# Unit-2

INHERITANCE ,JAVA PACKAGES

## UNIVERSAL CLASS (OBJECT CLASS)

- Object class (in java.lang) is the root of the Java class hierarchy. Every class in Java either directly or indirectly extends Object.
- It provides essential methods like toString(), equals(), hashCode(), clone() and several others that support object comparison, hashing, debugging, cloning and synchronization.

- **1. toString() Method**
- toString() provides a String representation of an object and is used to convert an object to a String.

```
class Student{

    String name = "Vishnu";
    int age = 21;


    public String toString(){

        return "Student{name='" + name + "', age=" + age + "}";
    }

    public static void main(String[] args) {
        Student s = new Student();

       System.out.println(s.toString());
    }
}
```

- **2. hashCode() Method**
- hashCode() method returns the hash value of an object (not its memory address). Used heavily in hash-based collections like HashMap, HashSet, etc.
-

```
class Employee{

    int id = 101;


    public int hashCode(){

        return id * 51;
    }

    public static void main(String[] args) {
        Employee e = new Employee();
        System.out.println(e.hashCode());
    }
}
```

- **equals(Object obj) Method**
- equals() method compares the given object with the current object. It is recommended to override this method to define custom equality conditions.

```
• class Book{
•
•    String title;

•    Book(String title) {
•       this.title = title;
•    }

•    public boolean equals(Object obj){
•
•       Book b = (Book) obj;
•       return this.title.equals(b.title);
•    }

•    public static void main(String[] args) {
•       Book b1 = new Book("Java");
•       Book b2 = new Book("Java");
•       System.out.println(b1.equals(b2));
•    }
• }
```

## Access Specifiers (public, private, protected, default, private protected)

- **Private Access Modifier**
- The private access modifier is specified using the keyword private. The methods or data members declared as private are accessible only within the class in which they are declared.

```
class Person {

    private String name;
    public void setName(String name)  {

        this.name = name;
    }
    public String getName() { return name; }
}

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

        Person p = new Person();
        p.setName("Alice");
        System.out.println(p.getName());
    }
}
```

- **Default Access Modifier**
- When no access modifier is specified for a class, method, or data member, it is said to have the default access modifier by default. This means only classes within the same package can access it.

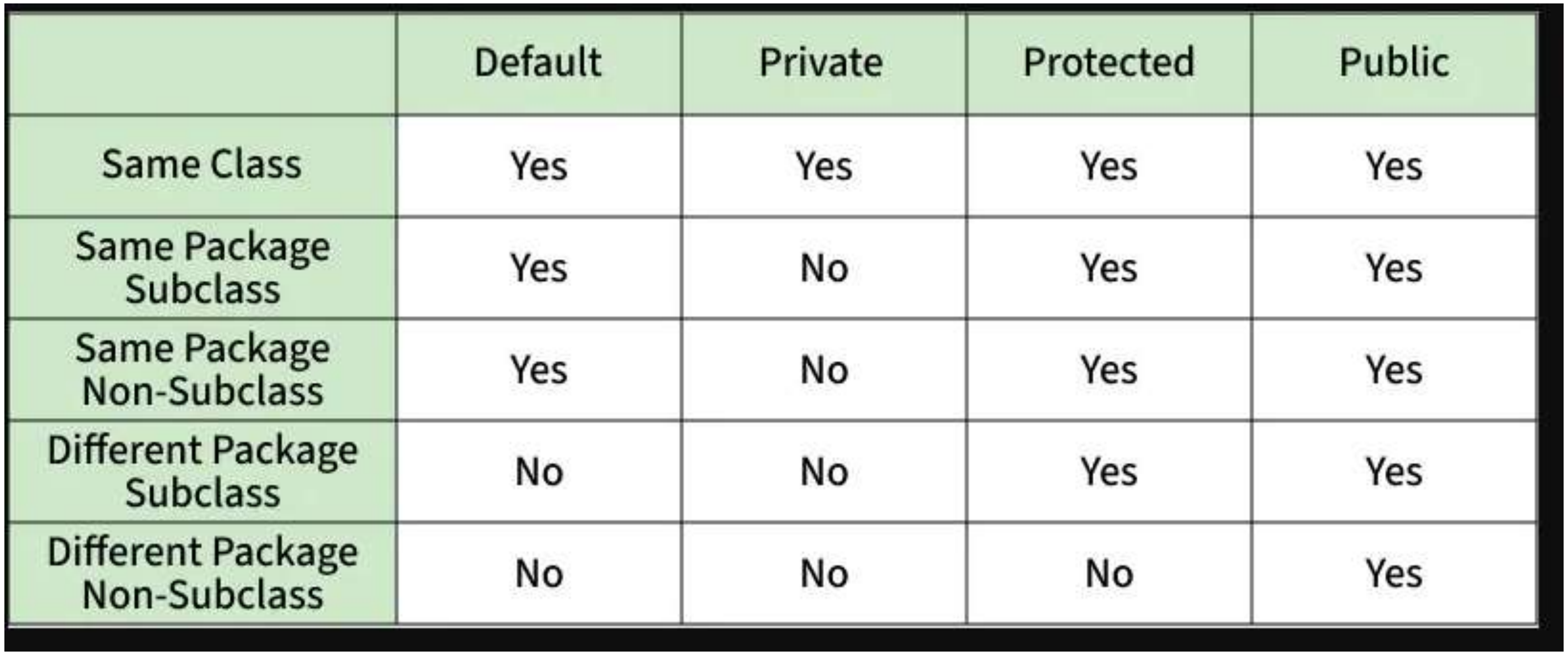
```
class Car {
    String model;
}

public class Main {

    public static void main(String[] args){

        Car c = new Car();
        c.model = "Tesla";
        System.out.println(c.model);
    }
}
```

