Classes and Objects in Java

Classes and Objects are basic concepts of Object Oriented Programming that revolve around real-life entities.

Class

A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers**: A class can be public or has default access (Refer to this for details).
2. **Class keyword:** class keyword is used to create a class.
3. **Class name:** The name should begin with an initial letter (capitalized by convention).
4. **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
5. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
6. **Body:** The class body is surrounded by braces, { }.

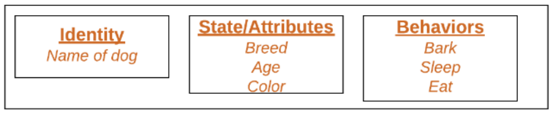
Constructors are used for initializing new objects. Fields are variables that provide the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.
There are various types of classes that are used in real-time applications such as nested classes, anonymous classes, and lambda expressions.

Object

It is a basic unit of Object-Oriented Programming and represents real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

1. **State**: It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior**: It is represented by the methods of an object. It also reflects the response of an object to other objects.
3. **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

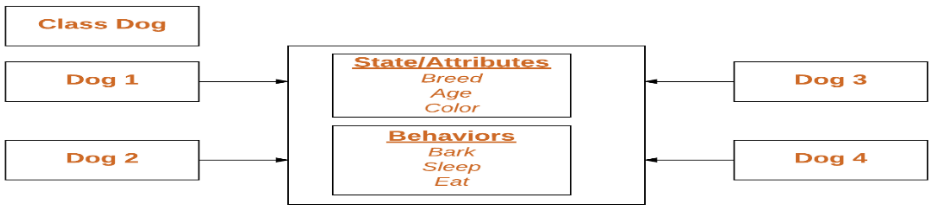
Example of an object: dog



Objects correspond to things found in the real world. For example, a graphics program may have objects such as "circle", "square", and "menu". An online shopping system might have objects such as "shopping cart", "customer", and "product".
Objects (Declaration, Initialization, Creation)

Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be **instantiated**. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.
Example:



As we declare variables like (type name;). This notifies the compiler that we will use the name to refer to data whose type is type. With a primitive variable, this declaration also reserves the proper amount of memory for the variable. So for reference variables, the type must be strictly a concrete class name. In general, we **can't** create objects of an abstract class or an interface.

```
Dog tuffy;
```

If we declare a reference variable(Tuffy) like this, its value will be undetermined(null) until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object.

Initializing an object

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the class constructor.

```
public class Dog{
    // Instance Variables
    String name;
    String breed;
    int age;
    String color;
    // Constructor Declaration of Class
    public Dog(String name, String breed,
                   int age, String color)
    {
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }
    // method 1
    public String getName(){
        return name;}
    // method 2
    public String getBreed() {
        return breed;}
    // method 3
    public int getAge(){
        return age; }
    // method 4
    public String getColor(){
        return color;}
    @Override
    public String toString(){
        return("Hi my name is "+ this.getName()+
               ".\nMy breed,age and color are " +
               this.getBreed()+"," + this.getAge()+
               ","+ this.getColor());
    }
    public static void main(String[] args){
        Dog tuffy = new Dog("tuffy","papillon", 5, "white");
        System.out.println(tuffy.toString());                                                    }}
```

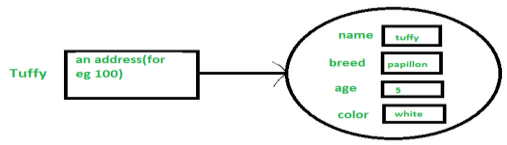
Output:

```
Hi my name is tuffy.
My breed,age and color are papillon,5,white
```

- This class contains a single constructor. We can recognize a constructor because its declaration uses the same name as the class and it has no return type. The Java compiler differentiates the constructors based on the number and the type of the arguments. The constructor in the *Dog* class takes four arguments. The following statement provides "tuffy","papillon",5,"white" as values for those arguments:

```
Dog tuffy = new Dog("tuffy","papillon",5, "white");
```

- The result of executing this statement can be illustrated as :



Note : All classes have at least **one** constructor. If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, also called the default constructor. This default constructor calls the class parent's no-argument constructor (as it contains only one statement i.e super();), or the *Object* class constructor if the class has no other parent (as the Object class is the parent of all classes either directly or indirectly).

Ways to create an object of a class

There are four ways to create objects in the java. Strictly speaking there is only one way(by using a *new* keyword), and the rest internally use *new* keyword.

