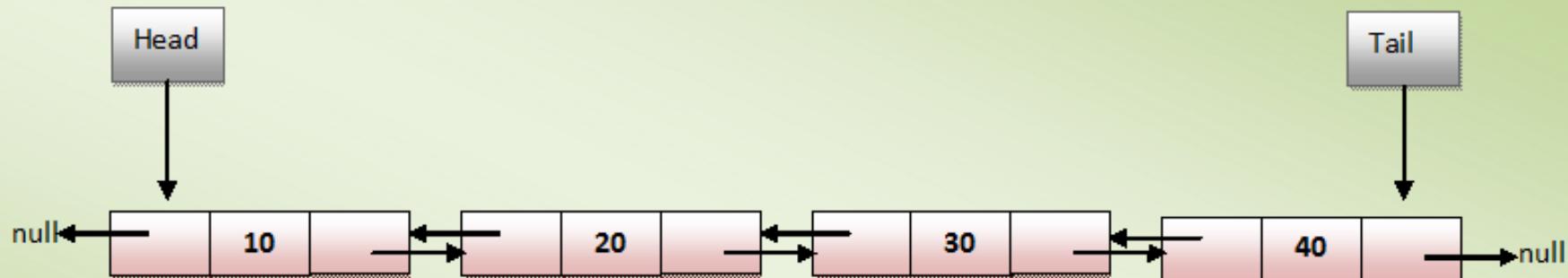
- ❖ LinkedList is a data structure where each element consist of three things.
- ❖ First one is the reference to previous element, second one is the actual value of the element and last one is the reference to next element.
- ❖ The LinkedList class in Java is an implementation of doubly linked list which can be used both as a List as well as Queue.
- ❖ The LinkedList can have any type of elements including null and duplicates.
- ❖ Elements can be inserted and can be removed from both the ends and can be retrieved from any arbitrary position.

LINKEDLIST CLASS

- ❖ LinkedList has two constructors.
- ❖ LinkedList() - It creates an empty LinkedList
- ❖ LinkedList(Collection C) - It creates a LinkedList that is initialized with elements of the Collection c

LINKEDLIST CLASS - PROPERTIES

- ❖ Elements in the LinkedList are called as **Nodes**. Where each node consist of three parts – Reference To Previous Element, Value Of The Element and Reference To Next Element.



Insertions and Removals in LinkedList are faster than ArrayList. Because, You don't need to resize LinkedList after each insertions and removals.

Retrieval operations in LinkedList are slow compared to ArrayList. Because, it needs traversing from the beginning or from end to reach the element.

LINKEDLIST CLASS - METHODS

❖ You can insert the elements at both the ends and also in the middle of the LinkedList. Below is the list of methods for insertion operations.

❖ Insertion At Head
❖ addFirst(E e)
❖ offerFirst(E e)

Insertion In The Middle
add(int index, E e)
addAll(int index, Collection c)

Insertion At Tail
add(E e)
addAll(Collection c)
offer(E e)
offerLast(E e)

LINKEDLIST CLASS - METHODS

❖ You can remove the elements from the head, from the tail and also from the middle of the LinkedList.

❖ Removing from head
❖ poll()
❖ pollFirst()
❖ remove()
❖ removeFirst()

Removing from middle
Remove(int index)

Removing from tail
pollLast()
removeLast()

LINKEDLIST CLASS - METHODS

- ❖ You can retrieve the elements from the head, from the middle and from the tail of the LinkedList.

- ❖ Retrieving from head
- ❖ element()
- ❖ getFirst()
- ❖ peek()
- ❖ peekFirst()

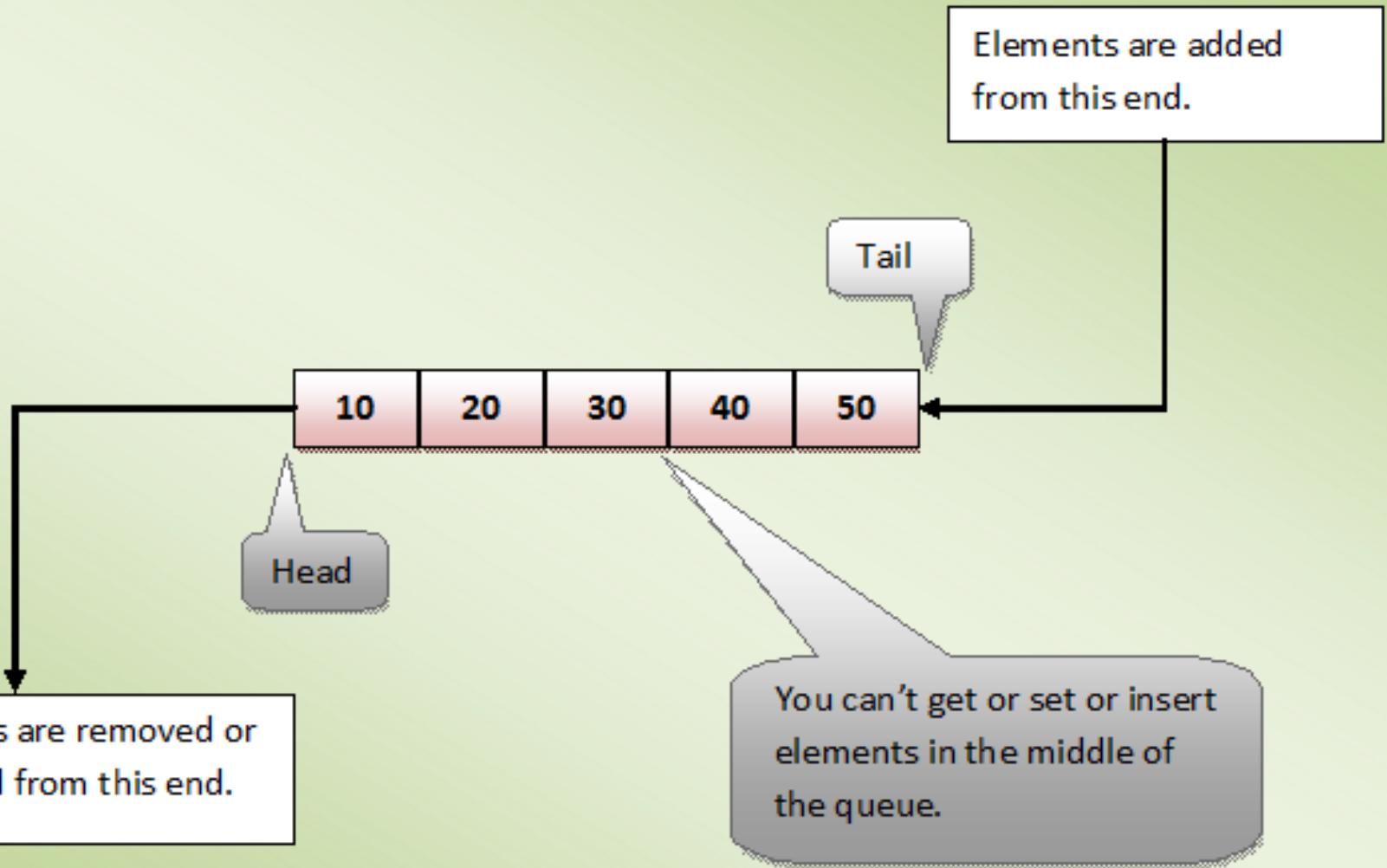
Retrieving from middle
get(int index)

Retrieving from tail
getLast()
peekLast()

QUEUE INTERFACE

- ❖ Queue interface defines queue data structure which is normally **First-In-First-Out**.
- ❖ Queue is a data structure in which elements are added from one end and elements are deleted from another end.
- ❖ But, exception being the **Priority Queue** in which elements are removed from one end, but elements are added according to the order defined by the supplied comparator.