- **Protected Access Modifier**
- The protected access modifier is specified using the keyword protected. The methods or data members declared as protected are accessible within the same package or subclasses in different packages.

```
class Vehicle {
    protected int speed;
}

class Bike extends Vehicle {
    void setSpeed(int s)
    {
        speed = s;
    }

    int getSpeed()
    {
        return speed;
    }
}

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

        Bike b = new Bike();
        b.setSpeed(100);
        System.out.println("Access via subclass method: "
                           + b.getSpeed());

        Vehicle v = new Vehicle();
        System.out.println(v.speed);
    }
}
```

- **Public Access Modifier**
- The public access modifier is specified using the keyword public. Public members are accessible from everywhere in the program. There is no restriction on the scope of public data members.
-

```java
class MathUtils {

    public static int add(int a, int b) {
        return a + b;
    }
}

public class Main {

    public static void main(String[] args) {

        System.out.println(MathUtils.add(5, 10));
    }
}
```



| | Default | Private | Protected | Public |
|---|---|---|---|---|
| Same Class | Yes | Yes | Yes | Yes |
| Same Package Subclass | Yes | No | Yes | Yes |
| Same Package Non-Subclass | Yes | No | Yes | Yes |
| Different Package Subclass | No | No | Yes | Yes |
| Different Package Non-Subclass | No | No | No | Yes |

## Constructor in Inheritance

- The sequence in which the constructors are invoked is from super class to sub class i.e, first the constructor of class base is executed then constructor of class drived is executed .
- The sequence of constructor invocation does not change even when *super* keyword is used. Now let’s consider a few examples to understand how constructors in inheritance works.
-

```
class A
{
        A()
        {
                System.out.println("Class A's constructor is invoked");
        }
}

class B extends A
{
        B()
        {
                System.out.println("Class B's constructor is invoked");
        }
}

public class Driver
{
        public static void main(String[] args)
        {
                A obj = new A();
        }
}
```

- In the above example, object for class A is created. Since A is the super class only it’s constructor is invoked and the output will be:
- Class A’s constructor is invoked

```
class A
{
        A()
        {
                System.out.println("Class A's constructor is invoked");
        }
}

class B extends A
{
        B()
        {
                System.out.println("Class B's constructor is invoked");
        }
}

public class Driver
{
        public static void main(String[] args)
        {
                B obj = new B();
        }
}
```

- In the above example, we are creating object for the sub class B. Since B is a sub class, constructor of class A is invoked first and then constructor of class B is invoked. Output for above program is:
- Class A’s constructor is invoked
  Class B’s constructor is invoked
-

```
class A
{
        A()
        {
                System.out.println("Class A's constructor is invoked");
        }
}

class B extends A
{
        B()
        {
                System.out.println("Class B's constructor is invoked");
        }
}

class C extends B
{
        C()
        {
                System.out.println("Class C's constructor is invoked");
        }
}

public class Driver
{
        public static void main(String[] args)
        {
                C obj = new C();
        }
}
```

- In the above we can see multi-level inheritance A <- B <- C. The object we are creating is for sub class C. So the sequence in which constructors are executed is constructor of class A followed by class B followed by class C. Output of the above program is:
- Class A’s constructor is invoked
  Class B’s constructor is invoked
  Class C’s constructor is invoked

