Object Oriented Programming enables you to develop large scale software and GUIs effectively.

A class defines the properties and behaviours for objects. OOP involves programming using objects. Object represents an entity in the real world that can be distinctly identified. An object is a unique entity that has its own identity, state, behaviour. The state of an object is defined by data fields with their current value. The behaviour of an object is defined by its methods.

Objects of the same type are defined by using a common class. A class is a template, blue-print, or a contract that defines what an object’s data fields and methods will be. An object is an instance of a class. Creating an instance is referred to as instantiation.

Java uses variables to define data fields and methods to define actions. Additionally, a class provides methods of a special type, known as constructors, which are invoked to create a new object. A constructor can perform any action, but constructors are designed to perform initialising actions, such as initialising the data fields of an object.

*Example: Defining Classes and Creating Objects*

*class* Circle{

*double* radius;  *//Data Field*

    Circle(){           *//Constructor to set radius to 1*

        radius = 1;     *//No-Arg Constructor*

    }

    Circle(*double* newRadius){   *//Constructor to set a specified radius*

        radius = newRadius;

    }

*double* getArea(){

*return* radius\*radius\*Math.PI;

    }

*double* getPerimeter(){

*return* 2\*radius\*Math.PI;

    }

*void* setRadius(*double* newRadius){

        radius = newRadius;

    }

}

*public* *class* SimpleCircle{

*public* *static* *void* main(*String* [] args) {

*Circle* c1 = *new* Circle();   *//Create circle wth radius 1*

        System.out.println("Area of the circle with given radius is: " + c1.getArea());

*Circle* c2 = *new* Circle(25);     *//Create circle with radius 25*

        System.out.println("Area of the circle with given radius is: " + c2.getArea());

        c2.setRadius(40);   *//Changing the radius of circle from 25 to 40*

        System.out.println("Updated Area of the circle with given radius is: " + c2.getArea());

    }

}

Constructor is a special function in class which has function name equal to class name and has no return type. A constructor is invoked to create an object using the new operator. Constructors play the role of initializing objects. The constructor has exactly the same name as its defining class. Like regular methods, constructors can be overloaded (i.e., multiple constructors can have the same name but different signatures), making it easy to construct objects with different initial data values. Constructors are a special kind of method. They have three peculiarities:

* A constructor must have the same name as the class itself.
* Constructors do not have a return type—not even void.
* Constructors are invoked using the new operator when an object is created.

Constructor has exactly the same name as its defining class. Like regular methods, constructors can be overloaded (i.e., multiple constructors, can have the same name but different signatures), making it easy to construct objects with different initial data values.

Constructors are used to construct objects. To construct an object from a class, invoke the constructor of the class using the new operator. If arguments are passed in a constructor, that is called as a **parametrised constructor**. This is shown in below syntax:

*new* ClassName(arguments);

A class normally provides a constructor without arguments (e.g., Circle() ). Such a constructor is called as **no-arg** or a **no-argument constructor**.

A class may or may not be defined without constructors. In this case, a public no-arg constructor with empty body is implicitly defined in the class. The constructor is called as a **default constructor**, provided automatically only if no constructors are explicitly defined in the class.

*Copy Constructors*

* Sometimes, we face a problem where we required to create an exact copy of an existing object of the class. There is also a condition, if we have made any changes in the copy it should not reflect in the original one and vice-versa. For such cases, Java provides the concept of a copy constructor.
* In Java, a copy constructor is a special type of constructor that creates an object using another object of the same Java class. It returns a duplicate copy of an existing object of the class. We can assign a value to the final field but the same cannot be done while using the clone() method. It is used if we want to create a deep copy of an existing object. It is easier to implement in comparison to the clone() method.
* Use of copy constructor:
  + Create a copy of an object that has multiple fields.
  + Generate a deep copy of the heavy objects.
  + Avoid the use of the Object.clone() method.
* Advantages of copy constructor:
  + If a field declared as final, the copy constructor can change it.
  + There is no need for typecasting.
  + Its use is easier if an object has several fields.
  + Addition of field to the class is easy because of it. We need to change only in the copy constructor.

Both the copy constructor and the clone() method are used to create a copy of an existing object of the class. But the use of copy constructor is easier and better in comparison to the clone() method because of the reasons given below:

* If we are using the clone() method, it is necessary to import the Cloneable The method may throw the exception CloneNotSupportException. So, handling the exception in a program is a complex task. While in copy constructor there are no such complexities.
* We cannot assign a value if the fields are final. While in the copy constructor we can assign values to the final fields.
* The object returned by the clone() method must be typecast. While in copy constructor there is no such requirement.

*Accessing Objects via Reference Variables*

An Object’s data and methods can be accessed through the dot (.) operator via the object’s reference variable. Newly created objects are allocated in the memory. They can be accessed via reference variables.

Objects are accessed via object’s reference variables, which contain references to the objects. Such references are declared using following syntax:

*ClassName* objectRefVar;

A class is essentially a programmer-defined type. A class is a reference-type, which means that a variable of the class toe can reference an instance of the class. Following statement declares a variable myCircle to be a Circle type. The variable myCircle can reference a Circle object. The next statement creates an object and assigns its reference to myCircle.

*Circle* myCircle;

myCircle = *new* Circle();

A single statement can be written as:

*ClassName* objectRefVar = *new* ClassName();

*NOTE*: An object reference variable that appears to hold an object actually contains a reference to that object. Strictly speaking, an object reference variable and an object are different, but most of the time the distinction can be ignored. Therefore, it is fine, for simplicity, to say that myCircle is a Circle object rather than use the longer-winded description that myCircle is a variable that contains a reference to a Circle object.