QUEUE INTERFACE



QUEUE INTERFACE - PROPERTIES

- ❖ Null elements are not allowed in the queue. If you try to insert null object into the queue, it throws NullPointerException.
- ❖ Queue can have duplicate elements.
- ❖ Unlike a normal list, queue is not random access. i.e you can't set or insert or get elements at an arbitrary positions.
- ❖ In most of cases, elements are inserted at one end called tail of the queue and elements are removed or retrieved from another end called head of the queue.

QUEUE INTERFACE - PROPERTIES

- ❖ In the Queue Interface, there are two methods to obtain and remove the elements from the head of the queue. They are poll() and remove(). The difference between them is, poll() returns null if the queue is empty and remove() throws an exception if the queue is empty.
- ❖ There are two methods in the Queue interface to obtain the elements but don't remove. They are peek() and element(). peek() returns null if the queue is empty and element() throws an exception if the queue is empty.

SET INTERFACE

- ❖ The set is a linear collection of objects with **no duplicates**.
- ❖ Duplicate elements are not allowed in a set.
- ❖ The Set interface extends Collection interface. Set interface does not have it's own methods. All it's methods are inherited from Collection interface.
- ❖ The only change that has been made to Set interface is that add() method will return false if you try to insert an element which is already present in the set.

PROPERTIES OF SET

- ❖ Set contains only **unique elements**. It does not allow duplicates.
- ❖ Set can contain only one null element.
- ❖ Random access of elements is not possible.
- ❖ Order of elements in a set is implementation dependent. **HashSet** elements are ordered on hash code of elements. **TreeSet** elements are ordered according to supplied Comparator (If no Comparator is supplied, elements will be placed in ascending order) and **LinkedHashSet** maintains insertion order.
- ❖ Set interface contains only methods inherited from Collection interface. It does not have it's own methods. But, applies restriction on methods so that duplicate elements are always avoided.

HASHSET CLASS

- ❖ The HashSet class in Java is an implementation of Set interface. HashSet is a collection of objects which contains only unique elements.
- ❖ Duplicates are not allowed in HashSet.
- ❖ HashSet gives constant time performance for insertion, removal and retrieval operations. It allows only one null element.
- ❖ The HashSet internally uses **HashMap** to store the objects. The elements you insert in HashSet will be stored as keys of that HashMap object and their values will be a constant called **PRESENT**.

HASHSET CLASS - PROPERTIES

- ❖ HashSet does not allow duplicate elements. If you try to insert a duplicate element, older element will be overwritten.
- ❖ HashSet doesn't maintain any order. The order of the elements will be largely unpredictable. And it also doesn't guarantee that order will remain constant over time.

LINKEDHASHSET CLASS

- ❖ The LinkedHashSet in java is an ordered version of HashSet which internally maintains one **doubly linked** list running through it's elements.
- ❖ This doubly linked list is responsible for maintaining the insertion order of the elements.
- ❖ Unlike HashSet which maintains no order, LinkedHashSet maintains insertion order of elements. i.e elements are placed in the order they are inserted.
- ❖ LinkedHashSet is recommended over HashSet if you want a unique collection of objects in an insertion order.

SORTEDSET INTERFACE

- ❖ The SortedSet interface extends Set interface.
- ❖ SortedSet is a set in which elements are placed according to supplied comparator.
- ❖ This Comparator is supplied while creating a SortedSet. If you don't supply comparator, elements will be placed in ascending order.

METHODS OF SORTEDSET INTERFACE

Comparator<? super E> comparator()	Returns Comparator used to order the elements. If no comparator is supplied, it returns null.
SortedSet<E> subSet(E fromElement, E toElement)	Returns a portion of this set whose elements range from 'fromElement' (Inclusive) and 'toElement' (Exclusive).
SortedSet<E> headSet(E toElement)	Returns a SortedSet whose elements are in the range from first element of the set (Inclusive) to 'toElement' (exclusive).
SortedSet<E> tailSet(E fromElement)	Returns a SortedSet whose elements are in the range from 'fromElement' (Inclusive) to last element of the set (exclusive).
E first()	Returns first element of the SortedSet.
E last()	Returns last element of the SortedSet.

SORTEDSET INTERFACE - PROPERTIES

- ❖ SortedSet **can not have null elements**. If you try to insert null element, it gives NullPointerException at run time.
- ❖ As SortedSet is a set, duplicate elements are not allowed.
- ❖ Inserted elements must be of **Comparable** type and they must be mutually Comparable.
- ❖ You can retrieve first element and last elements of the SortedSet. You can't access SortedSet elements randomly. i.e Random access is denied.
- ❖ SortedSets returned by headSet(), tailSet() and subSet() methods are just views of the original set. So, changes in the returned set are reflected in the original set and vice versa.

TREESET CLASS

- ❖ Elements in TreeSet are sorted according to supplied **Comparator**.
- ❖ You need to supply this Comparator while creating a TreeSet itself. If you don't pass any Comparator while creating a TreeSet, elements will be placed in their natural ascending order.
- ❖ Elements inserted in the TreeSet must be of Comparable type and elements must be mutually comparable. If the elements are not mutually comparable, you will get ClassCastException at run time.
- ❖ TreeSet does not allow even a single null element.
- ❖ Iterator returned by TreeSet is of fail-fast nature. That means, If TreeSet is modified after the creation of Iterator object, you will get ConcurrentModificationException.