## Method Overriding

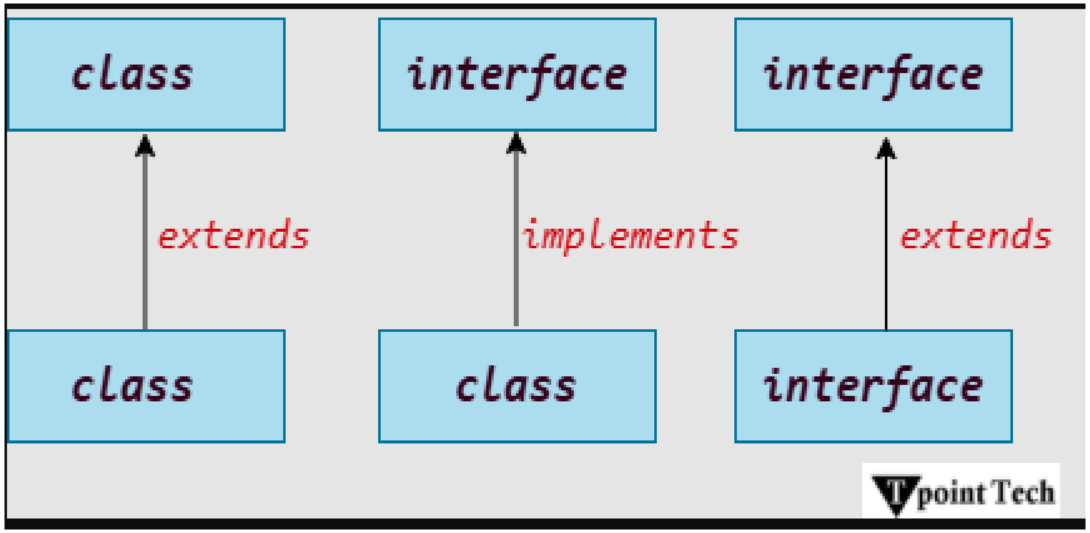
- Method Overriding allows a subclass to provide its own specific implementation of a method that is already defined in its parent class.
- Method overriding supports runtime polymorphism by enabling dynamic method binding, where the method call is resolved at runtime based on the actual object type.

```
class Vehicle{

  void run(){System.out.println("Vehicle is running");}
}

class Bike extends Vehicle{

  void run(){System.out.println("Bike is running safely");}
}

public class Main{
  public static void main(String args[]){
    Vehicle v=new Vehicle();
     v.run();
    Bike obj = new Bike();
    obj.run();
  }
}
```

## Interface

- An **interface** in Java is used to define a common set of methods that a class must implement.
- An interface helps in achieving abstraction and supports multiple inheritance. In this chapter, we will learn how to define an interface, how to achieve abstraction and multiple inheritance using interfaces.

- Syntax of Interface Declaration
- Here is the syntax to declare an interface:
- **interface** <interface_name> {
-     *// declare constant fields*
-     *// declare abstract methods*
- }

