Core Java Part - II

Chapters:

OBJECT ORIENTED PROGRAMMING

- ***Encapsulation (Data hiding & Method abstraction)***
- ***Inheritance (IS-A Relationship)***
- ***Polymorphism***
    - ***Method Signature***
    - ***Overloading***
    - ***Overriding***
- ***Constructors***
- ***this or super***
- ***Final & static***
- ***Abstract***
- ***Interface***

Encapsulation

Data hiding

One of the sound object-oriented programming techniques is **hiding the data** within the class by declaring them with **private** accessibility modifier. Making such variables available outside the class through **public setter and getter methods**. Data hiding says "**restrict** access and modification of internal data items through the use of getter and setter methods".

Example:

```
public class Authentication{
        //Data hiding
        private String username;
        private String password;

        public void setUsername(String name){
                if((name != null) && name!=""){
                        username = name;
                }else{
                        Sop("Username cannot be null or empty");
                }
        }
        public String getUsername(){
                return username;
        }
        ….
        ….
}
Authentication obj = new Authentication();
//obj.username="aspire";//Compilation error.
obj.setUsername("aspire");//Ok
obj.setUsername("");//validation error saying "Username cannot be null or empty."
```

Method Abstraction

One of the sound object-oriented programming techniques is “Hiding method implementation complexity from outside users” is called as method abstraction.

```
//Method abstraction public
boolean isValidUser(){
        //Use JDBC API for obtaining connection.
        Connection con = DriverManager.getConnection(“jdbc:oracle:thin:@localhost:1521:XE”, “scott”,
        “tiger”);
        Check with database whether user is already registered or
        not. If valid user then
                return true;
        else
                return false;
}
```

Encapsulation = Data hiding + Method Abstraction

Encapsulation combines (or bundles) data members with method members together (or the facility that bundles data with the operations that perform on that data is called as encapsulation).

Example:

```
public class Authentication{
                //Data hiding
                private String username;
                private String password;

                public void setUsername(String name){
                        if((name != null) && name!=”” ){
                                username = name;
                        }
                }
                public void setPassword(String pwd){
                        if((pwd != null) && pwd!=”” ){
                                Password = pwd;
                        }
                }

                //Method abstraction
                 public boolean isValidUser(){
                        //Use JDBC API for obtaining connection.
                        Connection con = DriverManager.getConnection(“jdbc:oracle:thin:@localhost:1521:XE”,
                        “scott”, “tiger”);
                        Check with database whether user is already registered or not.
                        If valid user then
                                return true;
                        else
                                return false;
```

Phone: (+91) 9865358359

```
                }
}
```

//Client program

```
Public class AccessSite{
  public static void main(String[] args){
        Authentication auth = new
        Authentication(); //set website username
        and password auth.setUsername("aspire");
        auth.setPassword("aspire123");

        boolean success= auth.isValidUser();
        if(success){
                System.out.println("Logged into INBOX…");
        }else{
                System.out.println("Invalid username / password…");
        }
 }
}
```

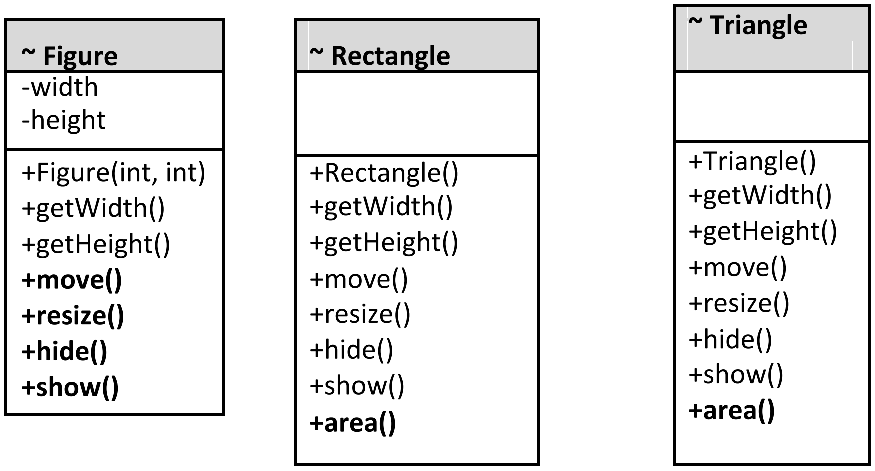
Inheritance

Define common methods in parent class and specific methods in child class.

Example:

```
class Figure{           //Parent class
        private int width;
        private int height;
        public Figure(int w, int h )
        {
        width = w; height = h;
        }
        public int getWidth() {
                return width;
        }
        public int getHeight() {
                return height;
        }
        public void move(int x, int y){
                System.out.println("Move method...");
        }
        public void resize(int width, int height){
                System.out.println("Resize method....");
        }
        public void hide(){
                System.out.println("Hide method...");
        }
        public void show(){
                System.out.println("show method...");
        }
}
class Rectangle extends Figure{ //child class
        public Rectangle(int w, int h)
         {
                super(w,h);//Invokes immediate parent class matching constructor
        }
        //subclass specific method.
        double area(){
                return getHeight()* getWidth();
        }
}

class Triangle extends Figure{
        public Triangle(int w, int h){
                super(w,h);
        }
        //subclass specific method.
        double area(){
                Return 0.5 * getHeight()* getWidth();
         }
}
```



Note:

Accessibility modifier	UML symbol
Private	-
Default	~
Protected	#
Public	+

In the above example, Rectangle **IS-A** Figure. Similarly, Triangle **IS-A** Figure. So, inheritance is also called as **IS-A** relationship.

The base class common methods move(), resize(), hide(), and show() are **reused** in the subclasses Rectangle and Triangle. Hence the main advantage with inheritance is Code **REUSABILITY**.

The base class is also called as superclass, supertype, or parent class. The derived class is also called as subclass, subtype, or child class.

In java, the **extends** keyword is used to inherit a superclass.

Example:

class Rectangle **extends** Figure{}

In java, a class can inherit at most one superclass. This is called single level inheritance.

Example:

```
class C extends A, B{}              //Compilation error.
```

Conclusion: Java does not support multiple inheritance with classes but supports with interfaces.

The inheritance relationship is represented as **Hierarchical classification**.

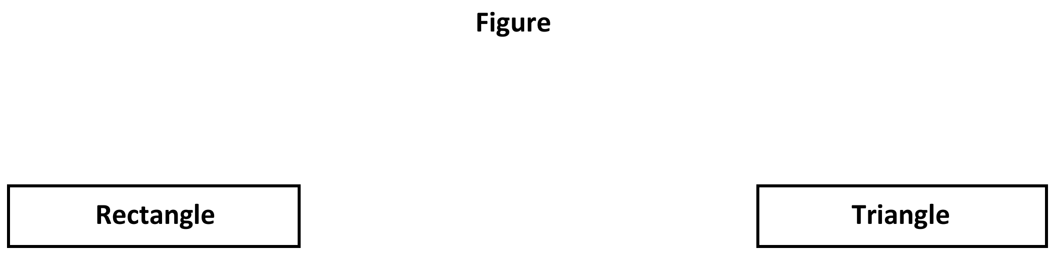


Figure 1: Inheritance hierarchy

The classes higher up in the hierarchy are more **generalized** and the classes lower down in the hierarchy are more **specialized**.

Case 1: Subclass specific methods cannot be invoked using superclass reference variable.

Example:

```
Figure fRef = new Figure(10,20);
fRef.move();
                                    //Compilation error. Undefined method.
//fRef.area();
```

Case 2: As expected, subtype object is assigned to subtype reference variable:

Example:

```
Rectangle rRef = new                          Rectangle(10,20);
rRef.move();            //Compiles and runs without errors.
rRef.area();
```

Case 3: The subclass object can be assigned to a superclass reference variable. But, the subclass specific methods cannot be invoked using superclass reference variable, because the compiler only knows reference type methods but not actual runtime object methods.

This assignment involves a widening reference conversion (upcasting).

Example:

```
Figure fRef = new Rectangle(10,30); //Reference type widening. Subtype object is assigned to supertype ref var.
fRef.move();//Ok
```

```
fRef.area();                    // Compilation error. Undefined method.
```

Polymorphism

Polymorphism translates from Greek as **many forms** (*poly - many & morph - forms).* Overloading and overriding are the two different types of polymorphisms.

Overloading – Static polymorphism
Overriding – Dynamic polymorphism

Method Signature

It is the combination of the method name and its parameters list. Method signature does not consider return type and method modifiers.

