Object class in Java

Object class is present in java.lang package. Every class in Java is directly or indirectly derived from the Object class. If a Class does not extend any other class then it is direct child class of Object and if extends other class then it is an indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.

1. toString() : toString() provides String representation of an Object and used to convert an object to String. The default toString() method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object.
   It is always recommended to override toString() method to get our own String representation of Object.
   Note : Whenever we try to print any Object reference, then internally toString() method is called.

2 hashCode() : For every object, JVM generates a unique number which is hashcode. It returns distinct integers for distinct objects. A common misconception about this method is that hashCode() method returns the address of object, which is not correct. It convert the internal address of object to an integer by using an algorithm.

```
// Java program to demonstrate working of

// hasCode() and toString()

public class Student

{

        static int last_roll = 100;

        int roll_no;



        // Constructor

        Student()

        {

                roll_no = last_roll;

                last_roll++;

        }
        // Overriding hashCode()
        @Override
        public int hashCode()
        {
                return roll_no;
        }

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

                // Below two statements are equivalent
                System.out.println(s);
                System.out.println(s.toString());
        }
}
```

2. equals(Object obj) : Compares the given object to "this" object (the object on which the method is called). It gives a generic way to compare objects for equality. It is recommended to override equals(Object obj) method to get our own equality condition on Objects.

Java Abstract Classes and Methods

Data abstraction is the process of hiding certain details and showing only essential information to the user.
Abstraction can be achieved with either abstract classes or interfaces

The `abstract` keyword is a non-access modifier, used for classes and methods:

- Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

- Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

An abstract class can have both abstract and regular methods:

```
abstract class Animal {

  public abstract void animalSound();

  public void sleep() {

    System.out.println("Zzz");

  }

}
```

From the example above, it is not possible to create an object of the Animal class:

```
Animal myObj = new Animal(); // will generate an error
```

To access the abstract class, it must be inherited from another class.

```
// Abstract class

abstract class Animal {

  // Abstract method (does not have a body)

  public abstract void animalSound();

  // Regular method

  public void sleep() {
    System.out.println("Zzz");
  }
}

// Subclass (inherit from Animal)
class Pig extends Animal {
  public void animalSound() {
    // The body of animalSound() is provided here
    System.out.println("The pig says: wee wee");
  }
}

class MyMainClass {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Why And When To Use Abstract Classes and Methods?

To achieve security - hide certain details and only show the important details of an object.

Note: Abstraction can also be achieved with Interfaces

Java Interface

Another way to achieve abstraction in Java, is with interfaces.

An `interface` is a completely "abstract class" that is used to group related methods with empty bodies:

Example

```
// interface

interface Animal {

  public void animalSound(); // interface method (does not have a body)

  public void run(); // interface method (does not have a body)

}
```

To access the interface methods, the interface must be "implemented" (kinda like inherited) by another class with the `implements` keyword (instead of `extends`). The body of the interface method is provided by the "implement" class:

Example

```
// Interface

interface Animal {

  public void animalSound(); // interface method (does not have a body)

  public void sleep(); // interface method (does not have a body)

}



// Pig "implements" the Animal interface

class Pig implements Animal {

  public void animalSound() {

    // The body of animalSound() is provided here

    System.out.println("The pig says: wee wee");
  }

  public void sleep() {

    // The body of sleep() is provided here

    System.out.println("Zzz");

  }

}



class MyMainClass {

  public static void main(String[] args) {

    Pig myPig = new Pig();  // Create a Pig object

    myPig.animalSound();

    myPig.sleep();

  }

}
```

Notes on Interfaces:

- Like abstract classes, interfaces cannot be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
- Interface methods do not have a body - the body is provided by the "implement" class
- On implementation of an interface, you must override all of its methods
- Interface methods are by default `abstract` and `public`
- Interface attributes are by default `public`, `static` and `final`
- An interface cannot contain a constructor (as it cannot be used to create objects)

Why And When To Use Interfaces?

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can implement multiple interfaces. Note: To implement multiple interfaces, separate them with a comma

```java
interface FirstInterface {

  public void myMethod(); // interface method

}


interface SecondInterface {

  public void myOtherMethod(); // interface method

}


class DemoClass implements FirstInterface, SecondInterface {

  public void myMethod() {

    System.out.println("Some text..");

  }

  public void myOtherMethod() {

    System.out.println("Some other text...");