- **Using new keyword:** It is the most common and general way to create an object in java. Example:

```
// creating object of class Test
Test t = new Test();
```

- **Using Class.forName(String className) method:** There is a pre-defined class in java.lang package with name Class. The forName(String className) method returns the Class object associated with the class with the given string name. We have to give a fully qualified name for a class. On calling new Instance() method on this Class object returns new instance of the class with the given string name.

```
// creating object of public class Test
// consider class Test present in com.p1 package
Test obj = (Test)Class.forName("com.p1.Test").newInstance();
```

- **Using clone() method:** clone() method is present in the Object class. It creates and returns a copy of the object

```
// creating object of class Test
Test t1 = new Test();
// creating clone of above object
Test t2 = (Test)t1.clone();
```

- **Deserialization:** De-serialization is a technique of reading an object from the saved state in a file. Refer to Serialization/De-Serialization in java

```
FileInputStream file = new FileInputStream(filename);
ObjectInputStream in = new ObjectInputStream(file);
Object obj = in.readObject();
```

Creating multiple objects by one type only (A good practice)

- In real-time, we need different objects of a class in different methods. Creating a number of references for storing them is not a good practice and therefore we declare a static reference variable and use it whenever required. In this case, the wastage of memory is less. The objects that are not referenced anymore will be destroyed by Garbage Collector of java. Example:

```
Test test = new Test();
test = new Test();
```

- In the inheritance system, we use a parent class reference variable to store a sub-class object. In this case, we can switch into different subclass objects using the same referenced variable. Example:

```
class Animal {}
class Dog extends Animal {}
class Cat extends Animal {}
public class Test
{
    // using Dog object
    Animal obj = new Dog();

    // using Cat object
    obj = new Cat();
}
```

Anonymous objects

Anonymous objects are objects that are instantiated but are not stored in a reference variable.

- They are used for immediate method calling.
- They will be destroyed after method calls.
- They are widely used in different libraries. For example, in AWT libraries, they are used to perform some action on capturing an event(eg a key press).
- In the example below, when a key button(referred by the btn) is pressed, we are simply creating an anonymous object of EventHandler class for just calling the handle method.

```
btn.setOnAction(new EventHandler()
{
    public void handle(ActionEvent event) {
        System.out.println("Hello World!");}
});
```

'this' reference in Java

'this' is a reference variable that refers to the current object.

Following are the ways to use 'this' keyword in java :

1. Using 'this' keyword to refer current class instance variables

```
//Java code for using 'this' keyword to
//refer current class instance variables
class Test
{
    int a;
    int b;

    // Parameterized constructor
    Test(int a, int b)
    {
        this.a = a;
        this.b = b;
    }

    void display()
    {
        //Displaying value of variables a and b
        System.out.println("a = " + a + "  b = " + b);
    }

    public static void main(String[] args)
    {
        Test object = new Test(10, 20);
        object.display();
    }
}
```

Output:

```
a = 10  b = 20
```

2. Using this() to invoke current class constructor

Java

```
// Java code for using this() to
// invoke current class constructor
class Test
{
    int a;
    int b;
    //Default constructor
    Test()
    {
```

```java
        this(10, 20);
        System.out.println("Inside  default constructor \n");
    }
        //Parameterized constructor
    Test(int a, int b)
    {
        this.a = a;
        this.b = b;
        System.out.println("Inside parameterized constructor");
    }
    public static void main(String[] args)
    {
        Test object = new Test();}}
```

Output:

```
Inside parameterized constructor
Inside  default constructor
```

3. Using 'this' keyword to return the current class instance

Java

```java
//Java code for using 'this' keyword
//to return the current class instance
class Test
{
    int a;
    int b;
    //Default constructor
    Test()
    {
        a = 10;
        b = 20;
    }
       //Method that returns current class instance
    Test get()
    {
        return this;
    }
        //Displaying value of variables a and b
    void display()
    {
        System.out.println("a = " + a + "  b = " + b);
    }
    public static void main(String[] args)
    {
        Test object = new Test();
        object.get().display();
    }
}
```

Output:

```
a = 10  b = 20
```

4. Using 'this' keyword as method parameter

```java
// Java code for using 'this'
// keyword as method parameter
class Test
{
    int a;
    int b;
        // Default constructor
    Test()
    {
        a = 10;
        b = 20;
    }
       // Method that receives 'this' keyword as parameter
    void display(Test obj)
    {
        System.out.println("a = " +obj.a + "  b = " + obj.b);
    }
     // Method that returns current class instance
    void get()
    {
        display(this);
    }
    public static void main(String[] args)
    {
        Test object = new Test();
        object.get();
    }
}
```