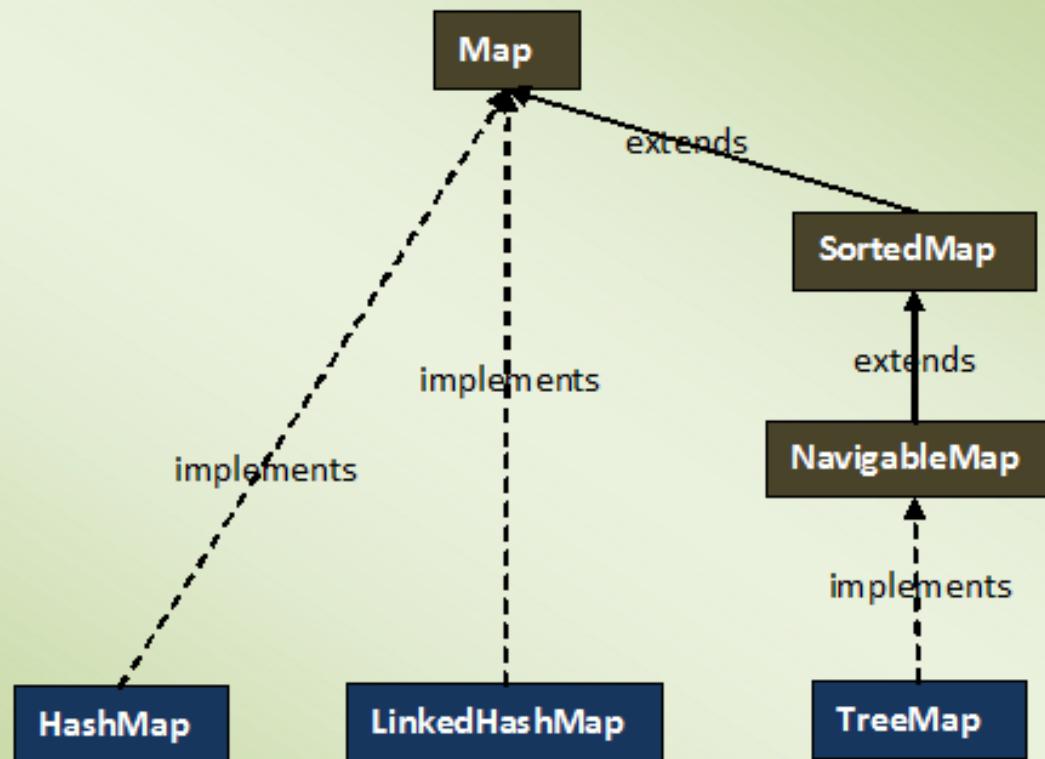
TREESET CLASS

- ❖ Constructors:
- ❖ **TreeSet()** - It is used to construct an empty tree set that will be sorted in an ascending order according to the natural order of the tree set.
- ❖ **TreeSet(Collection c)** - It is used to build a new tree set that contains the elements of the collection c.
- ❖ **TreeSet(Comparator comp)** - It is used to construct an empty tree set that will be sorted according to given comparator.
- ❖ **TreeSet(SortedSet ss)** - It is used to build a TreeSet that contains the elements of the given SortedSet.

MAP INTERFACE

- ❖ The **Map** interface in java is one of the four top level interfaces of Java Collection Framework along with List, Set and Queue interfaces.
- ❖ But, unlike others, it doesn't inherit from Collection interface. Instead it starts its own interface hierarchy for maintaining the key-value associations.
- ❖ Map is an object of **key-value** pairs where each key is associated with a value.

MAP INTERFACE



MAP INTERFACE

- ❖ Map interface is a part of Java Collection Framework, but it doesn't inherit Collection Interface.
- ❖ 2) Map interface stores the data as a key-value pairs where each key is associated with a value.
- ❖ 3) A map can not have duplicate keys but can have duplicate values.
- ❖ 4) Each key at most must be associated with one value.
- ❖ 5) Each key-value pairs of the map are stored as Map.Entry objects. Map.Entry is an inner interface of Map interface.
- ❖ 6) The common implementations of Map interface are **HashMap**, **LinkedHashMap** and **TreeMap**.

MAP INTERFACE

- ❖ 7) Order of elements in map is implementation dependent. HashMap doesn't maintain any order of elements. LinkedHashMap maintains insertion order of elements. Where as TreeMap places the elements according to supplied Comparator.
- ❖ 8) The Map interface provides three methods, which allows map's contents to be viewed as a **set of keys (keySet() method)**, **collection of values (values() method)**, or **set of key-value mappings (entrySet() method)**.

HASHMAP CLASS - CONSTRUCTOR

- ❖ **HashMap()** - It is used to construct a default HashMap.
- ❖ **HashMap(Map m)** - It is used to initializes the hash map by using the elements of the given Map object m.
- ❖ **HashMap(int capacity)** - It is used to initializes the capacity of the hash map to the given integer value, capacity.
- ❖ **HashMap(int capacity, float fillRatio)** - It is used to initialize both the capacity and fill ratio of the hash map by using its arguments.

HASHMAP CLASS- METHODS

- ❖ **public V put(K key, V value)** - This method inserts specified key-value mapping in the map. If map already has a mapping for the specified key, then it rewrites that value with new value.
- ❖ 2) **public void putAll(Map m)** - This method copies all of the mappings of the map m to this map.
- ❖ 3) **public V get(Object key)** - This method returns the value associated with a specified key.
- ❖ 4) **public int size()** - This method returns the number of key-value pairs in this map.
- ❖ 5) **public boolean isEmpty()** - This method checks whether this map is empty or not.

HASHMAP CLASS- METHODS

- ❖ 6) **public boolean containsKey(Object key)** - This method checks whether this map contains the mapping for the specified key.
- ❖ 7) **public boolean containsValue(Object value)** - This method checks whether this map has one or more keys mapping to the specified value.
- ❖ 8) **public V remove(Object key)** - This method removes the mapping for the specified key.
- ❖ 9) **public void clear()** - This method removes all the mappings from this map.
- ❖ 10) **public Set<K> keySet()** - This method returns the Set view of the keys in the map.

HASHMAP CLASS- METHODS

- ❖ 11) **public Collection<V> values()** - This method returns Collection view of the values in the map.
- ❖ 12) **public Set<Map.Entry<K, V>> entrySet()** - This method returns the Set view of all the mappings in this map.
- ❖ 13) **public V putIfAbsent(K key, V value)** - This method maps the given value with specified key if this key is currently not associated with a value or mapped to a null.
- ❖ 13) **public boolean remove(Object key, Object value)** - This method removes the entry for the specified key if this key is currently mapped to a specified value.

HASHMAP CLASS- METHODS

- ❖ 14) **public boolean replace(K key, V oldValue, V newValue)** - This method replaces the oldValue of the specified key with newValue if the key is currently mapped to oldValue.
- ❖ 15) **public V replace(K key, V value)** - This method replaces the current value of the specified key with new value.

HASHMAP VS. HASHTABLE

- ❖ 1) **Thread Safe:**
- ❖ HashTable is internally synchronized. Therefore, it is very much safe to use HashTable in multi threaded applications. Whereas HashMap is not internally synchronized. Therefore, it is not safe to use HashMap in multi threaded applications without external synchronization. You can externally synchronize HashMap using Collections.synchronizedMap() method.
- ❖ 2) **Inherited From**
- ❖ Though both HashMap and HashTable implement Map interface, but they extend two different classes. HashMap extends AbstractMap class whereas HashTable extends Dictionary class which is the legacy class in java.
- ❖ ashTable.

HASHMAP VS. HASHTABLE

- ❖ **3) Null Keys And Null Values:**
- ❖ HashMap allows maximum one null key and any number of null values. Whereas HashTable doesn't allow even a single null key and null value.

- ❖ **4) Traversal:**
- ❖ HashMap returns only Iterators which are used to traverse over the elements of HashMap. HashTable returns Iterator as well as Enumeration which can be used to traverse over the elements of HashTable.