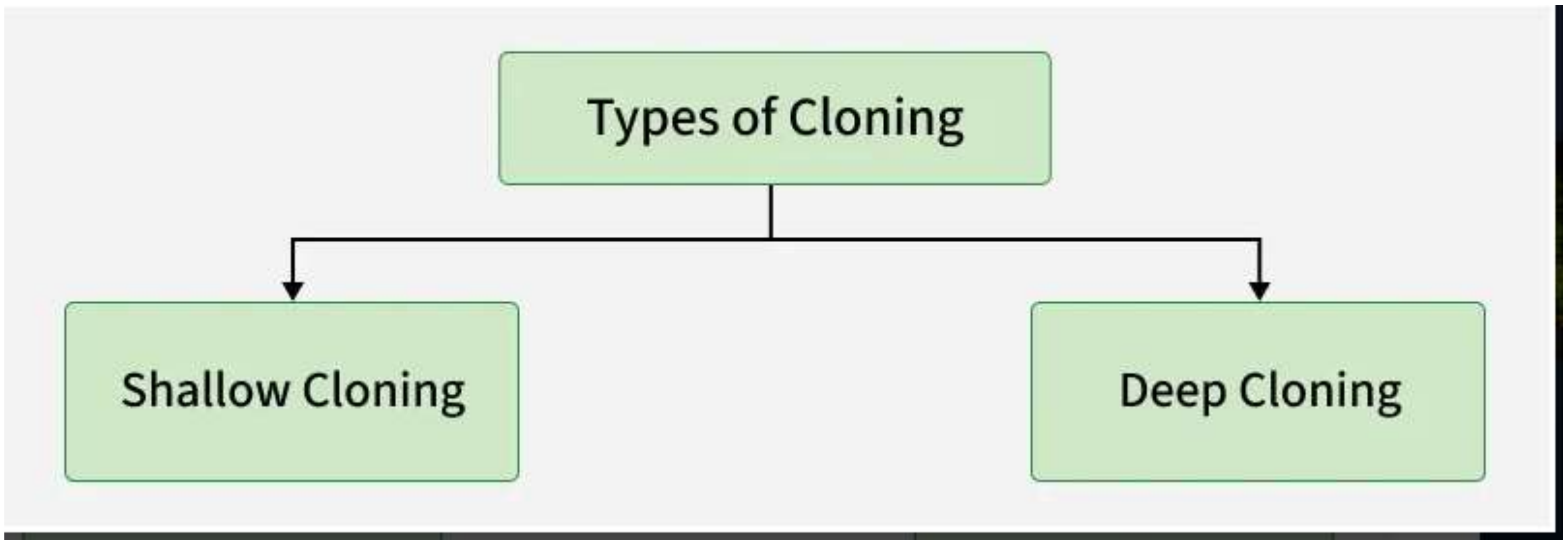

class
interface
interface
extends
implements
extends
class
class
interface
point Tech

```java
interface Printable{
  void print();
}
class Printer implements Printable{
  public void print(){System.out.println("Hello");}
}
public class Main{
  public static void main(String args[]){
    Printable p=new Printer();
    p.print();
 }
}
```

```
interface Bank{
  float rateOfInterest();
}
class SBI implements Bank{
  public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
  public float rateOfInterest(){return 9.7f;}
}
class HDFC implements Bank{
  public float rateOfInterest(){return 8.7f;}
}

public class Main{
  public static void main(String[] args){
    Bank b;
    b=new SBI();
    System.out.println("SBI ROI: "+b.rateOfInterest());
    b=new PNB();
    System.out.println("PNB ROI: "+b.rateOfInterest());
    b=new HDFC();
    System.out.println("HDFC ROI: "+b.rateOfInterest());
  }
}
```

## OBJECT CLONING

- **Object cloning in Java** refers to creating an exact copy of an object. The **clone() method in Java** is used to clone an object. It creates a new instance of the class of the current object and initializes all its fields with exactly the contents of the corresponding fields of this object.


Types of Cloning
Shallow Cloning
Deep Cloning

## SHALLOW CLONING

- Shallow cloning in Java creates a new object, but the new object that is created has the same reference as the original object, so instead of creating an actual copy, it involves copying the references.
- Shallow cloning is done by calling super.clone() method as default implementation of the clone is present in the Object class.

- SHALLOW OBJECT CLONE

```
class abc{

        int i;
        int j;
        public String toString(){
                return  "abc  i=="+ i+ "abc   j=="+ j;

        }

}
class clo{

        public static void main(String[] arg){
                abc o=new abc();
                o.i=5;
                o.j=6;


                abc o1=o;
                o1.j=9;
                System.out.println(o);
                System.out.println(o1);


        }
}
```

## DEEP CLONING

- Deep cloning creates a copy of an object along with all the objects it references, ensuring complete independence from the original object. Unlike shallow cloning, changes in the original object's referenced objects do not affect the cloned object.

```
class abc{

        int i;
        int j;
        public String toString(){
                return  "abc  i=="+ i+ "abc   j=="+ j;

        }

}
class clo{

        public static void main(String[] arg){
                abc o=new abc();
                o.i=5;
                o.j=6;
                abc o1=new abc();
                o1.i=o.i;
                o1.j=o.j;
                o1.j=10;
                System.out.println(o);
                System.out.println(o1);


        }
}
```

## Clone() |Method

- **Object cloning in** Java refers to creating an exact copy of an object. The **clone() method in Java** is used to clone an object. It creates a new instance of the class of the current object and initializes all its fields with exactly the contents of the corresponding fields of this object.

```
class abc implements Cloneable {
   int x, y;

   public Object clone() throws CloneNotSupportedException {

      return super.clone();
   }
}

public class clo {

   public static void main(String[] args)
     throws CloneNotSupportedException {
      abc o1 = new abc();
                        o1.x=200;
                        o1.y=900;
      abc o2 = (abc) o1.clone();
                        o2.x=800;
      System.out.println("o1: " + o1.x + " " + o1.y);
      System.out.println("o2: " + o2.x + " " + o2.y);
   }
}
```

## Nested and inner class

- In Java, it is possible to define a class within another class, such classes are known as *nested* classes. They enable you to logically group classes that are only used in one place, thus this increases the use of encapsulation and creates more readable and maintainable code.

- Nested classes are divided into two categories:
  - **static nested class:** Nested classes that are declared *static* are called static nested classes.
  - **inner class:** An inner class is a non-static nested class.

- **Static nested classes**
- In the case of normal or regular inner classes, without an outer class object existing, there cannot be an inner class object. i.e., an object of the inner class is always strongly associated with an outer class object.