Output:

```
a = 10  b = 20
```

5. Using 'this' keyword to invoke current class method

Java

```java
// Java code for using this to invoke current
// class method
class Test {

    void display()
    {
        // calling function show()
        this.show();

       System.out.println("Inside display function");
    }

    void show() {
        System.out.println("Inside show function");
    }


    public static void main(String args[]) {
        Test t1 = new Test();
        t1.display();
    }
}
```

Output:

```
Inside show function
Inside display function
```

6. Using 'this' keyword as an argument in the constructor call

Java

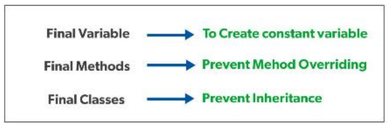
```java
// Java code for using this as an argument in constructor
// call
// Class with object of Class B as its data member
class A
{
    B obj;

    // Parameterized constructor with object of B
    // as a parameter
    A(B obj)
    {
        this.obj = obj;

     // calling display method of class B
        obj.display();
    }

}

class B
{
    int x = 5;

    // Default Constructor that create a object of A
    // with passing this as an argument in the
   // constructor
    B()
    {
        A obj = new A(this);
    }

    // method to show value of x
    void display()
    {
        System.out.println("Value of x in Class B : " + x);
    }

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

Output:

```
Value of x in Class B : 5
```

final Keyword in Java

final keyword is used in different contexts. First of all, the *final* is a non-access modifier applicable only to a variable, a method, or a class. The following are different contexts where the final is used.



Final Variable

When a variable is declared with the ***final keyword***, its value can't be modified, essentially, a constant. This also means that you must initialize a final variable. If the final variable is a reference, this means that the variable cannot be re-bound to reference another object, but the internal state of the object pointed by that reference variable can be changed i.e. you can add or remove elements from the final array or final collection. It is good practice to represent final variables in all uppercase, using underscore to separate words.

Illustration:

```
final int THRESHOLD = 5;
// Final variable
final int THRESHOLD;
// Blank final variable
static final double PI = 3.141592653589793;
// Final static variable PI
static final double PI;
// Blank final static  variable
```

Initializing a final Variable

We must initialize a final variable, otherwise, the compiler will throw a compile-time error. A final variable can only be initialized once, either via an initializer or an assignment statement. There are three ways to initialize a final variable:

1. You can initialize a final variable when it is declared. This approach is the most common. A final variable is called a **blank final variable** if it is **not** initialized while declaration. Below are the two ways to initialize a blank final variable.
    1. A blank final variable can be initialized inside an instance-initializer block or inside the constructor. If you have more than one constructor in your class then it must be initialized in all of them, otherwise, a compile-time error will be thrown.
    2. A blank final static variable can be initialized inside a static block.

Let us see these two different ways of initializing a final variable:

```java
// Java Program to demonstrate Different
// Ways of Initializing a final Variable

// Main class
class GFG {

    // a final variable
    // direct initialize
    final int THRESHOLD = 5;

    // a blank final variable
    final int CAPACITY;

    // another blank final variable
    final int  MINIMUM;

    // a final static variable PI
    // direct initialize
    static final double PI = 3.141592653589793;

    // a  blank final static  variable
    static final double EULERCONSTANT;

    // instance initializer block for
    // initializing CAPACITY
    {
        CAPACITY = 25;
    }

    // static initializer block for
    // initializing EULERCONSTANT
    static{
        EULERCONSTANT = 2.3;
    }

    // constructor for initializing MINIMUM
    // Note that if there are more than one
    // constructor, you must initialize MINIMUM
    // in them also
    public GFG()
    {
        MINIMUM = -1;}}
```

Geeks there was no main method in the above code as it was simply for illustration purposes to get a better understanding in order to draw conclusions:

Observation 1: When to use a final variable?

The only difference between a normal variable and a final variable is that we can re-assign the value to a normal variable but we cannot change the value of a final variable once assigned. Hence final variables must be used only for the values that we want to remain constant throughout the execution of the program.

Observation 2: Reference final variable.

When a final variable is a reference to an object, then this final variable is called the reference final variable. For example, a final StringBuffer variable looks defined below as follows:

```
final StringBuffer sb;
```

As we all know that a final variable cannot be re-assign. But in the case of a reference final variable, the internal state of the object pointed by that reference variable can be changed. Note that this is not re-assigning. This property of the *final* is called *non-transitivity*. To understand what is meant by the internal state of the object as shown in the below example as follows:

Example 1:

```
// Java Program to demonstrate
// Reference of Final Variable
// Main class
class GFG {
    // Main driver method
    public static void main(String[] args)
    {
        // Creating sn object of StringBuilder class
        // Final reference variable
        final StringBuilder sb = new StringBuilder("Geeks");