HASHMAP VS. HASHTABLE

- ❖ 5) **Fail-Fast Vs Fail-Safe**
- ❖ Iterator returned by HashMap are **fail-fast** in nature i.e they throw **ConcurrentModificationException** if the HashMap is modified after the creation of Iterator other than iterator's own remove() method. On the other hand, Enumeration returned by the HashTable are fail-safe in nature i.e they don't throw any exceptions if the HashTable is modified after the creation of Enumeration.
- ❖ 6) **Performance**
- ❖ As HashTable is internally synchronized, this makes HashTable slightly slower than the HashMap.
- ❖ 7) **Legacy Class**
- ❖ HashTable is a legacy class. It is almost considered as due for deprecation. Since JDK 1.5, **ConcurrentHashMap** is considered as better option than the HashTable.

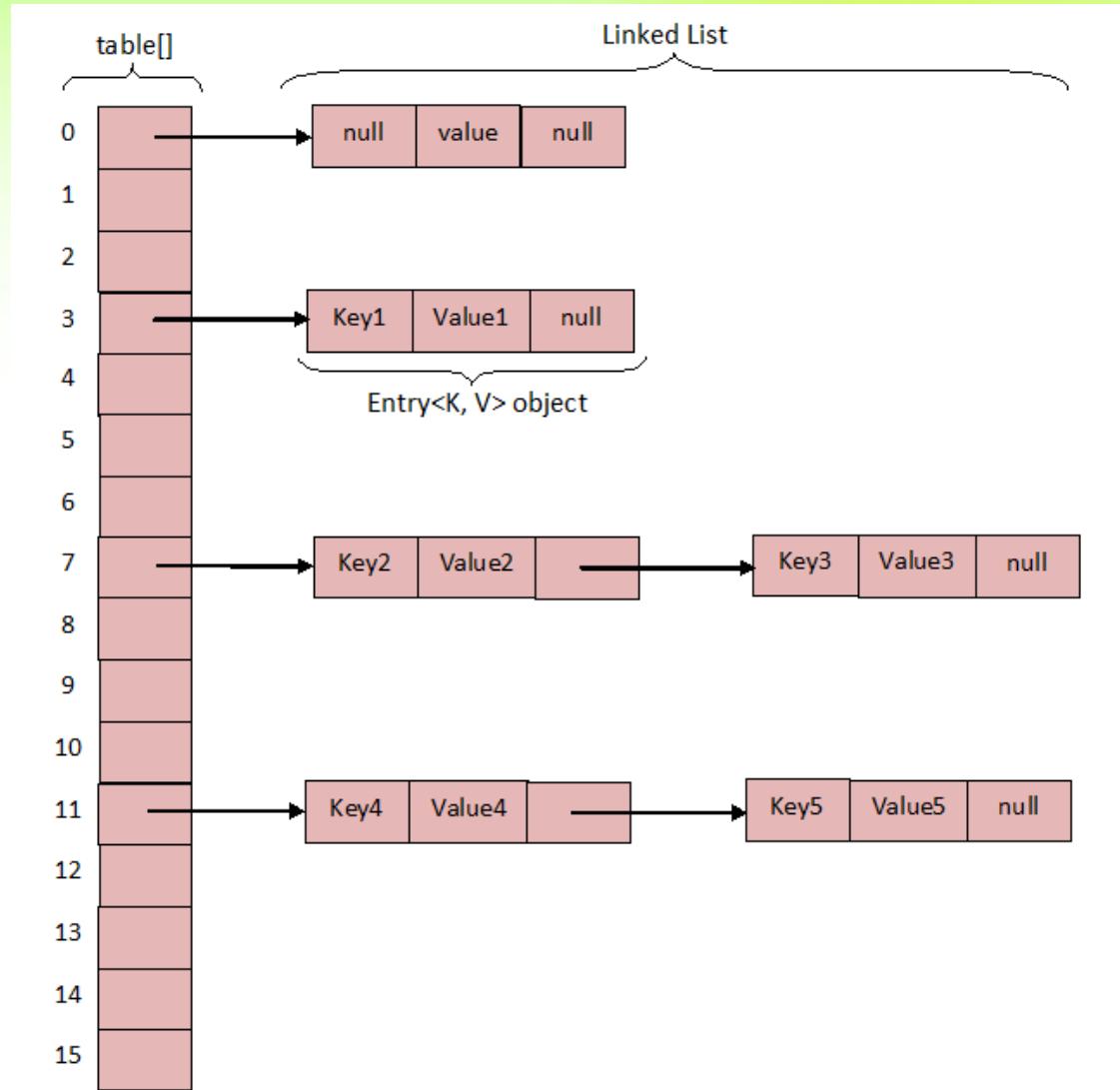
HASHCODE() AND EQUALS()

- ❖ An object of a class that override the equals() method can be used as elements in a collection.
- ❖ And if they override the hashCode() method also, they can be used as elements as keys in a HashMap.

HASHCODE() AND EQUALS()

- ❖ If two objects are equal according to the **equals(Object)** method, then calling the **hashCode** method on each of the two objects must produce the same integer result.
- ❖ It is not required that if two objects are unequal according to the **equals(java.lang.Object)** method, then calling the **hashCode** method on each of the two objects must produce distinct integer results.

HASHMAP INTERNAL STRUCTURE



COLLECTIONS UTILITY CLASS

- ❖ Collections is an utility class in java.util package. It consists of only static methods which are used to operate on objects of type Collection.
- ❖ For example, it has the method to find the maximum element in a collection, it has the method to sort the collection, it has the method to search for a particular element in a collection.

COLLECTIONS

- ❖ What is Collection's Hierarchy?
- ❖ What is the difference between Collection and Collections?
- ❖ Why Map interface does not extend Collection interface?
- ❖ What is the difference between comparable and comparator?
- ❖ Which methods you need to override to use any object as key in HashMap ?
- ❖ What is the importance of hashCode() and equals() methods?
- ❖ How can we sort a list of Objects?
- ❖ What is the difference between Fail- fast iterator and Fail-safe iterator

COLLECTIONS

- ❖ There are two objects a and b with same hashCode. I am inserting these two objects inside a hashmap.
- ❖ `hMap.put(a,a);`
- ❖ `hMap.put(b,b);`
- ❖ where `a.hashCode() == b.hashCode()`
- ❖ Now tell me how many objects will be there inside the hashmap?

FILE HANDLING IN JAVA

- ❖ Java I/O (Input and Output) is used to process the input and produce the output.
- ❖ Java uses the concept of stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.
- ❖ We can perform file handling in java by Java I/O API.