## Syntax of object

- OuterClass. nested   O= new OuterClass.nested();

```
class OuterClass {

    static int outer_x = 10;
    int outer_y = 20;
    private static int outer_private = 30;
    static class StaticNestedClass {
        void display()
        {

            System.out.println("outer_x = " + outer_x);
            System.out.println("outer_private = "
                                + outer_private);
             // OuterClass out = new OuterClass();
              System.out.println("outer_y = " + out.outer_y);


        }
    }
}
```

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

        OuterClass.StaticNestedClass nestedObject
            = new OuterClass.StaticNestedClass();
        nestedObject.display();
    }
}
```

- **Inner classes**
- To instantiate an inner class, you must first instantiate the outer class. Then, create the inner object within the outer object with this
- syntax:
- OuterClass outerObject=new OuterClass();
- OuterClass.InnerClass   i = outerObject.new InnerClass();

```
class OuterClass {
    static int outer_x = 10;

    int outer_y = 20;

    private int outer_private = 30;

    class InnerClass {
        void display()
        {
            System.out.println("outer_x = " + outer_x);

            System.out.println("outer_y = " + outer_y);

            System.out.println("outer_private = "
                               + outer_private);
        }
    }
}

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

        OuterClass outerObject = new OuterClass();

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

        innerObject.display();
    }
}
```

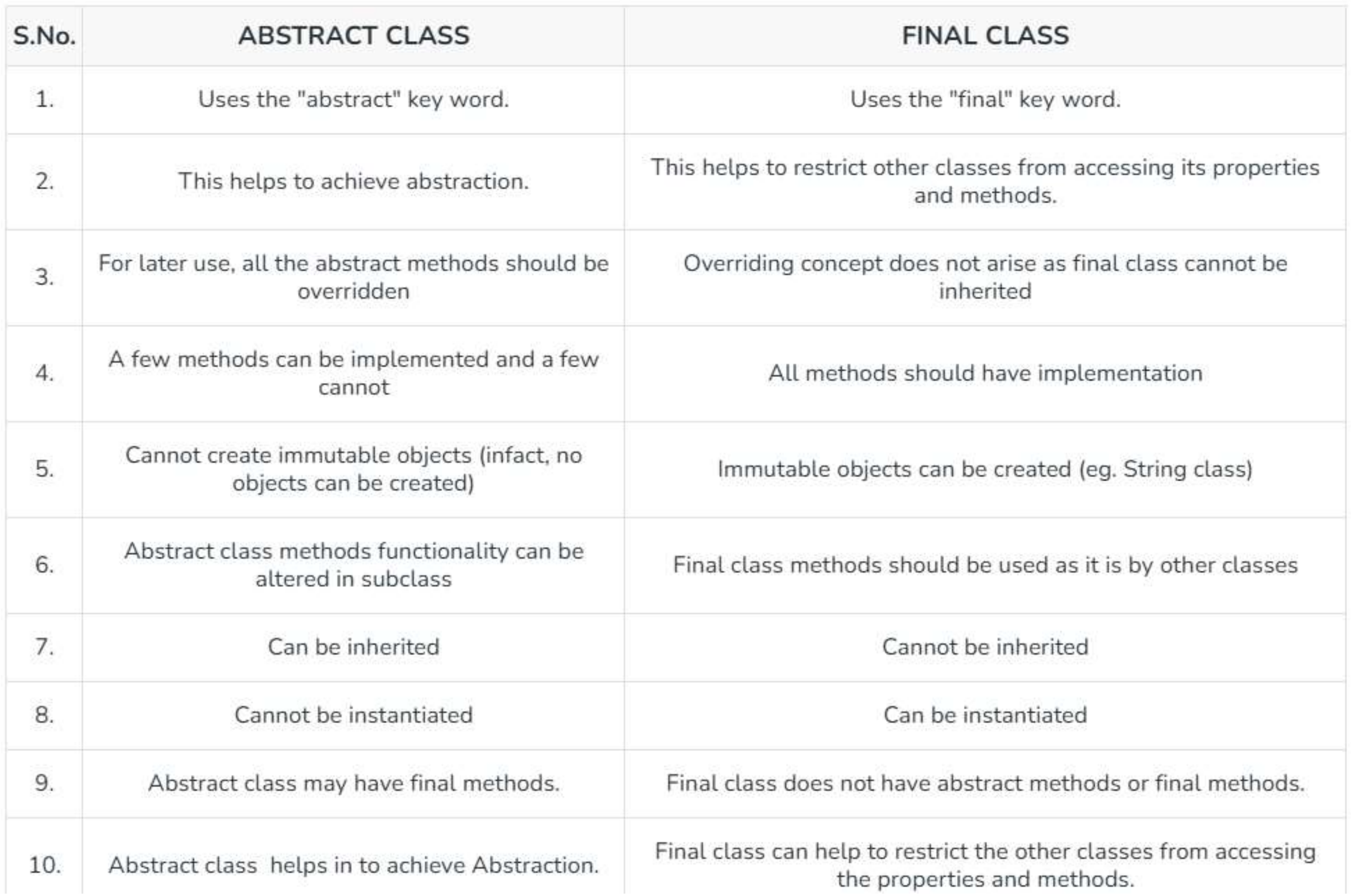
## Abstract and Final Class

- **Final Class:** A class which is declared with the "Final" keyword is known as the final class
- The final keyword is used to finalize the implementations of the classes, the methods and the variables used in this class.
- The main purpose of using a final class is to prevent the class from being inherited (i.e.) if a class is marked as final, then no other class can inherit any properties or methods from the final class. If the final class is extended, Java gives a compile-time error