Example:

public int **add(int, int)**{}
public float **add(float, float)**{}

Method overloading

Two or more methods are said to be overloaded if and only if methods with same name but differs in parameters list (Differs in Number of parameters, Parameter type, or Parameter order), and without considering return type, method modifiers i.e. Overloaded methods must have different method signature irrespective of their return type, modifiers.

Example: class
Addition{

```
        public int add(int x, int y){}
        public int add(int x, int y, int z){}

        public float add(float x, float y){} public
        double add(double x, double y){}

        public float add(int x, float y){}
        public float add(float x, int y){}
}
```

All above methods are overloaded with each other.

Addition obj = new Addition();
obj.add(10,20);//**compiler** maps with add(int, int)
obj.add(10,20,30); //**compiler** maps with add(int, int, int)

The overloaded methods are resolved by the java compiler at compilation time. Hence overloading is also known as 'Static Polymorphism' or 'Early Binding'.

Method overriding

A subclass can re-implement superclass methods to provide subclass specific implementation. The method in superclass is called as **overridden** method, whereas method in subclass is called as **overriding** method.

Method overriding rules:

a) The accessibility modifier of subclass method must be same or less restrictive. **AND**
b) The superclass and subclass methods must have same return type. **AND**

c) The superclass and subclass method signature must be same (i.e., method name, the number of parameters, parameter types, parameter order). **AND**
d) The throws clause in overriding method can throw all, none or a subset of the checked exceptions which are specified in the throws clause of the overridden method, but not more checked exceptions.

Example:
```
class Figure {
        int width;
        int height;
        Figure(int w, int h) {
                width  =  w;
                height = h;
        }
        //overridden method
        public double area(){
                return 0.0; //unknow
        }
}
class Rectangle extends Figure {
        Rectangle(int w, int h) {
                super(w, h);
        }
        //overriding method
        public double area() {
                return width * height;
        }
}
public class OverridingDemo {
        public static void main(String[] args) {
                //superclass object assigned to superclass reference variable
                Figure fRef = new Figure();
                Sop(fRef.area()); //0.0          //area() from Figure class

                //subclass object assigned to subclass reference
                variable
                Figure fRef1 = new Rectangle(10,20);
                Sop(fRef1.area()); //200.0       //area() from Rectangle class.
        }
}
```

The method overriding is resolved at runtime based on **actual object type** but not Reference type. Hence method overriding is also known as Late Binding or dynamic polymorphism.

Dynamic Method Dispatch (DMD):

Method overriding is also known as Dynamic Method Dispatching.

Example:
```
class Figure{
```

```
        void area(){}
}
class Rectangle extends
        Figure{ void area(){}
}
class Triangle extends
        Figure{ void area(){}
}

Figure fRef = new Rectangle(10,20);
fRef.area();            //area() method from Rectangle object is invoked.

fRef = new Triangle(10,20);
fRef.area();            //area() method from Triangle object is invoked.
```

The method overriding is also called as Dynamin Method Dispatching, Dynamic Polymorphism, or Late Binding.

Difference between overloading and overriding:

Criteria	Overriding	Overloading
Method signature	Must be same	Must be different.
Return type	Must be same.	Never considered.
Accessibility modifier	Must be same or Less restrictive.	Never considered.
Throws clause	Must not throw new checked exceptions.	Never considered.
Declaration context	A method must only be overridden in a subclass.	A method can be overloaded in the same or in a sub class.
Method call resolution	Method overriding resolved at runtime based on actual object type. It is also called as **dynamic polymorphism** or **late binding**.	Method overloading resolved at compile time based on Reference type. It is also called as **static polymorphism** or **early binding**.

Constructor

The purpose of the constructor is **initialization** followed by **instantiation**.

Constructor Rules:

a) The name of constructor must be same as class name.

b) The only applicable modifiers for the constructors are:

i) Public ii) protected iii) default modifier iv) private

c) The constructor must not have any return type not even void.

Example:

```
public class Box{
        Int width;
        Int height;
        Int depth;
        public Box(int w, int h, int d){ //Initialization
```

```
                width = w;
                height = h;
                depth = d;
        }
}

Box b = new Box(1,2,3); //Instantiation
```

Constructors are either Default constructors or Parameterized constructors:

Constructors

Default constructors | Parameterized construcotrs

Implicit default constructor | Explicit default constructors | Primitive type | Reference type

Default constructor

Constructor without parameters is called as Default constructor. It is also called as **no-arg constructor**.
There are two types of default constructors:

a) Implicit Default Constructor

If no constructor is declared explicitly inside class, the Java Compiler automatically includes constructor without parameters called as Implicit Default Constructor.

Example:

```
public class Box{}

//After compilation
public class Box{
        public Box(){} //implicit default constructor
}
```

b) Explicit Default Constructor

We can declare constructor without parameters list explicitly is called as explicit default constructor.

Example:

```
Class Box{
        Box(){ //explicit default
                constrctor Width = 1;
                Height = 2;
                Depth = 3;
        }
}
```

Parameterized Constructor

Constructor can take either Primitive or Reference type as its parameters is called as **parameterized constructor**.

Example:

```
class Box{
        int  width;
        int height;
        int depth;
        Box(){  //no-arg  constructor
                Width =1;
                Height = 2;
                Depth = 3;
        }
        Box(int width){
                this.width = width;
                height = 2;
                depth = 3;
        }
        Box(int  width,  int  height){
                this.width  =  width;
                this.height           =
                height; depth = 3;
        }
        /*Box(int width, int height, int depth){
                this.width = width;
                this.height = height;
                this.depth = depth;
        }*/
        //short cut approach
        Box(int width, int height, int depth){
                this(width, height); //calls matching constructor from the same class
                this.depth = depth;
        }
        Box(Box b){
                this.width  =  b.width;
                this.height = b.height;
                this.depth = b.depth;
        }
}
public class ConstructorDemo {
        public static void main(String[] args) {
                Box b1 = new Box(1,2,3); Box
                b2 = new Box(b1);
        }
}
```

Constructors are overloaded with each other. Also, Constructors cannot be inherited to subclass.

Constructor Chaining

Super class constructors are always executed before subclass constructors is called as constructor chaining.

Example:

```
class One{
	One(){
		System.out.println("1");
	}
}
class Two extends One{
	Two(){
		System.out.println("2");
	}
}
class Three extends
	Two{ Three(){
		System.out.println("3");
	}
}
public class ConstructorChainingDemo {
	public static void main(String[] args) {
		Three obj = new Three();
	}
}
```

O/P:

```
1
2
3
```

super or this

There are two forms of 'super' or 'this': i) without parantheses ii) with parantheses

'this' without parentheses are used to invoke **fields** and **methods** from the same class.

'super' without parantheses is used to invoke its immediate superclass **fields** and **methods**.

'this()' is used to call matching constructor from the same class.

'super()' is used to call matching constructor from immediate superclass.

Example:

```
	class One{
		Int age = 20;
	}
	class Two extends One{
		Int age = 30;
		Void disp(){
			SOP(age); // 30
			SOP(this.age);//30
			SOP(super.age); // 20
		}
```

```
        }
```

final

The ‘final’ is a modifier used with:

a) Variables b) Methods c) Classes

final w.r.t variables

The value of the final variable is always ***fixed*** i.e., once it is initialized, it never re-initialized.

Example:

```
        final int SIZE = 10;
        //SIZE = 20; //C.E. Final variable cannot be re-initialized. It is
```
always recommended to use Upper case for final variables names.

Final w.r.t methods

Final methods are always fully (completely) implemented.

Example:

```
        class Figure{
                final void move(){}
                final void resize(){}
                final void show(){}
                final void hide(){}
                public double area(){
                        return 0.0; //Unknow
                }
        }
```

Final methods cannot be overridden i.e., final keyword **prevents** method overriding.

final w.r.t classes

If all methods in class are completely implemented, then instead of declaring every method as final, better declare class itself as final.

By default, all methods in a final class are final.

Example:

```
        public final class One{ public
                void meth1(){} public
                void meth2(){}
        }
```

Final classes cannot be extended.

Example:

```
        class Two extends One{}//C.E
```

Note:

1) Final with variables prevents re-initialization.
2) Final with methods prevents method overriding.
3) Final with classes prevents inheritance.

static

The 'static' modifier is used with:

a) Variables b) methods c) Blocks d) Inner class

Static members are **Independent** of any object creation i.e., static members are **Common** across all objects. Static members never associated with any object but directly associated with class. Hence, static variables are also called as **Class variables or Global variables**.

Example:

```
class Box{
        Private int width;
        Private int height;
        Private int depth;
        public static int count; //static variable
        Box(int w, int h, int d){width = w; height = h; depth = d; count++;}
}
Box b1 = new Box(1,2,3);
Box b2 = new Box(4,5,6);
sop(Box.count); //2
```

Static members are executed when class is loaded into JVM. Hence, static members are accessed without (or before) creating an object itself.