*NOTE*: Arrays are treated as objects in Java. Arrays are created using the new operator. An array variable is actually a variable that contains a reference to an array.

An object reference variable that appears to hold an object actually contains a reference to that object. Strictly speaking, an object reference variable and an object are different, but most of the time the distinction can be ignored. Therefore, it is fine, for simplicity, to say that myCircle is a Circle object rather than use the longer-winded description that myCircle is a variable that contains a reference to a Circle object:

* objectRefVar.dataField references a data field in the object
* objectRefVar.method(arguments) invokes a method on the object

The data fields can be of reference types. For example, the following Student class contains a data field name of the String type. String is a predefined Java class.

*class* Student {

*String* name; *// name has the default value null*

*int* age; *// age has the default value 0*

*boolean* isScienceMajor; *// isScienceMajor has default value false*

*char* gender; *// gender has default value '\u0000'*

}

If a data field of a reference type does not reference any object, the data field holds a special Java value, null. null is a literal just like true and false. While true and false are Boolean literals, null is a literal for a reference type.

A static variable is shared by all objects of the class. A static method cannot access instance members of the class.

If you want all the instances of a class to share data, use *static variables*, also known as class variables. Static variables store values for the variables in a common memory location. Because of this common location, if one object changes the value of a static variable, all objects of the same class are affected. Java supports static methods as well as static variables. *Static methods* can be called without creating an instance of the class. This is shown in the following example:

*public* *class* simpleCircle{

*public* *static* *void* main(*String* [] args) {

        System.out.println("Before creating Objects");

        System.out.println("The number of objects existing = " + circleWithStaticMembers.getNumberOfObjects());

*//Creating Object C1*

        circleWithStaticMembers c1 = *new* circleWithStaticMembers();

        System.out.println("After creating object 1");

        System.out.println("radius: " + c1.radius);

        System.out.println("Number of objects: " + circleWithStaticMembers.noOfObjs);

*//Creating Object C2*

        circleWithStaticMembers c2 = *new* circleWithStaticMembers(5);

        System.out.println("After creating object 2");

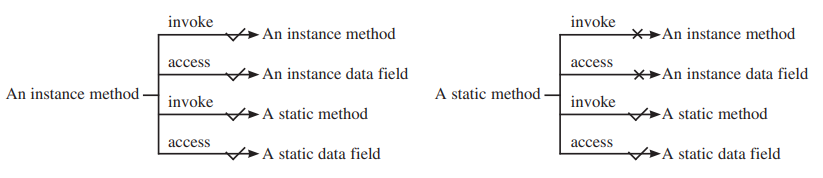
        System.out.println("radius: " + c2.radius);

        System.out.println("Number of objects: " + circleWithStaticMembers.noOfObjs);

    }

}

An instance method can invoke an instance or static method and access an instance or static data field. A static method can invoke a static method and access a static data field. However, a static method cannot invoke an instance method or access an instance data field, since static methods and static data fields don’t belong to a particular object. The relationship between static and instance members is summarized in the following diagram:



Visibility Modifiers can be used to specify the visibility of a class and its members. A visibility modifier specifies how data fields and methods in a class can be accessed from outside the class. There is no restriction on accessing data fields and methods from inside the class.

You can use the public visibility modifier for classes, methods, and data fields to denote that they can be accessed from any other classes. If no visibility modifier is used, then by default the classes, methods, and data fields are accessible by any class in the same package. This is known as package-private or package-access.

Public Specifiers achieves the highest level of accessibility.

Classes, methods, and fields declared as public can be accessed from any class in the Java program, whether these classes are in the same package or in another package.

Private Specifiers achieves the lowest level of accessibility.

Private methods and fields can only be accessed within the same class to which the methods and fields belong.

Private methods and fields are not visible within subclasses and are not inherited by subclasses. Using Private Specifier, we can hide data from the outside world.

Methods and fields declared as protected can only be accessed by the subclasses in other package or any class within the package.

When you don't set access specifier for the element, it will follow the default accessibility level.

There is no default specifier keyword. Classes, variables, and methods can be default accessed.

Using default specifier we can access class, method, or field which belongs to same package, but not from outside this package.

Passing an object to a method is to pass the reference of the object. You can pass objects to methods. Like passing an array, passing an object is actually passing the reference of the object.

An array can hold objects as well as primitive type values. When an array of objects is created using the new operator, each element in the array is a reference variable with a default value of null.

You can define immutable classes to create immutable objects. The contents of immutable objects cannot be changed. Normally, you create an object and allow its contents to be changed later.

However, occasionally it is desirable to create an object whose contents cannot be changed once the object has been created. We call such an object as immutable object and its class as immutable class. The String class, for example, is immutable.

The scope of instance and static variables is the entire class, regardless of where the variables are declared.

Instance and static variables in a class are referred to as the class’s variables or data fields. A variable defined inside a method is referred to as a local variable.

The scope of a class’s variables is the entire class, regardless of where the variables are declared. A class’s variables and methods can appear in any order in the class. The exception is when a data field is initialized based on a reference to another data field.

If a local variable has the same name as a class’s variable, the local variable takes precedence and the class’s variable with the same name is hidden.

The keyword this refers to the object itself. It can also be used inside a constructor to invoke another constructor of the same class.

The ‘this’ keyword is the name of a reference that an object can use to refer to itself. You can use the ‘this’ keyword to reference the object’s instance members. The ‘this’ keyword can be used to reference a class’s hidden data fields. The ‘this’ keyword can be used to invoke another constructor of the same class.