```
• final class Super {
•     int data = 100;
• }
• public class Sub extends Super {
•     public static void main(String args[])
•     {

•     }
• }
```

- **Abstract Class:** A class that is declared using the "abstract" keyword is known as an abstract class.
- The main idea behind an abstract class is to implement the concept of Abstraction.
- An abstract class can have both abstract methods(methods without body) as well as the concrete methods(regular methods with the body).
- However, a normal class(non-abstract class) cannot have abstract methods. The following is an example of how an abstract class is declared.

```
abstract class Book {

   // Abstract method without body
   public abstract void page();
}


public class s extends Book {

   // Declaring the abstract method
   public void page()
   {
      System.out.println("OLD");
   }

   // Driver code
   public static void main(String args[])
   {
      Book obj = new s();
      obj.page();
   }
}
```



| S.No. | ABSTRACT CLASS | FINAL CLASS |
|---|---|---|
| 1. | Uses the "abstract" key word. | Uses the "final" key word. |
| 2. | This helps to achieve abstraction. | This helps to restrict other classes from accessing its properties and methods. |
| 3. | For later use, all the abstract methods should be overridden | Overriding concept does not arise as final class cannot be inherited |
| 4. | A few methods can be implemented and a few cannot | All methods should have implementation |
| 5. | Cannot create immutable objects (infact, no objects can be created) | Immutable objects can be created (eg. String class) |
| 6. | Abstract class methods functionality can be altered in subclass | Final class methods should be used as it is by other classes |
| 7. | Can be inherited | Cannot be inherited |
| 8. | Cannot be instantiated | Can be instantiated |
| 9. | Abstract class may have final methods. | Final class does not have abstract methods or final methods. |
| 10. | Abstract class  helps in to achieve Abstraction. | Final class can help to restrict the other classes from accessing the properties and methods. |

## Normal import and Static Import

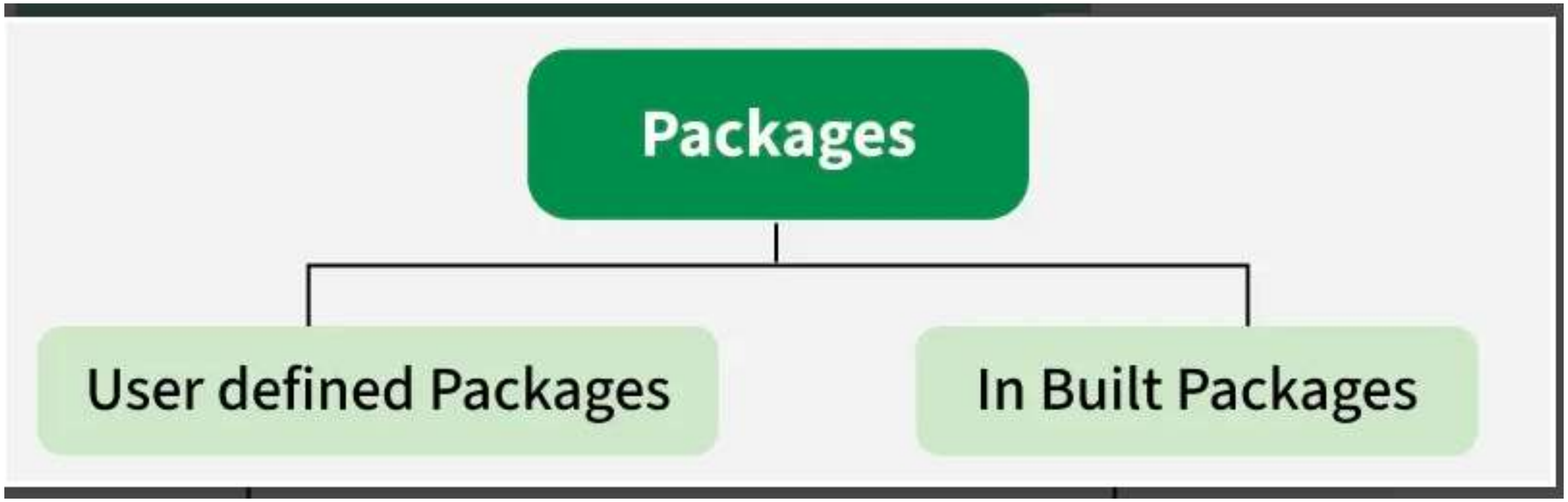
- In Java, static import concept is introduced in 1.5 version. With the help of static import, we can access the static members of a class directly without class name or any object

## NORMAL IMPORT