        // Printing the element in StringBuilder object
        System.out.println(sb);

        // changing internal state of object reference by
        //  final reference variable sb
        sb.append("ForGeeks");

        // Again printing the element in StringBuilder
        // object after appending above element in it
        System.out.println(sb);}}
```

Output

```
Geeks
GeeksForGeeks
```

The *non-transitivity* property also applies to arrays, because arrays are objects in Java. Arrays with the **final keyword** are also called final arrays.

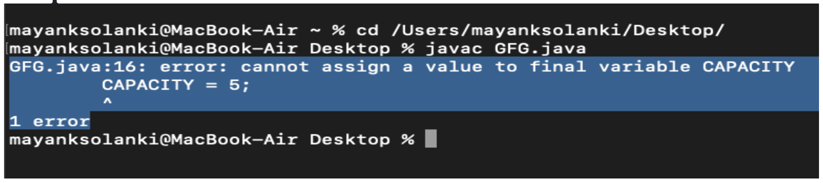
Note: As discussed above, a final variable cannot be reassign, doing it will throw compile-time error.

Example 2:

Java

```
// Java Program to Demonstrate Re-assigning
// Final Variable will throw Compile-time Error
// Main class
class GFG {
    // Declaring and customly initializing
    // static final variable
    static final int CAPACITY = 4;

    // Main driver method
    public static void main(String args[])
    {
        // Re-assigning final variable
        // will throw compile-time error
        CAPACITY = 5;   }}
```

Output:



```
mayanksolanki@MacBook-Air ~ % cd /Users/mayanksolanki/Desktop/
mayanksolanki@MacBook-Air Desktop % javac GFG.java
GFG.java:16: error: cannot assign a value to final variable CAPACITY
        CAPACITY = 5;
        ^
1 error
mayanksolanki@MacBook-Air Desktop %
```

Remember: When a final variable is created inside a method/constructor/block, it is called local final variable, and it must initialize once where it is created. See below program for local final variable.

Example:

Java

```
// Java program to demonstrate
// local final variable
// Main class
class GFG {
    // Main driver method
    public static void main(String args[])
    {
        // Declaring local final variable
        final int i;
        // Now initializing it with integer value
        i = 20;
        // Printing the value on console
        System.out.println(i);   }}
```

Output

```
20
```

Remember the below key points as perceived before moving forward as listed below as follows:

1. Note the difference between C++ *const* variables and Java *final* variables. const variables in C++ must be assigned a value when declared. For final variables in Java, it is not necessary as we see in the above examples. A final variable can be assigned value later, but only once.
2. *final* with foreach loop: final with for-each statement is a legal statement.

Example

```
// Java Program to demonstrate Final
// with for-each Statement
// Main class
class GFG {
    // Main driver method
    public static void main(String[] args)
    {
        // Declaring and initializing
        // custom integer array
        int arr[] = { 1, 2, 3 };
        // final with for-each statement
        // legal statement
        for (final int i : arr)
            System.out.print(i + " "); }}
```

output

```
1 2 3
```

Output explanation: Since the "*i*" variable goes out of scope with each iteration of the loop, it is actually re-declared in each iteration, allowing the same token (i.e. i) to be used to represent multiple variables.

Final classes

When a class is declared with the *final* keyword, it is called a final class. A final class cannot be extended(inherited).

There are two uses of a final class:

Usage 1: One is definitely to prevent inheritance, as final classes cannot be extended. For example, all Wrapper Classes like Integer, Float, etc. are final classes. We can not extend them.

```
final class A
{
     // methods and fields
}
// The following class is illegal
class B extends A
{
    // COMPILE-ERROR! Can't subclass A
}
```

Usage 2: The other use of final with classes is to create an immutable class like the predefined String class. One can not make a class immutable without making it final.

Final Methods

When a method is declared with *final* keyword, it is called a final method. A final method cannot be overridden. The Object class does this—a number of its methods are final. We must declare methods with the final keyword for which we are required to follow the same implementation throughout all the derived classes.

Illustration: Final keyword with a method

```
class A {
    final void m1()
    {
        System.out.println("This is a final method.");
    }}
class B extends A
{
    void m1()
    {
        // Compile-error! We can not override
        System.out.println("Illegal!");     }}
```

For more examples and behavior of final methods and final classes, please see Using final with inheritance. Please see the abstract in java article for differences between the final and abstract.
Related Interview Question(Important): Difference between final, finally, and finalize in Java