Note:

1) Instance variables are associated with **Heap area**.
2) Static variables are associated with **Method area.**
3) Local variables are associated with **Stack area**.
4) Final variables are associated with **Constant Pool area**.

Static methods

Rules:

1) A static method cannot access instance variables.
2) A static method cannot access instance methods.
3) Cannot use this or super from static context.

We can use static members from non-static context, but not vice versa.

It is always recommended to use class name to access static members rather than object name.

abstract

The **'abstract'** modifier is used
with: a) methods b)classes.

Declare method as abstract if the method implementation is **unknown**. Always, abstract methods must ends with **semi-colon**.

Example:

```
public abstract double area();
```

Abstract class Rules:

Declare class as abstract if:

a) Atleast one abstract method, but all other methods are concrete.
b) All methods are abstract methods, i.e., none of the method is a concrete method.
c) All methods are concrete, i.e., none of the method is an abstract method.

If a class inherits abstract class using extends keyword, the subclass must either implement all abstract methods or declare subclass also as abstract class.

Abstract classes must not be instantiated, but abstract classes are used as reference variables to achieve dynamic polymorphism.

Example:

```
public abstract class
        Figure{ Int width;
        Int height;
        Figure(int width, int height){this.width = width; this.height = height;}
        Public final void move(){}
        Public final void resize(){}
        Public final void hide(){}
        Public final void show(){}

        public abstract double area();
}
class Rectangle extends Figure{
        Rectangle(int w, int h){super(w,
        h);} @Override
        public double area(){
                Return width * height;
        }
}
Class Triangle extends Figure{ Triangle(int
        w, int h){super(w, h);} @Override

        Public double area(){
                Return 0.5 * width * height;
        }
}

Public class AbstractDemo{
        Public static void main(String[] args){
                //Figure fRef = new Figure(1,2); //C.E since abstract classes cannot be instantiated.
                Figure fRef = new Rectangle(10,20);//Subclass object is assigned to superclass ref variable
                SOP(fRef.area());        // The area() method from rectangle is invoked.
                fRef = new Triangle(10,20);
                SOP(fRef.area());        // The area() method from triangle is invoked.
        }
}
```

Advantages with Abstract classes:

1) Forces the subclass to provide method implementation
2) Achieves Dynamic Polymorphism.

Interfaces

Interface provides service details but not its implementation details.

For example, the stack interface says its operations push() and pop() but not its actual implementation details such as array or linked list. We (service provider) can change stack implementation from array to linked list without affecting to client applications (service consumer).

Service Consumer

Service Details (interface)

Service Provider

Syntax:

<modifier> **interface** <interface name> **[extends** interface$_{1}$,… interface$_{n}$**]**{
    [field declarations]
    [method declarations]
}

Fields w.r.t Interface

By default, all interface fields are **public static final**.
Hence, All of the following field declarations are equal:

a) **public static final** int SIZE = 10; //Declaring public static final explicitly is redundant but not error.
b) static final int SIZE = 10;
c) final int SIZE = 10;
d) int SIZE = 10;

All interface variables must be initialized in the declaration section.
Example:

Public static final int SIZE = **10**; //declaration cum initialization
Public static final int SIZE; //C.E saying Missing Initialization

Methods w.r.t Interface

By default, all interface methods are **public abstract i.e.,** interface is a pure abstract class. Hence, All of the following methods declarations are equal:

a) **public abstract** void meth(); //Declaring public abstract explicitly is redundant but not error.
b) public void meth();
c) abstract void meth();
d) void meth();

All methods in an interface are abstract, hence, interfaces cannot be instantiated. However, interfaces can be used as reference variables.

Example:

```
public interface Stack{
        public void push(int x);
        public int pop();
}

public class StackImpl implements
        Stack{ private final int SIZE = 10;
        public void push(int x){
                //impl using arrays
        }
        public int pop(){
                //impl using arrays
        }
}

Stack sRef = new StackImpl(); //Ok
sRef.push(10);
sRef.pop();
```

Constructors w.r.t interface

All variables in an interface are final constants, which must be initialized in the declaration section itself.
Also, all methods in an interface are abstract, such an interfaces cannot be instantiated.
Hence, Interface does not need constructors at all i.e., **interfaces never have constructors at all**.

Inheritance w.r.t Interface

A class inherits another class using **extends** keyword.
A class inherits interface(s) using **implements** keyword.
An interface inherits interface(s) using **extends** keyword.
Either class or an interface can inherit any number of interfaces.

Example:

Public interface One{}
Public class Two
implements One{}
	Public interface One{}
	Public interface Two
	extends One{}
		Public interface One{}
		Public interface Two{}
		Public class Three
		implements One, Two{}
			Public interface One{}
			Public interface Two{}
			Public interface Three
			extends One, Two{}

Java supports multiple inheritance by inheriting multiple interfaces.

The subclass of an interface either implement all abstract methods or declare subclass as an abstract class.
The accessibility modifier of the subclass method must be always **public**.

Example:

```
        Public Interface One{
                public void meth1();
```

```
            Public void meth2();
    }

     class Two implements One{              abstract class Two implements One{
            public void meth1(){}                      public void meth1(){}
            public void meth2(){}           }
     }
```

Marker Interface

If our class gets special privilege by inheriting an interface, such an interface is called as marker interface.

Example:

Java.io.Serializable
Java.lang.Runnable
…

If an interface does not have any methods, it is always called as Marker interface.

Example:

Java.io.Serializable

Even though an interface contains methods which gives special privilege to our class, then such an interface is also called as Marker Interface.

Example:

Java.lang.Runnable - run() method.

Question: Though interface is a pure abstract class, why do we need an interface?

To support multiple inheritance and if a subclass already implements another class, then subclass cannot inherit our abstract class. This restriction is not applicable for an interface, since a class can inherit any number of interfaces.

Difference between abstract class and Interface:

Abstract class	Interface
Can contain concrete methods.	All methods are abstract.
Can contains non final & non static variables.	All variables must be static and final.
Can contains any number of constructors.	Never contains constructors at all.
Does not support multiple inheritance.	**Supports multiple inheritance.**

EXCEPTIONS

Introduction

Definition: An **unexpected problem** occurs while running our application at **runtime** is called as an exception.
For example: ArithmeticExcetion, ArrayIndexOutOfBoundsException, FileNotFoundException, etc.

Example:

```
public class ExceptionDemo {
        public static void main(String[] args) {
                int a = Integer.parseInt(args[0]);
                int b = Integer.parseInt(args[1]);
                int c = a/b;
                System.out.println("The result="+c);
                System.out.println("End of main method");
        }
}
```

o/p:v java 10 0

```
Exception in thread "main" java.lang.ArithmeticException: / by
        zero at ExceptionDemo.main(ExceptionDemo.java:5)
```

[Abnormal Termination]