STREAM

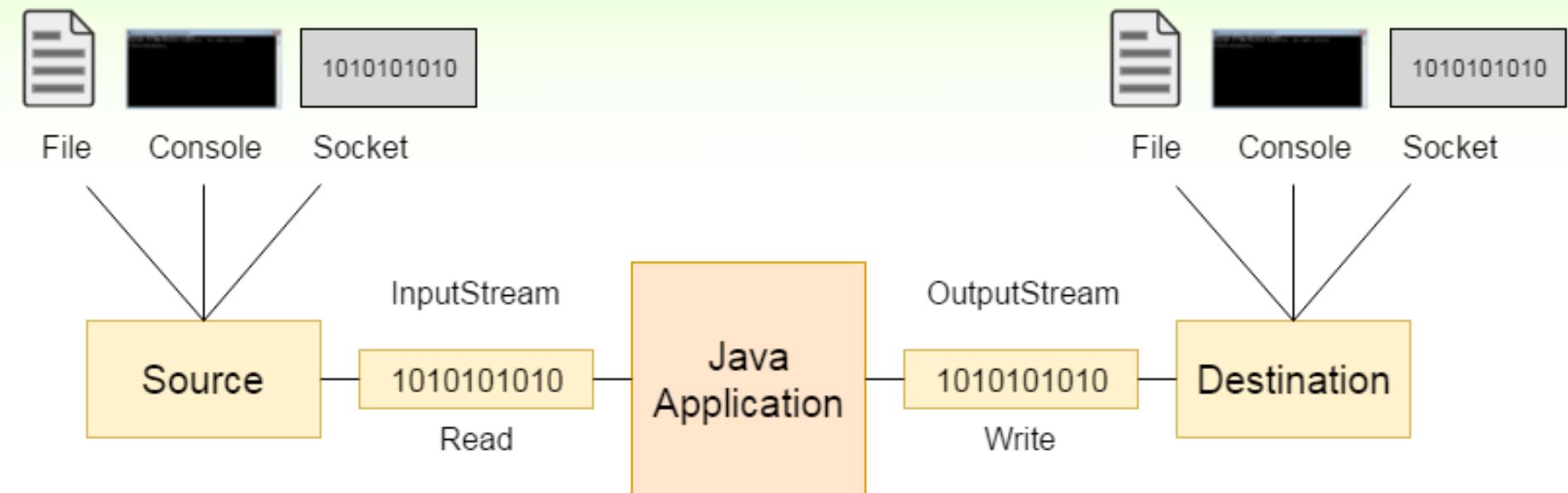
- ❖ A stream is a sequence of data. In Java a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.
- ❖ In java, 3 streams are created for us automatically. All these streams are attached with console.
 - ❖ 1) **System.out**: standard output stream
 - ❖ 2) **System.in**: standard input stream
 - ❖ 3) **System.err**: standard error stream

OUTPUTSTREAM/INPUTSTREAM

- ❖ OutputStream
- ❖ Java application uses an output stream to write data to a destination, it may be a file, an array, peripheral device or socket.

- ❖ InputStream
- ❖ Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket.

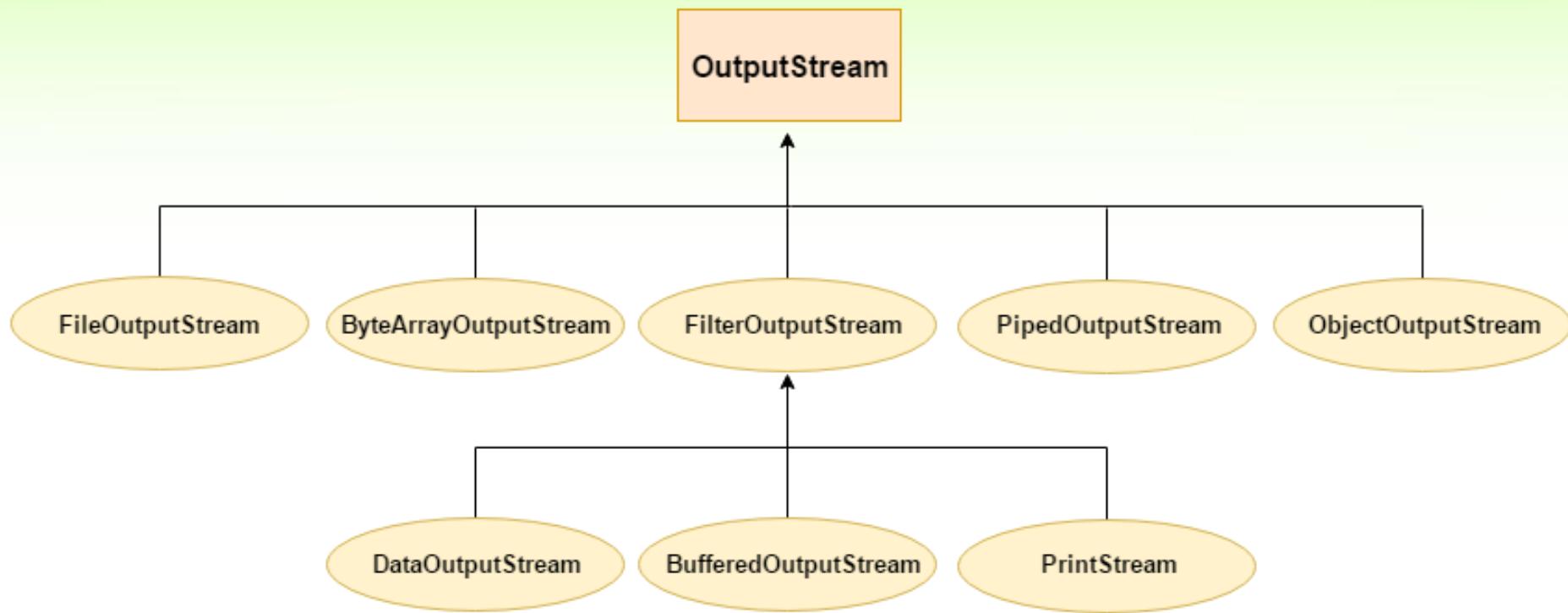
OUTPUTSTREAM/INPUTSTREAM



OUTPUTSTREAM CLASS

- ❖ OutputStream class is an abstract class. It is the super class of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.
- ❖ Method
- ❖ 1) **public void write(int) throws IOException** - is used to write a byte to the current output stream.
- ❖ 2) **public void write(byte[]) throws IOException** - is used to write an array of byte to the current output stream.
- ❖ 3) **public void flush() throws IOException** - flushes the current output stream.
- ❖ 4) **public void close() throws IOException** - is used to close the current output stream.

OUTPUTSTREAM HIERARCHY



FILEOUTPUTSTREAM CLASS

- ❖ Java FileOutputStream is an output stream used for writing data to a file.
- ❖ If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use FileWriter than FileOutputStream.

FILEOUTPUTSTREAM CLASS METHODS

- ❖ **protected void finalize()** - It is used to clean up the connection with the file output stream.
- ❖ **void write(byte[] ary)** - It is used to write ary.length bytes from the byte array to the file output stream.
- ❖ **void write(byte[] ary, int off, int len)** - It is used to write len bytes from the byte array starting at offset off to the file output stream.
- ❖ **void write(int b)** - It is used to write the specified byte to the file output stream.
- ❖ **FileChannel getChannel()** - It is used to return the file channel object associated with the file output stream.
- ❖ **FileDescriptor getFD()** - It is used to return the file descriptor associated with the stream.
- ❖ **void close()** - It is used to closes the file output stream.