```
import java.lang.*;
class a {
    public static void main(String[] args)
    {
        System.out.println(Math.sqrt(4));
        System.out.println(Math.pow(2, 2));
        System.out.println(Math.abs(6.3));
    }
}
```

## Java packages

- A package in Java is a mechanism to group related classes, interfaces, and sub-packages into a single unit. Packages help organize large applications, avoid naming conflicts, provide access protection, and make code modular and maintainable.
- Avoiding name conflicts (two classes with the same name can exist in different packages)
- Providing access control using public, protected, and default access
- Reusability: packaged code can be imported and used anywhere
- Encouraging modular programming

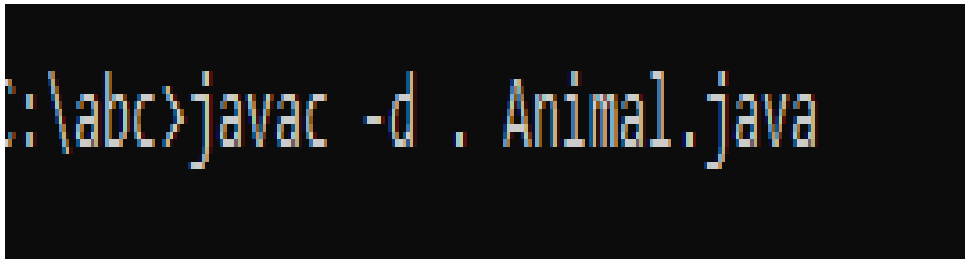

Packages
User defined Packages
In Built Packages

## User defined packages

- User-defined packages are those packages that are designed or created by the developer to categorize classes and packages. It can be imported into other classes and used in the same way as we use built-in packages.

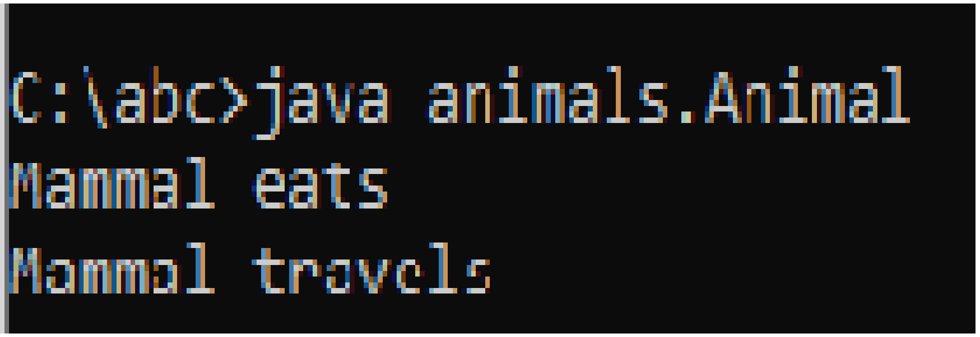
```
package animals;

public class  Animal {
   public void eat() {
      System.out.println("Mammal eats");
   }
   public void travel() {
      System.out.println("Mammal travels");
   }

   public static void main(String args[]) {
     Animal m = new Animal();
      m.eat();
      m.travel();
   }
}
```

## how to compile



```
:\abc>javac -d . Animal.java
```

## how to run



```
C:\abc>java animals.Animal
Mammal eats
Mammal travels
```

## Buit-in package

- java.lang,
- java.util
- java.io,
- java.net
- java.awt,
- java.awt.event
- java.applet,
- java.swing

## java.lang Package Classes (Math, Wrapper Classes, String, StringBuffer)

- **Java Math Class**
- Java.lang.Math Class methods help to perform numeric operations like square, square root, cube, cube root, exponential and trigonometric operations.
- **Methods of Math Class in Java**
- 1.abs()        Return the absolute value
- 2.max()      Returns the maximum out of the two arguments

- 3.min()

Returns the minimum out of the two arguments.

- 4.pow()

Returns(pow(a,b)) the value of ab.

- 5. round()

Returns the closest int to the argument, with ties rounding to positive infinity