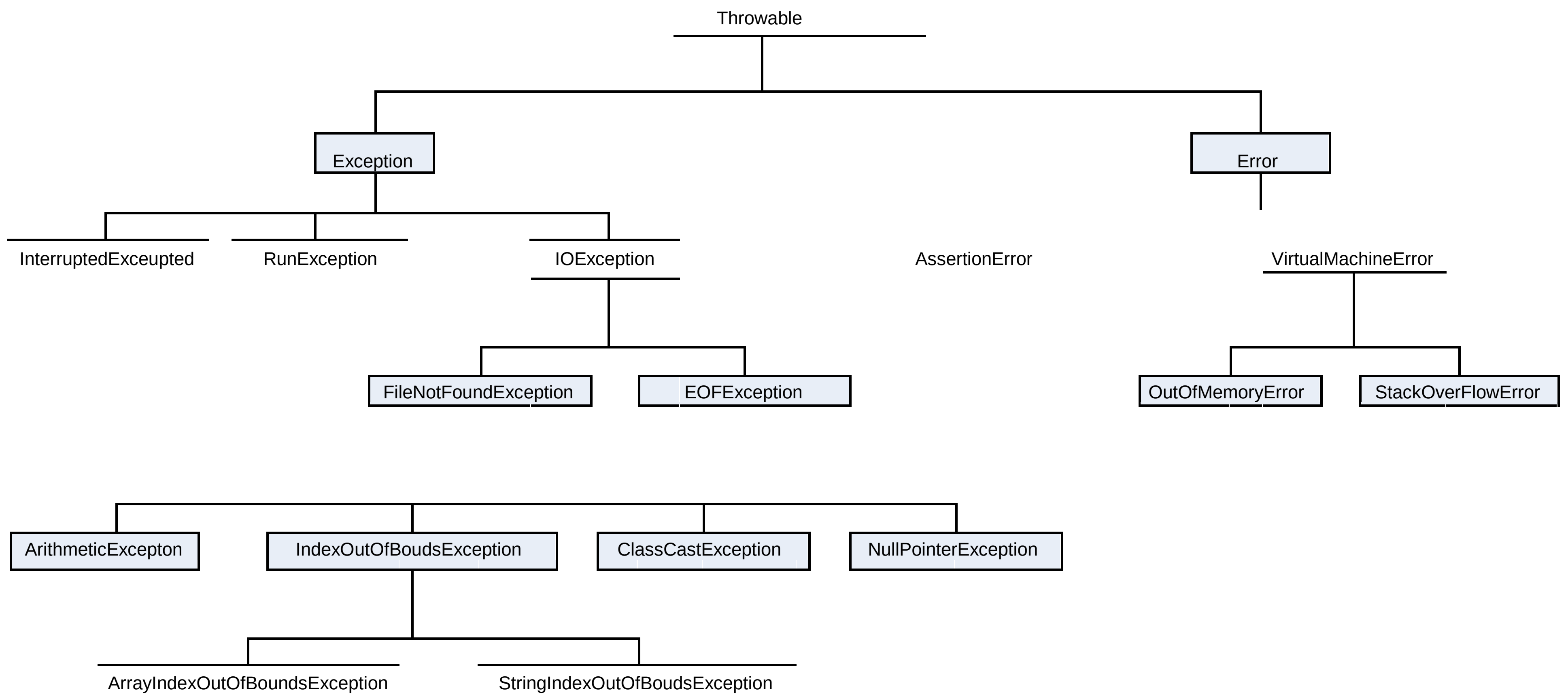
Exception Hierarchy

The root class for all exceptions is **java.lang.Throwable**.
The two main subclasses of Throwable are Exception and Error.

Difference between Exception and Error:

Exception	Error
Exceptions are **caused by our program** and hence **recovarable**. After recovering (handling) exception, the program will continue as if there is no exception.	Errors are not caused by our application and rather they are caused due to **lack of system resources** such as memory, etc. Hence, errors are **not recoverable**. The program will be terminated abnormally.

R.ARAVIND


Throwable
Exception
Error
InterruptedExceupted
RunException
IOException
AssertionError
VirtualMachineError
FileNotFoundException
EOFException
OutOfMemoryError
StackOverFlowError
ArithmeticExcepton
IndexOutOfBoudsException
ClassCastException
NullPointerException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoudsException

Methods to display error Information:

The **Java.lang.Throwable** root class contains the following 3 methods to print exception information:

1) **Public void printStackTrace()**

This method is used to print following error information:

a) Exception name
b) Exception Message
c) Line number which causes the exception
d) Method belongs to the above line number
e) Class belongs to the above method.
f) File name belongs to the above class.

Example:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at ExceptionDemo.main(ExceptionDemo.java:5)
```

2) **Public String toString()**

This method is used to print following error information:

a) Exception name
b) Exception Message

Example:

```
java.lang.ArithmeticException: / by zero
```

3) **Public String getMessage()**

This method is used to print following error information. a) Exception Description.

Example:

```
/ by zero
```

Exception handling using try-catch-finally

It is always a good programming practice to handle an exception for graceful termination (Normal termination) of the program i.e., the purpose of handling an exception is to continue program execution as if there is no exception.

Try block

It contains the statements which likely to throw an exception.
If an exception is raised, the remaining statements in the try block are **skipped**.
Try block must be followed by either catch or finally or both.

Catch block

Catch block is used to handle an exception.
The catch block is executed only if try block throws an exception.
Catch block cannot be written without try block. For each try block there can be zero or more catch blocks.

Finally block

It is not recommended to write **clean up code** inside try block, because there is no guaranty for the execution of all statements inside try.

It is not recommended to write clean up code inside catch block because it won't executed if there is no Exception.

We required one place to maintain cleanup code which should be executed always irrespective of whether exception is raised or not and exception is handle or not. Such type of block is called as finally block. Hence the main objective of finally block is to maintain **cleanup code**.
Only one finally block can be associated with try block.

Example:

```
public class ExceptionHandlingDemo {
    public static void main(String[] args) {
            int a = Integer.parseInt(args[0]);
            int b = Integer.parseInt(args[1]);
            try{
                    int c = a/b;
            }catch(Exception ae){
                     System.out.println("Denominator should not be 0");
                     ae.printStackTrace();
            }finally{
                    System.out.println("finally block");
            }
            System.out.println("End of main method.");
    }
}
```

Java ExceptionHandlingDemo 10 0
O/P:

```
Denominator should not be 0
java.lang.ArithmeticException: / by zero
        at ExceptionHandlingDemo.main(ExceptionHandlingDemo.java:6)
finally block
End of main method.
[Normal Termination]
```

Checked and Unchecked exceptions

If an exception(or error) is directly or indirectly inherited from **RuntimeException(or Error)** is called as Unchecked exception i.e., in the exception hierarchy, if RuntimeException (or Error) presents then it is a Unchecked exception, otherwise, it is a checked exception.
Example:

Unchecked exception	Checked exception

java.lang.Object
    java.lang.Throwable
        java.lang.Exception
            java.lang.RuntimeException
            └ java.lang.ArithmeticException
	java.lang.Object
	    java.lang.Throwable
	        java.lang.Exception
	            java.lang.ClassNotFoundException

Checked exception must be handled, otherwise the program will not be compiled.
Example:

```
Try{
        Thread.sleep(1000);
}catch(InterruptedException ie){}
```

Unchecked exception may or may not be handled. The compiler will ignore unchecked exceptions.

Example:

```
Void meth(){
      SOP(10/0);
}
```

The unchecked exceptions are caused due to invalid input data entered by end user at runtime, whose information never known by compiler in advance. Hence, unchecked exceptions are ignored by java compiler. The checked exceptions must be handled by the developer otherwise the compiler will not let the code to be compiled.

Try with Multiple Catch Blocks

The way of handling an exception is varied from exception to exception, hence it is recommended to place separate catch block for every exception.

If the statements in try block throws multiple exceptions, it is recommended to handle them individually.

In case of try with multiple catch blocks, the **order of catch blocks present is very important and it should be from subclass exception type to superclass exception type**.

The catch block for a super class exception should not shadow the catch block for the sub class exception i.e. the order of the catch blocks is from sub class to the super class exception, otherwise the program will not be compiled.

Example:

Try{
        Int[] arr = new int[5];
        SOP(arr[10]);
        String str = “hello”;
        SOP(str.charAt(10));
}catch(ArrayIndexOutOfBoundsException e){ e.printStackTrace();
}catch(StringIndexOutOfBoundsException e){
  Sop(e.toString());
}
//OK
	Try{
	        Int[] arr = new int[5];
	        SOP(arr[10]);
	        String str = “hello”;
	        SOP(str.charAt(10));
	}catch(ArrayIndexOutOfBoundsException ae){
	}catch(**IndexOutOfBoundsException** e){}
	//OK
		Try{
		        Int[] arr = new int[5];
		        SOP(arr[10]);
		        String str = “hello”;
		        SOP(str.charAt(10));
		}catch(IndexOutOfBoundsException ae){
		}catch(ArrayIndexOutOfBoundsException e){ }
		//C.E.

Nested try blocks

Try block with in another try block is called as nested try block.

Example:

```
try{
        try{
        }catch(){}
}catch(){}
```

If an exception is raised inside inner try block, initially it will look for inner catch block, if it is not matched, then it will look for outer catch block.
If an exception is raised in outer try block, then directly it will look for outer catch block.

User defined (or Customized) exceptions

Like built-in exceptions, user can define application (project) specific exceptions is called as user defined exceptions.
Like built-in exceptions, there are two types of user defined exceptions as well:

- ✓ Un-checked Exceptions
- ✓ Checked Exceptions.

Un-checked Exceptions	Checked Exceptions

package edu.aspire;
public class DivByOneException extends
RuntimeException{
Public DivByOneException( ){super();}
Public DivByOneException (String message){
super(message);
}
}
	Public class DivByOneException extends
	Exception{
	Public DivByOneException ( ){}
	Public DivByOneException (String message){
	Super(message);
	}
	}

throw

The '**throw**' keyword is used to propagate (or delegate) an exception to its caller inorder to let the calling method to handle an exception rather than implementation method.

Syntax:

throw <**throwable object**>;

Example:

throw new ArithmeticException("/ by zero");

The 'throw'keyword is always used at **block level** but not at method signature level.

All built-in exceptions are automatically thrown by the JVM. But, built-in exceptions can be programmatically thrown by the developer using throw keyword.
All user defined exceptions must be programmatically thrown using 'throw' keyword.

Example:

```
public class Division{
        public static void main(String[]
        args){ try{
                div(10,0);                //method calling
        }catch(ArithmeticException ae) {
                ae.printStackTrace();
        }
        }
        public static void div(int x, int y){ //method implementation If(y==0){

                        throw new ArithmeticException("/ by zero");
                }
```

```
                    System.out.println(x/y);
          }
}
```

Built-in exceptions w.r.t throw keyword

Using throw keword with built-in exceptions is **optional**. Hence the result of the following two programs is

exactly same. class Test{

```
public static void main(String[] args){
      System.out.println(10/0);
  }
}
//R.E : A.E
```

```
Class Test{
     public static void main(String[] args){
         throw new ArithmeticException("/ by Zero");
     }
  }
//R.E : A.E
```

User defined exceptions w.r.t throw keyword

User defined exceptions must be thrown explicitly using throw keyword. Hence the result of the the following

two programs is not same. class Test{

```
public static void main(String[] args){
      System.out.println(10/1);
  }
}
```

Compiles and Runs without exception.
O/P: 10

```
Class Test{
     public static void main(String[] args){
              throw new DivByOneException("/ by one");
     }
  }
// R.E :
```

edu.aspire.DivByOneException: / by one

throws

Throws keyword is used at **method signature level** to specify the type of exception a method throws.

Syntax:

<method modifiers> <return type> method_name(<formal parameter list>)
<throws> <ExceptionType^{1}> … <,ExceptionType^{n}>{
          <statement>
}

Checked exceptions w.r.t throws keyword:

If the throw propagates checked exceptions inside method block, then such checked exceptions must be specified at method signature using '**throws**' keyword.

Example:

class Test{
    p.s.v main(String[] args)throws I.E{
      doStuff();
    }
  Public static void doStuff() throws IE{
	class Test{
	    p.s.v main(String[] args){
	      doStuff();
	   }
	  public static void doStuff(){

throw new InterruptedException();
}
//R.E: InterruptedException
	throw new InterruptedException();
	}
	//C.E: Unhandled exception type :

InterruptedException.

If an exception is propagated to JVM, it prints the stack trace followed by terminates the thread which causes the exception. This is called as **Default Handler**.

Unchecked exceptions w.r.t throws keyword:

The compiler does not verify the unchecked exceptions in throws clause i.e., if the throw propagates unchecked exceptions inside method block, then throws may or may not specify unchecked exceptions at method signature.

Example:

class Test{
p.s.v.main(String[] args)**throws ArithmeticException**{
    throw new ArithmeticException();
}
}
//R.E: A.E
	class Test{
	p.s.v.main(String[] args){
	    throw new ArithmeticException();
	}
	}
	//R.E: A.E

Throws w.r.t method overriding

The subclass method must not throws more checked exceptions than superclass method, i.e., the overriding method in subclass may throw **none, all, or sub-set of checked exceptions from overridden method from superclass, but not more checked exceptions.** This rule is not applicable for unchecked exceptions.

Example:

class One{
  public void meth()throws IOException{
  }
}
class Two extends One{
  public void meth()throws Exception{
  }
}
//C.E: Sub class method must not throw supertype exception.
	class One{
	  public void meth()throws IOException{
	  }
	}
	class Two extends One{
	  public void meth()throws InterruptedException{
	  }
	}
	//C.E: Sub class method must not throw different checked exceptions.
		class One{
		public void meth()throws IOException{
		}
		}
		class Two extends One{
		public void meth()throws FileNotFoundException{
		  }
		}
		//R.E: IOException

The overriding method in subclass may throw different unchecked exceptions than overridden method in superclass i.e., the overriding method in subclass may or may not specify unchecked exceptions in overridden method unchecked exceptions in superclass.

class One{
public void meth()throws IndexOutOfBoundsException{
  }
}
class Two extends One{
public void meth()
  }
	class One{
	public void meth()throws IndexOutOfBoundsExceptions{
	}
	}
	class Two extends One{
	public void meth()throws RuntimeException
		class One{
		public void meth()throws IndexOutOfBoundsExceptions{
		}
		}
		class Two extends One{
		public void meth()throws ArrayIndexOutOfBoundsException

} **//No compilation error.**	} } **//No compilation error.**	} } **//No compilation error.**

Difference between Checked and Unchecked exceptions :

CheckedException	**UnCheckedException**
Except for RuntimeException, Error, and their subclasses, all other exceptions are called checked exceptions.	Exceptions inherited from Error or RuntimeException class and their subclasses are known as unchecked exceptions.
All checked exceptions must be handled, otherwise the program will not be compiled.	The compiler will not force unchecked exceptions to be handled i.e., unchecked exceptions may or may not be handled.
If a method throw checked exceptions, they must be specified in the throws clause. But, more checked exceptions must not be specified in the throws clause.	Specifying unchecked exceptions in throws clause is optional.

JAVA.LANG PACKAGE

- **Hierarchy**
  - **Object class**
  - **String**
  - **StringBuffer & StringBuider (Jdk1.5)**
  - **Wrapper classes**
  - **Autoboxing and Unboxing (Jdk1.5)**

This package contains most common classes in java. Hence, this package is automatically imported into every source file at compile time.

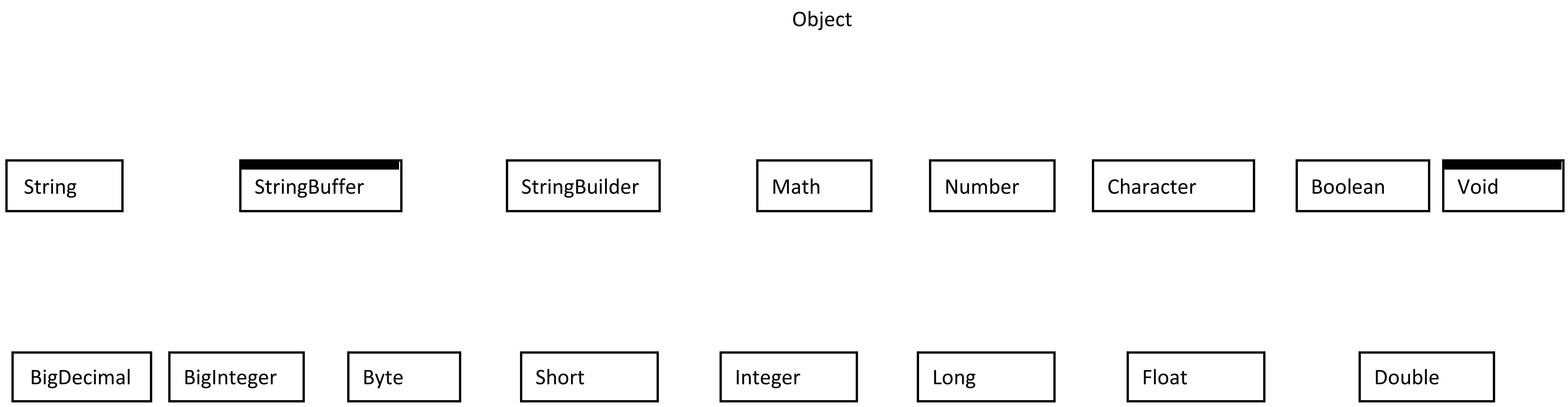


Figure: Partial Hierarchy Of java.lang Package classes

Note:

1) The six numeric wrapper classes are Byte, Short, Integer, Long, Float, Double.
2) The three non numeric wrapper classes are Character, Boolean, and Void.
3) All wrapper classes, String, StringBuffer, StringBuilder classes are final.
4) All wrapper classes and String classes are immutable.
5) All wrapper classes and String classes are Comparable and Serializable.

Object class

Every class in java, whether it is built-in or user defined, is implicitly inherited from Object class, i.e., the Object is a **root** class for all classes in java.

Object class defines following methods:

1. public String toString()
2. public boolean equals(Object)
3. public int hashCode()
4. protected Object clone()
5. public Class getClass()
6. public void wait()
7. public void notify()
8. public void notifyAll()
9. protected void finalize()

public String toString()

It is always recommended to override toString() method to provide object state information.

Example:

```
class Box{
        int  width;
        int height;
        int depth;
        Box(int w, int h, int
                d){ Width = w;
                Height = h;
                Depth = d;
        }

        Int volume(){
                return width * height * depth;
        }
        @Override
        public String toString(){
                return “Width=”+width +”  Height=”+height +” Depth=”+depth;
        }
}
Box b1 = new Box(1, 2,3);
Box b2 = new Box(4,5,6);
System.out.println(b1.toString());
System.out.println(b2);//The toString() method will be automatically invoked.
```

O/P:

```
Width=1   Height = 2    Depth = 3
Width=4   Height = 5    Depth = 6
```

Public boolean equals()

Actually, equals() method is used to compare two object states (content comparision).
The **==** operator is used to compare reference values but not object states (Reference comparision).
Example:

```
        Box b1 = new Box(1,2,3);
        Box b2 = new Box(1,2,3);
        Box b3 = b1;

        SOP(b1.equals(b2));//true
        SOP (b1 == b2); //false
        SOP(b1.equals(b3));//true
        SOP (b1 == b3); //true
        SOP (b2 == b3); //false
```

It is always recommended to override equals() method to compare two object
states. @Override

```
public boolean equals(Object o) {
        //Comparing with null reference always returns false.
        if (o == null)
                return false;
        //Comparing incompatible types always throws
        ClassCastException. if (!(o instanceof Box))
                throw new ClassCastException();
        //alias comparision.
        if (this == o) return
                true;

        Box b = (Box)o;
        //content comparison
        if ((this.width == b.width) && (this.height == b.height) && (this.depth == b.depth)) {
                return true;
        } else {
                return false;
        }
}
```

```
Box b1 = new Box(1,2,3);
Box b2 = new Box(1,2,3);
Box b3 = new Box(4,5,6);
Box b4 = b1;
Box b5 = null;
System.out.println(b1 == b2);//false
System.out.println(b1.equals(b2));//true
System.out.println(b1 == b3);//false
System.out.println(b1.equals(b3)); //false
System.out.println(b1==b4); //true
System.out.println(b1.equals(b4));//true
```

Difference between == operator and equals() method

== operator	equals() method
Used to compare reference values but not actual object states i.e., used for **Reference comparision.**	Used to compare actual object states (**Content comparision**).

public int hashCode()

The hashcode is associated with object. More than one object may have same hashcode value. The hashcode is different from reference value. The reference value is returned to reference variable, but hashcode is associated with object.

The hashing mechanism is used to **achieve better search results**.

The hashcode take us to the appropriate bucket. The equals() method is used to choose one of the object among many objects within the bucket.

```
@Override
public int hashCode() {
        final int prime =
        31; int result = 1;
        result = prime * result + depth;
        result = prime * result +
        height; result = prime * result
        + width; return result;
}

Box b1 = new Box(1,2,3);
Box b2 = new Box(4,5,6);

Box b3 = new Box(10,20,30);
Box b4 = new Box(40,50,60);
```

Contract between equals() and hashCode() methods:

1) If two objects are equal by equals() method, then their hashcodes must be same.
2) If two objects are not equal by equals() method, then their hashcodes may or may not be same.
3) If two objects hashcodes are equal by hashCode() method, then their equals() method may or may not return true.
4) If two objects hashcodes are not equal by hashCode() method, then their equals() method must return false.

Java.lang.String

In java, String is a sequence of characters but not array of characters. Once string object is created, we are not allowed to change existing object. If we are trying to perform any change, with those changes a new object is created. This behavior is called as **Immutability**.

<u>Strings are created in 4 ways:</u>

1) String Literals

    Example:

    String str = “aspire”;

2) String objects

    Example:

        String str = new String(“aspire”);

3) String constant expressions.

    Example:

        String str = “aspire ” + “ technologies”’;

4) String concatenation operation.

    Example:

        String str1 = “aspire”’;
        String str2 =” technologies”;
        String str = str1 + str2;

Example:

String str = new String(“aspire”);
str.concat(“technology”); //new string object is created.

<u>**Heap Memory**</u>

aspire

str

aspire technology

StringBuffer sb = new StringBuffer(“Aspire”);
Sb.append(“technology”);

<u>**Heap Memory**</u>

Aspire technology

sb

String class does not support **append()** method. Once we create StringBuffer object, we can perform changes. Hence it is a **mutable** object.

String Constructors

1) Public String()
   Creates an empty string whose length is zero.
2) Public String(String str)
   Example:
       String str = new String(“hello”); //string object
3) Public String(char[] value)
   Example:
       char[] values = {‘a’,’b’,’c’};
       String str = new String(values); //”abc”
4) Public String(StringBuffer sb)
   Example:
       StringBuffer sb = new StringBuffer(“hello”);
       String str = new String(sb);

5) Public String(StringBuilder sb)
   Example:
   StringBuilder builder = new StringBuilder("hello");
   String str = new String(sb);

Methods

1. **Public char charAt(int index)**
   Returns the char value at the specified index.
   Example:
   String str = "aspire";
   SOP(str.charAt(2)); // **p**
2. **Public String concat(String str)**
   Appends at the end of the invoking string. Always returns new object.
   Example:
   String str1 = ''aspire ''; //string literal
   String str2 = str1.**concat**("technologies"); //string object Sop(str1); // "aspire "
   Sop(str2);// "aspire technologies"

3. **Public String substring(int beginIndex, int endIndex)**
   Returns a new string that is a substring of this string. beginIndex is **inclusive**, but, endIndex is **exclusive**.
   Example:
   String str1 = "aspire technologies"
   String str2 = str1.substring(0, 6); //"aspire"

4. **public String substring(int beginIndex)**
   If the endIndex is not specified, it returns till end of the string.
   Example:
   String str1 = "aspire technologies";
   String str2 = str1.substring(7);//"technologies"

5. **The following lastIndexOf() methods are overloaded.**
   Public int lastIndexOf(char ch)
   Public int lastIndexOf(String str)
   Example:
   String name = "java.lang.**Integer**";
   String cName = name.substring(name.lastIndexOf('.')+1); //"Integer"
   String pName = name.substring(0,name.lastIndexOf('.')); // "java.lang"

6. **The following indexOf() methods are overloaded.**
   Public int indexOf(int ch)
   Public int indexOf(String str)
   Example:
   String pack = "**java**.lang.String";
   pack.substring(0,pack.indexOf('.')); //java

7 . **public int length()**
Returns the number of characters in the string object.
Example:
```
String str = “hello”;
SOP(str.length()); //5
```

8. **public String toLowerCase()**
Returns string in lowercase format.
Example:
```
String name=”RAMESH”;
String after = name.toLowerCase(); //ramesh
```

9. **public String toUpperCase()**
Returns string in uppercase format.
Example:
```
String name=”ramesh”;
String after = name.toUpperCase(); //RAMESH
```

10. **public String trim()**
Removes leading and trailing spaces, if any.
Example:
```
String str = “ ramesh ”
Sop(str.trim());//”ramesh”
```

11. **Public String replace(char oldChar, char newChar)**
Replaces all occurrences of old characters with new character.
Example:
```
String str = “ababab”;
String result =str.replace(‘b’,’a’); //aaaaaa
```

12. **Public int compareTo(Object obj)**
Returns –ve, if o1 < o2
Returns +ve, then o1 > o2;
Returns 0, then o1 == o2 are equal.
Example:
```
Sop(“A”.compareTo(“B”));//-1
Sop(“B”.compareTo(“A”));//+1
Sop(“A”.compareTo(“A”));//0
```

Java.lang.StringBuffer

For every change in string object, a new string object is created, because string is immutable object, which causes memory overhead. To avoid this, use StringBuffer, the changes are applied on the existing object rather than creating new object every time. Hence StringBuffer is **muttable** object.

Constructors

1) Public StringBuffer()
Constructs an empty string buffer with default initial capacity is **16.**

2) Public StringBuffer(String str)
   Constructs a string buffer initialized to the contents of the specified string.
   The initial capacity of the string buffer is : **16 + str.length()**
   Example:

```
StringBuffer  sb  =  new  StringBuffer("hello");
SOP(sb.capacity()); //21
SOP(sb.length());  //5
```

3) Public StringBuffer(int initialCapacity)
   Creates an empty string buffer with specified initial capacity.

Methods

1. Public int capacity() **//Not available in String class**
   Returns the current capacity of the StringBuffer. It means, the maximum number of characters in can hold. Once it reaches it capacity, it automatically increases.
2. Public int length()
   Returns the actual number of characters contained in the StringBuffer.
3. Public char charAt(int index)
4. The following are append() overloaded methods in StringBuffer class

```
Public StringBuffer append(String s)
Public StringBuffer append(Boolean
b) Public StringBuffer append(char s)
Public StringBuffer append(int s)
Public StringBuffer append(long s)
Public StringBuffer append(double
s) Public StringBuffer append(float s)
Public StringBuffer append(Object s)
```

   Note: Such methods are not available in String class.

5. The following are overloaded insert() methods in StringBuffer class