FILEINPUTSTREAM CLASS

- ❖ Java FileInputStream class obtains input bytes from a file. It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc.
- ❖ You can also read character-stream data. But, for reading streams of characters, it is recommended to use FileReader class.

FILEINPUTSTREAM CLASS METHODS

- ❖ **int available()** It is used to return the estimated number of bytes that can be read from the input stream.
- ❖ **int read()** It is used to read the byte of data from the input stream.
- ❖ **int read(byte[] b)** It is used to read up to b.length bytes of data from the input stream.
- ❖ **int read(byte[] b, int off, int len)** It is used to read up to len bytes of data from the input stream.
- ❖ **protected void finalize()** It is used to ensure that the close method is call when there is no more reference to the file input stream.
- ❖ **void close()** It is used to closes the stream.

BUFFEREDOUTPUTSTREAM CLASS

- ❖ Java BufferedOutputStream class is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.
- ❖ For adding the buffer in an OutputStream, use the BufferedOutputStream class. Let's see the syntax for adding the buffer in an OutputStream:
- ❖

```
OutputStream os= new BufferedOutputStream(new FileOutputStream("D:\\IO Package\\testout.txt"));
```

BUFFEREDOUTPUTSTREAM

- ❖ Constructor:
- ❖ **BufferedOutputStream(OutputStream os)** It creates the new buffered output stream which is used for writing the data to the specified output stream.
- ❖ **BufferedOutputStream(OutputStream os, int size)** It creates the new buffered output stream which is used for writing the data to the specified output stream with a specified buffer size.

BUFFEREDOUTPUTSTREAM METHODS

- ❖ Method:
- ❖ **void write(int b)** It writes the specified byte to the buffered output stream.
- ❖ **void write(byte[] b, int off, int len)** It write the bytes from the specified byte-input stream into a specified byte array, starting with the given offset
- ❖ **void flush()** It flushes the buffered output stream.

BUFFEREDINPUTSTREAM CLASS

- ❖ Java BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.
- ❖ The important points about BufferedInputStream are:
 - ❖ When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
 - ❖ When a BufferedInputStream is created, an internal buffer array is created.

BUFFEREDINPUTSTREAM CLASS

- ❖ Constructor
- ❖ **BufferedInputStream(InputStream IS)** It creates the BufferedInputStream and saves it argument, the input stream IS, for later use.
- ❖ **BufferedInputStream(InputStream IS, int size)** It creates the BufferedInputStream with a specified buffer size and saves it argument, the input stream IS, for later use.

BUFFEREDINPUTSTREAM METHODS

- ❖ Method
- ❖ **int available()** It returns an estimate number of bytes that can be read from the input stream without blocking by the next invocation method for the input stream.
- ❖ **int read()** It read the next byte of data from the input stream.
- ❖ **int read(byte[] b, int off, int ln)** It read the bytes from the specified byte-input stream into a specified byte array, starting with the given offset.
- ❖ **void close()** It closes the input stream and releases any of the system resources associated with the stream.
- ❖ **void reset()** It repositions the stream at a position the mark method was last called on this input stream.

READER CLASS

- ❖ The Java Reader class (`java.io.Reader`) is the base class for all Reader subclasses in the Java IO API.
- ❖ A Reader is like an `InputStream` except that it is character based rather than byte based.
- ❖ In other words, a Java Reader is intended for reading text, whereas an `InputStream` is intended for reading raw bytes.
- ❖ Some of the implementation class are `BufferedReader`, `CharArrayReader`, `FilterReader`, `InputStreamReader`, `PipedReader`, `StringReader`
- ❖ The **read()** method of a Reader returns an int which contains the char value of the next character read. If the `read()` method returns -1, there is no more data to read in the Reader, and it can be closed.

FILEREADER CLASS

- ❖ The Java.io.FileReader class is a convenience class for reading character files.
- ❖ FileReader is meant for reading streams of characters. For reading streams of raw bytes, use FileInputStream.
- ❖ Constructors of FileReader class:
 - ❖ **FileReader(String file)** - It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.
 - ❖ **FileReader(File file)** - It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.

FILEREADER CLASS - METHODS

- ❖ **int read()** - It is used to return a character in ASCII form. It returns - 1 at the end of file.
- ❖ **void close()** - It is used to close the FileReader class.

WRITER CLASS

- ❖ The Java Writer class (`java.io.Writer`) is the base class for all Writer subclasses in the Java IO API. A Writer is like an OutputStream except that it is character based rather than byte based.
- ❖ Subclasses of Writer include `OutputStreamWriter`, `CharArrayWriter`, `FileWriter`

FILEWRITER CLASS

- ❖ Java FileWriter class is used to write character-oriented data to a file. It is character-oriented class which is used for file handling in java.
- ❖ Unlike FileOutputStream class, you don't need to convert string into byte array because it provides method to write string directly.
- ❖ Constructors of FileWriter class:
- ❖ **FileWriter(String file)** - Creates a new file. It gets file name in string.
- ❖ **FileWriter(File file)** - Creates a new file. It gets file name in File object.

FILEWRITER CLASS - METHODS

- ❖ **void write(String text)** It is used to write the string into FileWriter.
- ❖ **void write(char c)** It is used to write the char into FileWriter.
- ❖ **void write(char[] c)** It is used to write char array into FileWriter.
- ❖ **void flush()** It is used to flushes the data of FileWriter.
- ❖ **void close()** It is used to close the FileWriter.

BUFFEREDREADER CLASS

- ❖ java BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by readLine() method. It makes the performance fast.
- ❖ It inherits Reader class.
- ❖ BufferedReader class constructors:
 - ❖ **BufferedReader(Reader rd)** It is used to create a buffered character input stream that uses the default size for an input buffer.
 - ❖ **BufferedReader(Reader rd, int size)** It is used to create a buffered character input stream that uses the specified size for an input buffer.
- ❖ default buffer size is 8192 characters capacity

BUFFEREDREADER CLASS METHODS

- ❖ **int read()** It is used for reading a single character.
- ❖ **int read(char[] cbuf, int off, int len)** It is used for reading characters into a portion of an array.
- ❖ **boolean markSupported()** It is used to test the input stream support for the mark and reset method.
- ❖ **String readLine()** It is used for reading a line of text.
- ❖ **boolean ready()** It is used to test whether the input stream is ready to be read.
- ❖ **long skip(long n)** It is used for skipping the characters.
- ❖ **void reset()** It repositions the stream at a position the mark method was last called on this input stream.
- ❖ **void mark(int readAheadLimit)** It is used for marking the present position in a stream.
- ❖ **void close()** It closes the input stream and releases any of the system resources associated with the stream.

BUFFEREDWRITER CLASS

- ❖ The Java.io.BufferedWriter class writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.
- ❖ Class constructors:
 - ❖ **BufferedWriter(Writer wrt)** It is used to create a buffered character output stream that uses the default size for an output buffer.
 - ❖ **BufferedWriter(Writer wrt, int size)** It is used to create a buffered character output stream that uses the specified size for an output buffer.