  }

}


class MyMainClass {

  public static void main(String[] args) {

    DemoClass myObj = new DemoClass();

    myObj.myMethod();

    myObj.myOtherMethod();
  }

}
```

Java Packages & API

A package in Java is used to group related classes. Think of it as a folder in a file directory. We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:

- Built-in Packages (packages from the Java API)
- User-defined Packages (create your own packages)

Built-in Packages

The Java API is a library of prewritten classes, that are free to use, included in the Java Development Environment.

The library contains components for managing input, database programming, and much much more. The complete list can be found at Oracles website: https://docs.oracle.com/javase/8/docs/api/.

The library is divided into packages and classes. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

To use a class or a package from the library, you need to use the `import` keyword:

Syntax

```
import package.name.Class;   // Import a single class

import package.name.*;   // Import the whole package
```

Import a Class

If you find a class you want to use, for example, the `Scanner` class, which is used to get user input, write the following code:

Example

```
import java.util.Scanner;
```

In the example above, `java.util` is a package, while `Scanner` is a class of the `java.util` package.

To use the `Scanner` class, create an object of the class and use any of the available methods found in the `Scanner` class documentation. In our example, we will use the `nextLine()` method, which is used to read a complete line:

Import a Package

There are many packages to choose from. In the previous example, we used the `Scanner` class from the `java.util` package. This package also contains date and time facilities, random-number generator and other utility classes.

To import a whole package, end the sentence with an asterisk sign (`*`). The following example will import ALL the classes in the `java.util` package:

Example

```
import java.util.*
```

User-defined Packages

To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer:

Example

```
└── root
  └── mypack
    └── MyPackageClass.java
```

To create a package, use the `package` keyword:

MyPackageClass.java

```
package mypack;

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

    System.out.println("This is my package!");

  }

}
```

Save the file as MyPackageClass.java, and compile it:

```
C:\Users\Your Name>javac MyPackageClass.java
```

Then compile the package:

```
C:\Users\Your Name>javac -d . MyPackageClass.java
```

This forces the compiler to create the "mypack" package.

The `-d` keyword specifies the destination for where to save the class file. You can use any directory name, like c:/user (windows), or, if you want to keep the package within the same directory, you can use the dot sign "`.`", like in the example above.

When we compiled the package in the example above, a new folder was created, called "mypack".

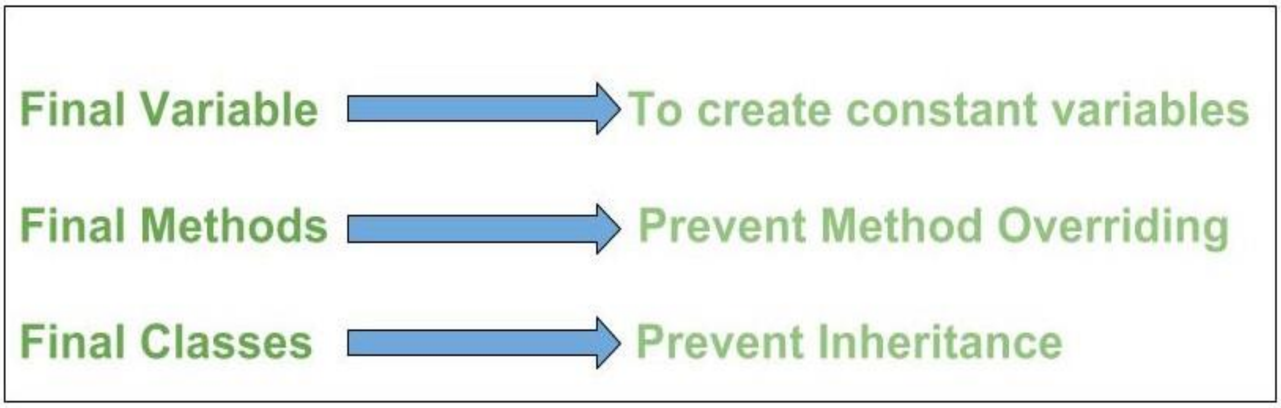
To run the MyPackageClass.java file, write the following:

```
C:\Users\Your Name>java mypack.MyPackageClass
```

The output will be:

```
This is my package!
```

final keyword is used in different contexts. Applicable **only to a variable, a method or a class**.Following are different contexts where final is used.


Final Variable
To create constant variables
Final Methods
Prevent Method Overriding
Final Classes
Prevent Inheritance