```
Public StringBuffer insert(int pos, String s)
Public StringBuffer insert (int pos, Boolean b)
Public StringBuffer insert (int pos, char c)
Public StringBuffer insert(int pos, int i) Public
StringBuffer insert (int pos, long l) Public
StringBuffer insert (int pos, float f) Public
StringBuffer insert(int pos, double f)
```

   Note: String does not have this method.
   Example:

```
StringBuffer sb = new StringBuffer("1221");
sb.insert(2,"33"); //123321
```

6. Public StringBuffer delete(int start, int end) //Not in string class
   Start is **inclusive**, but end is **exclusive**.
   Example:

```
StringBuffer sb = new StringBuffer("123321");
Sb.delete(2,4); //1221
```

7. Public StringBuffer deleteCharAt(int index)
8. Public StringBuffer reverse() //not in string class

Example:

```
StringBuffer sb = new StringBuffer("satyam");
System.out.println(sb.reverse()); // O/P: maytas
```

9. Public void trimToSize()
   If the capacity is larger than length, then extra space will be removed.
   Example:

```
StringBuffer sb = new
StringBuffer("hello"); SOP(sb.capacity());
//21 SOP(sb.length()); //5
Sb.trimToSize();
SOP(sb.capacity());        //5
SOP(sb.length()); //5
```

Java.lang.StringBuilder (jdk1.5)

StringBuffer is a synchronized class. It cannot be accessed by more than one thread at a time. Hence performance is not good.

StringBuilder is non-synchronized version of StringBuffer class. Hence, it can be accessed by more than one thread at a time. So, Performance is good with StringBuilder.

Differences between StringBuffer and StringBuilder:

StringBuffer	StringBuilder
Synchronized class (i.e., thread-safe)	Non-Synchronized class (i.e., non thread-safe)
Performance is low	Performance is high
Legacy class	Introduced in Jdk1.5

Wrapper classes

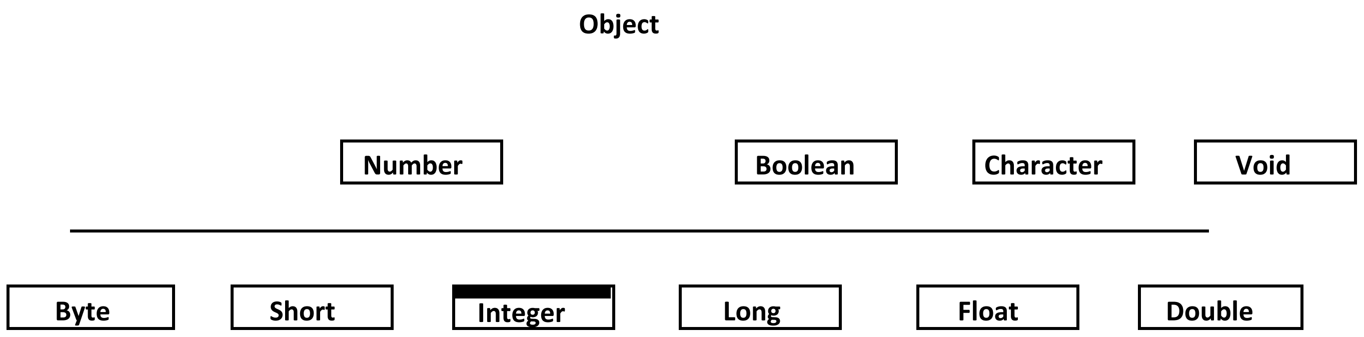


Fig: Wrapper classes

Wrapper classes are used to wrap (store) primitive data into an object.
Hence, primitive types can also be handled just like an object.
By default, all wrapper classes are final, hence cannot be inherited.

Also, the instance of all wrapper classes are immutable i.e., the value in the wrapper object cannot be changed. Public final class Integer extends Number{

```
        private final int value;
        public Integer(int value){
                this.value = value;
        }
        Public Integer(String value){
                value = parseInt(value)
        } public byte byteValue()
        {
                return (byte)value;
        }
        public short shortValue() {
                return (short)value;
        }
        public int intValue() {
                return (int)value;
        }
        public long longValue() {
                return (long)value;
        }
        public float floatValue() {
                return (float)value;
        }
         public double doubleValue() {
                return (double)value;
        }
}
```

Constructors

All wrapper classes except Character have two overloaded constructors: Primitive type and String type constructors.

Java.lang.Integer

Public Integer(int value){}
Public Integer(String value){}
Example:

Integer iRef = new Integer(4);
Integer iRef = new Integer("4");

Java.lang.Long

Public Long(long value){}
Public Long(String value){}
Example:

Long lRef = new Long(10);
Long lRef = new Long(10L);
Long lRef = new Long("10");

Java.lang.Float

Public Float(float value){}
Public Float(double value){}
Public Float(String value){}
Example:

Float fRef = new Float(3.5F);
Float fRef = new Float(3.5);
Float fRef = new Float("3.5F");

Java.lang.Character (Contains only one constructor)

Public Character(char value)
Example:

Character cRef = new Character('a');
Character cRef = new Character("a"); **//C.E**

Java.lang.Boolean

Public Boolean(boolean value)
Public Boolean(String value)
Allocates a Boolean object representing the value true if the string argument is not null and is equal, ignoring case, to the string "true".
Example:

Boolean bRef = new Boolean("true");
Boolean bRef = new Boolean("True"); //true
Boolean bRef = new Boolean("TRUE");

Boolean bRef = new Boolean(null);
Boolean bRef = new Boolean("yes"); } //false
Boolean bRef = new Boolean("no");

```
                    Boolean bRef = new Boolean("aspire");
```

Java.lang.Void

Although the Void class is considered as a wrapper class, it does not wrap any primitive value and is not instantiable i.e has no public constructors. It just denotes the Class object representing the keyword void.

Fields

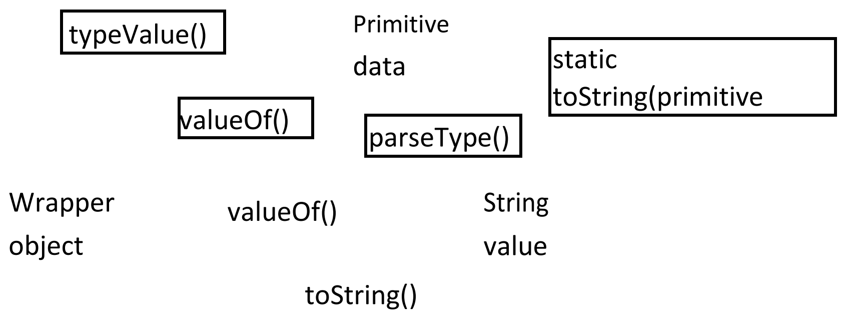
The following constants are declared for each wrapper class except Boolean:

```
Public static final primitive type MIN_VALUE
Public static final primitive type MAX_VALUE
Public static final int SIZE
Public static final Class TYPE
```

The following constants are declared only for Boolean class:

```
Public static final Boolean FALSE;
Public static final Boolean TRUE;
Public static final Class TYPE;
```

Methods



The methods are:

1. typeValue() methods → Retrieves primitive value from wrapper object.
2. parseType() methods → Converts numeric string into numeric value.
3. overloaded valueOf() → Alternative to constructors.
4. overloaded toString() → Returns string representation of primitive data.

1. <u>typeValue() methods</u>

Retrieving Primitive value from wrapper object.

a) Six Methods:

The following methods are applicable for all numeric wrapper classes:

```
public  byte  byteValue()
public short shortValue()
public int intValue()
```

All numeric wrapper classes [Byte, Short, Integer, Long, Float, Double] contains six methods.

public long longValue()
public float floatValue()
public double doubleValue()

b) booleanValue()

This method is applicable only for Boolean wrapper class.

Example:

```
Boolean bRef =new Boolean("yes");
SOP(bRef.booleanValue());  // false
```

c) charValue()

This method is applicable only for Character wrapper class.

Example:

```
Character cRef = new Character('a');
SOP(cRef.charValue()); // 'a'
```

2. parseType(String) Methods

Every wrapper class except Character class contains following static parseType() method to convert String value to Primitive data.

Wrapper class	parseType(String)	Example
Java.lang.Byte	parseByte(String)	byte b = Byte.parseByte("1"); byte b = Byte.parseByte("one"); //NFE
Java.lang.Short	parseShort(String)	short s = Short.parseShort("1");
Java.lang.Integer	parseInt(String)	Int I = Integer.parseInt("1");
Java.lang.Long	parseLong(String)	Long l = Long.parseLong("1");
Java.lang.Float	parseFloat(String)	float f = Float.parseFloat("1.0F");
Java.lang.Double	parseDouble(String)	double d = Double.parseDouble("1.0");
Java.lang.Boolean	parseBoolean(String)	Boolean b = Boolean.parseBoolean("no");

3. valueOf() method

The overloaded static valueOf() methods are **alternative to constructors** to create wrapper object.

Public static Wrapper type valueOf(primitive type)
Public static Wrapper type valueOf(String)

Example:

```
Integer iRef = Integer.valueOf(10);
Integer iRef = Integer.valueOf("10");
Integer iRef = Integer.valueOf("ten");//throws NFE
```

4. toString()

All wrapper classes overrides toString() method to return wrapper object value in string format. Also, all wrapper classes have overloaded static toString(primitive type) method to convert primitive value into string format.

Example:

```
String str = Integer.toString(10);   // "10"
```

```
Integer iRef = new Integer(10);
String str = iRef.toString(); // "10"
```

Also, the Integer, Float, Double classes contains the following xxxString() methods for converting primitive to Binary, Octal, and Hexa format.

	toBinaryString(int)	toOctalString(int)	toHexString(int / float/ double)
Integer	☑	☑	☑
Float or Double	☒	☒	☑

Example:

```
String s = Integer.toBinaryString(10); //1010
String s = Integer.toHexString(10); //a
```

Autoboxing and Unboxing

The Auto & Un boxing first time introduced in **JDK1.5**.

Autoboxing (jdk1.5)

The java compiler automatically converts Primitive data to Wrapper object is called as autoboxing.

Example:

Before compilation:

Integer iRef = 10.

After compilation:

Integer iRef = **new Integer(10)**; // Autoboxing

Unboxing

The java compiler automatically converts Wrapper object to primitive data is called as unboxing.

Example:

Before compilation:

Int I = new Integer(10);

After compilation:

Int I = **new Integer(10).intValue()**; //unboxing.

Reflection Mechanism

Reflection is the process of obtaining information about any java class i.e., getting information about fields, constructors, and methods of a java class.

Class Syntax:

<class modifiers> class <class-name> [extends <superclass name>] [implements interface$_1$, … interface$_n$] {
    //Fields
    <field modifiers> type name;
    …
    //Constructors
    <constructor modifiers> name(<parameters list>){}
    …
    //Methods
    <method modifiers> <return type> name(<parameters list>){}
}

To obtain information about classes, we have to use **java.lang.Class** and **java.lang.reflect** package.

Java.lang.Class

When the class is loaded into JVM, a class object is created in heap area. Such a class object contains complete information about specified class.

Methods:

1. public static Class **forName**(String fully qualified className) throws ClassNotFoundException
   Example:
   Class c = Class.forName("java.lang.String"); // Returns class object but not string type

2. Getting class modifiers
   Public int getModifiers();

3. Getting class name
   Public String getName() → Returns fully qualified class name.
   Public String getSimpleName() → Returns just class name without qualified with type.

4. Getting superclass information
   Public Class getSuperclass()

5. Getting interface(s) information
   Public Class[] getInterfaces()

6. Getting field(s) information
   Public **Field**[] getFields();

7. Getting constructor(s) information
   Public **Constructor**[] getConstructors()

8. Getting method(s) information

Public **Method**[] getMethods()

9. Creating new instance
   Public Object **newInstance()**

Java.lang.reflect.Field

Every field in a class must have the following format:
<field modifiers> <field type> <field name>;

Methods:

1) Getting field modifiers.
   Public int getModifiers()
2) Getting field type
   Public Class getType()
3) Getting field name
   Public String getName()

java.lang.reflect.Constructor

Every constructor in a class must have the following format:
<constructor modifier> <constructor name>(<parameters list>)

Methods:

1) Getting constructor modifier
   Public int getModifiers()

2) Getting constructor name
   Public String getName()

3) Getting constructor parameters
   Public Class[] getParameterTypes()

Java.lang.reflect.Method

Every method in a class must have the following format:
<method modifiers> <return type> <method name>(<parameters list>)

Methods:

1) Getting method modifiers
   Public int getModifiers()

2) Getting method return type
   Public Class getReturnType()

3) Getting method name
   Public String getName()

4) Getting method parameters list
   Public Class[] getParameterTypes()

Java Beans

[Reusable Software component]

Java bean is a reusable software component.
In generally, a Java Bean perform a specific task.
Some Java beans are Visible and others are Invisible.
A Java bean may contains **Properties**, **Methods**, and **Events**.

Java beans Rules:

1) Java Bean class must be declared with **public** modifier.
2) Java Bean must support **Persistence** (**Serialization)** by extending java.io.Serializable interface.
3) Java Bean must support **Introspection.**

To support introspection, Beans must provide access methods to properties. 4) Java Bean must provides Zero-argument constructors.

Example:

```
        package edu.aspire; //Recommended

        Public class Employee implements Serializable{

                Private int eno;

                Private String ename;              //Bean properties
                Private long mobile;

                Public Employee(){}

                Public void setEno(int eno){ this.eno = eno; }
                Public int getEno(){ return this.eno; }

                Public void setEname(String ename){ this.ename = ename; }

                Public String getEname(){ return ename; } //Bean property methods. Public void
                setMobile(long mobile){ this.mobile = mobile; }

                Public String getMobile() { return this.mobile;}

 }
```

Introspection Mechanism

Introspection is the process (or mechanism) of obtaining information about a Java Bean i.e., getting information about properties, events, and methods of a java bean class.
The Java bean class have following property types:

I. Simple properties
II. Boolean properties
III. Indexed properties

Naming Conventions
The following table detailing java bean class method naming conventions for every property type:

Property type	Naming patterns	Example

Simple	Public T getN()
	Public void setN(T value)
		Public String getName(){}
		Public void setName(String name){}

Boolean	Public boolean isN()
	Public boolean getN()
	Public void setN(boolean value)
		Public boolean isHoliday(){}
		Public boolean getHoliday(){}
		Public void setHoliday(boolean value){}

Indexed	Public T getN(int index)
	Public T[] getN()
	Public void setN(int index, T value)
	Public void setN(T[] value){}
		Public boolean getInputs(int index){}
		Public boolean[] getInputs()
		Public void setInputs(int index, boolean value){}
		Public void setInputs(boolean[] values){}

Introspection mechanism uses **java.beans** package from JDK API to analyze Java Beans.

Java.beans.Introspector

It contains static methods that allow us to obtain information about properties, events, and methods of a bean. Public static BeanInfo getBeanInfo(Class beanClass);
The above static method returns BeanInfo object which contains complete Java Bean information.

Java.beans.BeanInfo<<interface>>

The following methods from BeanInfo object gives information about Properties, Events, and Methods of a Java Bean class.

1) Public PropertyDescriptor[] **getPropertyDescriptors**()
2) Public MethodDescriptor[] **getMethodDescriptors**()
3) Public EventSetDescriptor[] **getEventSetDescriptors**()

Example:

```
import java.awt.Button;
import java.beans.BeanInfo;
import java.beans.EventSetDescriptor;
import java.beans.IntrospectionException;
import java.beans.Introspector;
import  java.beans.MethodDescriptor;
import java.beans.PropertyDescriptor;
public class ItrospectionEx1 {
        public static void main(String[] args) {
                try {
                        BeanInfo beanInfo = Introspector.getBeanInfo(Button.class);
                        System.out.println("   ---------------------");
                        System.out.println("Java Bean Properties:");
                    System.out.println("--------------------            ");
                    PropertyDescriptor[] props = beanInfo.getPropertyDescriptors();
                    for (PropertyDescriptor prop : props) {
                        System.out.println(prop.getPropertyType().getSimpleName() + " "
                                + prop.getName());
                    }
                    System.out.println("------------------         ");
                    System.out.println("Java Bean Methods:");
                    System.out.println("------------------         ");
                    MethodDescriptor[] methods = beanInfo.getMethodDescriptors();
                    for (MethodDescriptor method : methods) {
                        System.out.println(method.getName());
                    }
                    System.out.println("-----------------       ");
                    System.out.println("Java Bean Events:");
                    System.out.println("-----------------       ");
                    EventSetDescriptor[] events = beanInfo.getEventSetDescriptors();
                    for(EventSetDescriptor event : events){
                        System.out.println(event.getName());
                    }
                } catch (IntrospectionException e) {
                    e.printStackTrace();
                }
        }
}
```

Conclusion:

1) The Reflection mechanism work at a low level and deal only with fields, constructors, and methods of a class. The Introspection mechanism work at a higher level and deal with the properties, events, and methods of a Java Bean.
2) Introspection mechanism internally uses Reflection API to retrieve properties, events, or methods information.