BUFFEREDWRITER CLASS METHODS

- ❖ **void newLine()** It is used to add a new line by writing a line separator.
- ❖ **void write(int c)** It is used to write a single character.
- ❖ **void write(char[] cbuf, int off, int len)** It is used to write a portion of an array of characters.
- ❖ **void write(String s, int off, int len)** It is used to write a portion of a string.
- ❖ **void flush()** It is used to flushes the input stream.
- ❖ **void close()** It is used to closes the input stream

SCANNER CLASS

- ❖ Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double etc. and strings. It is the easiest way to read input in a Java program'
- ❖ To create an object of Scanner class, we usually pass the predefined object System.in, which represents the standard input stream. We may pass an object of class File if we want to read input from a file.
- ❖ To read numerical values of a certain data type XYZ, the function to use is nextXYZ(). For example, to read a value of type short, we can use nextShort()
- ❖ To read strings, we use nextLine().
- ❖ To read a single character, we use next().charAt(0). next() function returns the next token/word in the input as a string and charAt(0) function returns the first character in that string.

SCANNER CLASS METHODS

- ❖ **public String next()** it returns the next token from the scanner.
- ❖ **public String nextLine()** it moves the scanner position to the next line and returns the value as a string.
- ❖ **public byte nextByte()** it scans the next token as a byte.
- ❖ **public short nextShort()** it scans the next token as a short value.
- ❖ **public int nextInt()** it scans the next token as an int value.
- ❖ **public long nextLong()** it scans the next token as a long value.
- ❖ **public float nextFloat()** it scans the next token as a float value.
- ❖ **public double nextDouble()** it scans the next token as a double value.

SCANNER CLASS /BUFFEREDREADER

- ❖ The Scanner has a little buffer (1KB char buffer) as opposed to the BufferedReader (8KB byte buffer), but it's more than enough.
- ❖ BufferedReader is a bit faster as compared to scanner because scanner does parsing of input data and BufferedReader simply reads sequence of characters.

SCANNER CLASS /BUFFEREDREADER

- ❖ java.util.Scanner class is a simple text scanner which can parse primitive types and strings. It internally uses regular expressions to read different types.
- ❖ Java.io.BufferedReader class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of sequence of characters
- ❖ BufferedReader is synchronous while Scanner is not. BufferedReader should be used if we are working with multiple threads.
- ❖ BufferedReader has significantly larger buffer memory than Scanner.

SCANNER CLASS /BUFFEREDREADER

- ❖ The Scanner has a little buffer (1KB char buffer) as opposed to the BufferedReader (8KB byte buffer), but it's more than enough.
- ❖ BufferedReader is a bit faster as compared to scanner because scanner does parsing of input data and BufferedReader simply reads sequence of characters.

XML

- ❖ XML is a software- and hardware-independent tool for storing and transporting data.
- ❖ XML stands for eXtensible Markup Language
- ❖ XML is a markup language much like HTML
- ❖ XML was designed to store and transport data
- ❖ XML was designed to be self-descriptive
- ❖ XML is a W3C Recommendation

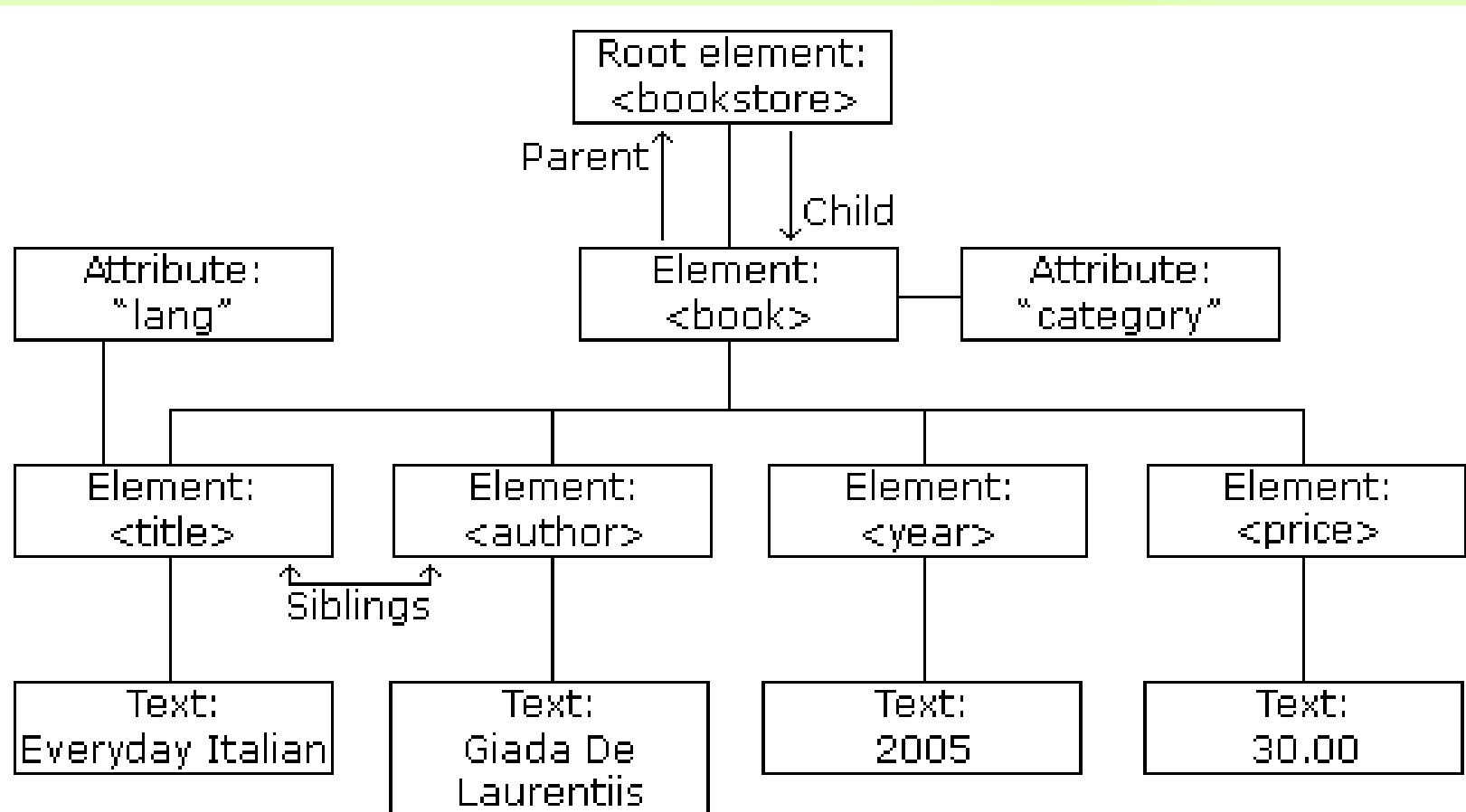
XML

- ❖ Sample xml

```
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

- ❖ XML Does Not Use Predefined Tags
- ❖ Most XML applications will work as expected even if new data is added (or removed).
- ❖ XML Separates Data from Presentation

XML TREE



XML SYNTAX

- ❖ XML uses a much self-describing syntax.
- ❖ A prolog defines the XML version and the character encoding:

```
<?xml version="1.0" encoding="UTF-8"?>
```

- ❖ The next line is the root element of the document
- ❖ The next line starts a <book> element
- ❖ The <book> elements have 4 child elements: <title>, <author>, <year>, <price>
- ❖ The next line ends the book element:

XML SYNTAX RULES

- ❖ XML Documents Must Have a Root Element
- ❖ All XML Elements Must Have a Closing Tag
- ❖ XML Tags are Case Sensitive
- ❖ XML Elements Must be Properly Nested
- ❖ XML Attribute Values Must be Quoted
- ❖ Entity References
 - ❖ < < less than
 - ❖ > > greater than
 - ❖ & & ampersand
 - ❖ ' ' apostrophe
 - ❖ " " quotation mark
- ❖ <!-- This is a comment -->

JAVA DOM PARSER

- ❖ The Document Object Model (DOM) is an official recommendation of the World Wide Web Consortium (W3C). It defines an interface that enables programs to access and update the style, structure, and contents of XML documents. XML parsers that support DOM implement this interface.
- ❖ When you parse an XML document with a DOM parser, you get back a tree structure that contains all of the elements of your document. The DOM provides a variety of functions you can use to examine the contents and structure of the document.

DOM INTERFACES

- ❖ The DOM defines several Java interfaces. Here are the most common interfaces –
 - ❖ Node – The base datatype of the DOM.
 - ❖ Element – The vast majority of the objects you'll deal with are Elements.
 - ❖ Attr – Represents an attribute of an element.
 - ❖ Text – The actual content of an Element or Attr.
 - ❖ Document – Represents the entire XML document. A Document object is often referred to as a DOM tree.

COMMON DOM METHODS

- ❖ `Document.getDocumentElement()` – Returns the root element of the document.
- ❖ `Node.getFirstChild()` – Returns the first child of a given Node.
- ❖ `Node.getLastChild()` – Returns the last child of a given Node.
- ❖ `Node.getNextSibling()` – These methods return the next sibling of a given Node.
- ❖ `Node.getPreviousSibling()` – These methods return the previous sibling of a given Node.
- ❖ `Node.getAttribute(attrName)` – For a given Node, it returns the attribute with the requested name.

STEPS TO USING DOM

- ❖ Following are the steps used while parsing a document using DOM Parser.
- ❖ Import XML-related packages.
- ❖ Create a DocumentBuilder
- ❖ Create a Document from a file or stream
- ❖ Extract the root element
- ❖ Examine attributes
- ❖ Examine sub-elements

XML VALIDATOR

- ❖ A "well formed" XML document is not the same as a "valid" XML document.
- ❖ A "valid" XML document must be well formed. In addition, it must conform to a document type definition.
- ❖ There are two different document type definitions that can be used with XML:
- ❖ DTD - The original Document Type Definition
- ❖ XML Schema - An XML-based alternative to DTD

XML DTD

- ❖ An XML document with correct syntax is called "Well Formed".
- ❖ An XML document validated against a DTD is both "Well Formed" and "Valid".

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "Note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

XML DTD

- ❖ The purpose of a DTD is to define the structure of an XML document. It defines the structure with a list of legal elements:

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

XML DTD

- ❖ !DOCTYPE note defines that the root element of the document is note
- ❖ !ELEMENT note defines that the note element must contain the elements: "to, from, heading, body"
- ❖ !ELEMENT to defines the to element to be of type "#PCDATA"
- ❖ !ELEMENT from defines the from element to be of type "#PCDATA"
- ❖ !ELEMENT heading defines the heading element to be of type "#PCDATA"
- ❖ !ELEMENT body defines the body element to be of type "#PCDATA"

XML SCHEMA

- ❖ An XML Schema describes the structure of an XML document, just like a DTD.
- ❖ An XML document with correct syntax is called "Well Formed".
- ❖ An XML document validated against an XML Schema is both "Well Formed" and "Valid".
- ❖ XML Schema is an XML-based alternative to DTD:

XML SCHEMA

```
<xss:element name="note">

<xss:complexType>
  <xss:sequence>
    <xss:element name="to" type="xs:string"/>
    <xss:element name="from" type="xs:string"/>
    <xss:element name="heading" type="xs:string"/>
    <xss:element name="body" type="xs:string"/>
  </xss:sequence>
</xss:complexType>

</xss:element>
```

XML SCHEMA

- ❖ <xs:element name="note"> defines the element called "note"
- ❖ <xs:complexType> the "note" element is a complex type
- ❖ <xs:sequence> the complex type is a sequence of elements
- ❖ <xs:element name="to" type="xs:string"> the element "to" is of type string (text)
- ❖ <xs:element name="from" type="xs:string"> the element "from" is of type string
- ❖ <xs:element name="heading" type="xs:string"> the element "heading" is of type string
- ❖ <xs:element name="body" type="xs:string"> the element "body" is of type string

XML SCHEMA

- ❖ The purpose of an XML Schema is to define the legal building blocks of an XML document:
- ❖ the elements and attributes that can appear in a document
- ❖ the number of (and order of) child elements
- ❖ data types for elements and attributes
- ❖ default and fixed values for elements and attributes

XML SCHEMA

- ❖ XML Schemas are More Powerful than DTD
 - ❖ XML Schemas are written in XML
 - ❖ XML Schemas are extensible to additions
 - ❖ XML Schemas support data types
-
- ❖ One of the greatest strengths of XML Schemas is the support for data types:
 - ❖ It is easier to describe document content
 - ❖ It is easier to define restrictions on data
 - ❖ It is easier to validate the correctness of data
 - ❖ It is easier to convert data between different data types

JAVA ANNOTATIONS

- ❖ Java Annotation is a tag that represents the metadata i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.
- ❖ There are several built-in annotations in java. Some annotations are applied to java code and some to other annotations.
- ❖ @Override
- ❖ @SuppressWarnings
- ❖ @Deprecated

@Override

- ❖ @Override annotation assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.

```
class Animal{  
void eatSomething(){System.out.println("eating  
something");}  
}
```

```
class Dog extends Animal{  
@Override  
void eatsomething(){System.out.println("eating  
foods");}//should be eatSomething  
}
```

@SUPPRESSWARNINGS

- ❖ @SuppressWarnings annotation: is used to suppress warnings issued by the compiler.

```
import java.util.*;  
class TestAnnotation2{  
    @SuppressWarnings("unchecked")  
    public static void main(String args[]){  
        ArrayList list=new ArrayList();  
        list.add("sonoo");  
        list.add("vimal");  
        list.add("ratan");  
    }  
}
```

@DEPRECATED

- ❖ @Deprecated annoation marks that this method is deprecated so compiler prints warning. It informs user that it may be removed in the future versions. So, it is better not to use such methods.

```
class A{  
void m(){System.out.println("hello m");}  
  
@Deprecated  
void n(){System.out.println("hello n");}  
}
```