```
• public class MathLibraryExample {
•     public static void main(String[] args) {
•         int i = 7;
•         int j = -9;
•         double x = 72.3;
•         double y = 0.34;
•         System.out.println("i is " + i);
•         System.out.println("j is " + j);
•         System.out.println("|" + i + "| is " + Math.abs(i));
•         System.out.println("|" + j + "| is " + Math.abs(j));
•         System.out.println(x +"Round==="+Math.round(x));
•         System.out.println(y +"Round==="+Math.round(y));
•         System.out.println("min(" + i + "," + j + ") is " + Math.min(i, j));
•         System.out.println("min(" + x + "," + y + ") is " + Math.min(x, y));
•         System.out.println("max(" + i + "," + j + ") is " + Math.max(i, j));
•         System.out.println("max(" + x + "," + y + ") is " + Math.max(x, y));
•         System.out.println("pow(2, 2) is " + Math.pow(2, 2));
•         System.out.println("pow(10.0, 3) is " + Math.pow(10.0, 3));
•         System.out.println("pow(-3, 2) is " + Math.pow(-3, 2));
•         for (i = 0; i < 10; i++) {
•             System.out.println("The square root of " + i + " is " + Math.sqrt(i));
•         }

•
•     }
• }
```

## Wrapper class

- Wrapper classes provide a way to use primitive data types (int, boolean, etc..) as objects.
- The object of the wrapper class contains or wraps its respective primitive data type.
- Converting primitive data types into object is called **boxing**, and this is taken care by the compiler.
- Therefore, while using a wrapper class you just need to pass the value of the primitive data type to the constructor of the Wrapper class.
- The table below shows the primitive type and the equivalent wrapper class:

| Primitive Data Type | Wrapper Class |
|---|---|
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |
| boolean | Boolean |
| char | Character |

```
Import java.lang.Wrapper.*;
public class Main {
public static void main(String[] args) {
        Integer myInt = 5;//myint is object,boxing
        myInt =myInt*10;//unboxing
        int a=10;
        Double myDouble = 5.99;
        Character myChar = 'A';
        System.out.println(myInt);
        System.out.println(a);//PRIMITIVE DATATYPE
        System.out.println(myDouble);
        System.out.println(myChar);
}
 }
```

## String class

- The String class has a set of built-in methods that you can use on strings.
- charAt()  Returns the character at the specified index (position)
- concat()Appends a string to the end of another string
- equals()Compares two strings. Returns true if the strings are equal, and false if not

- contains()Checks whether a string contains a sequence of characters
- endsWith()Checks whether a string ends with the specified character.

```
public class Main {
  public static void main(String[] args) {
    String mySt = "Hello";
    char result = mySt.charAt(0);
    System.out.println(result);
            String firstName = "John ";
    String lastName = "Doe";
    System.out.println(firstName.concat(lastName));
            String myStr1 = "Hello";
    String myStr2 = "Hello";
    String myStr3 = "Another String";
    System.out.println(myStr1.equals(myStr2));
    System.out.println(myStr1.equals(myStr3));

            String myStr = "Hello";
    System.out.println(myStr.contains("Hel"));
    System.out.println(myStr.contains("e"));
    System.out.println(myStr.contains("Hi"));
             String myStr4 = "Hello";
    System.out.println(myStr.endsWith("Hel"));
    System.out.println(myStr.endsWith("llo"));
    System.out.println(myStr.endsWith("o"));
  }
  }
```

## String Buffer

- StringBuffer class in Java represents a sequence of characters that can be modified, which means we can change the content of the StringBuffer without creating a new object every time. It represents a mutable sequence of characters.
- Unlike String, we can modify the content of the StringBuffer without creating a new object.
- All methods of StringBuffer are synchronized, making it safe to use in multithreaded environments.
- Ideal for scenarios with frequent modifications like append, insert, delete or replace operations.

- **StringBuffer():** It reserves room for 16 characters without reallocation
- **StringBuffer(int size):** It accepts an integer argument that explicitly sets the size of the buffer.
- **StringBuffer(String str):** It accepts a string argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation

```
import java.lang.*;
 class sbj {
    public static void main(String[] args) {

        // 1. Using default constructor
        StringBuffer sb1 = new StringBuffer();
        sb1.append("Hello");
        System.out.println("Default Constructor: " + sb1);

        // 2. Using constructor with specified capacity
        StringBuffer sb2 = new StringBuffer(70);
        sb2.append("Java Programming");
        System.out.println("With Capacity 70: " + sb1);

       // 3. Using constructor with String
       StringBuffer sb3 = new StringBuffer("Welcome");
        sb3.append(" to Java");
       System.out.println("With String: " + sb3);
    }
}
```