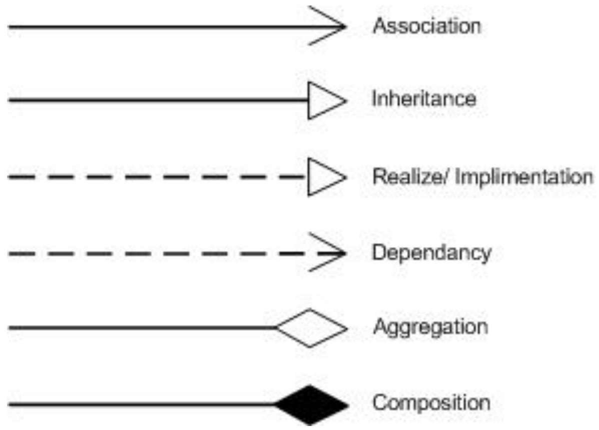
Tables of Content

1. **Class Diagram**

Class diagrams are widely used to describe the types of objects in a system and their relationships. Class diagrams model class structure and contents using design elements such as classes, packages and objects. Class diagrams describe three different perspectives when designing a system, conceptual, specification, and implementation. These perspectives become evident as the diagram is created and help solidify the design.

The Class diagrams, physical data models, along with the system overview diagram are in my opinion the most important diagrams that suite the current day rapid application development requirements.

UML notations:



http://www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep#Class_Diagram

2. **Basic Data Types**

Primitive Data Types:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword. Let us now look into detail about the eight primitive data types.

byte:

- Byte data type is an 8-bit signed two's complement integer.
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.
- Example: byte a = 100 , byte b = -50

short:

- Short data type is a 16-bit signed two's complement integer.
- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767 (inclusive) (2^15 -1)

Basic Data Types

- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int
- Default value is 0.
- Example: short s = 10000, short r = -20000

int:

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648.(-2^31)
- Maximum value is 2,147,483,647(inclusive).(2^31 -1)
- Int is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0.
- Example: int a = 100000, int b = -200000

long:

- Long data type is a 64-bit signed two's complement integer.
- Minimum value is -9,223,372,036,854,775,808.(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive). (2^63 -1)
- This type is used when a wider range than int is needed.

Basic Data Types

- Default value is 0L.
- Example: long a = 100000L, int b = -200000L

float:

- Float data type is a single-precision 32-bit IEEE 754 floating point.
- Float is mainly used to save memory in large arrays of floating point numbers.
- Default value is 0.0f.
- Float data type is never used for precise values such as currency.
- Example: float f1 = 234.5f

double:

- double data type is a double-precision 64-bit IEEE 754 floating point.
- This data type is generally used as the default data type for decimal values, generally the default choice.
- Double data type should never be used for precise values such as currency.
- Default value is 0.0d.
- Example: double d1 = 123.4

Basic Data Types

boolean:

- boolean data type represents one bit of information.
- There are only two possible values: true and false.
- This data type is used for simple flags that track true/false conditions.
- Default value is false.
- Example: boolean one = true

char:

- char data type is a single 16-bit Unicode character.
- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example: char letterA ='A'

Reference Data Types:

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy etc.

Basic Data Types

- Class objects, and various type of array variables come under reference data type.
- Default value of any reference variable is null.
- A reference variable can be used to refer to any object of the declared type or any compatible type.
- Example: Animal animal = new Animal("giraffe");

http://www.tutorialspoint.com/java/java_basic_datatypes.htm

Access Modifiers

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

- Visible to the package. the **default**. No modifiers are needed.
- Visible to the class only (**private**).
- Visible to the world (**public**).
- Visible to the package and all subclasses (**protected**).

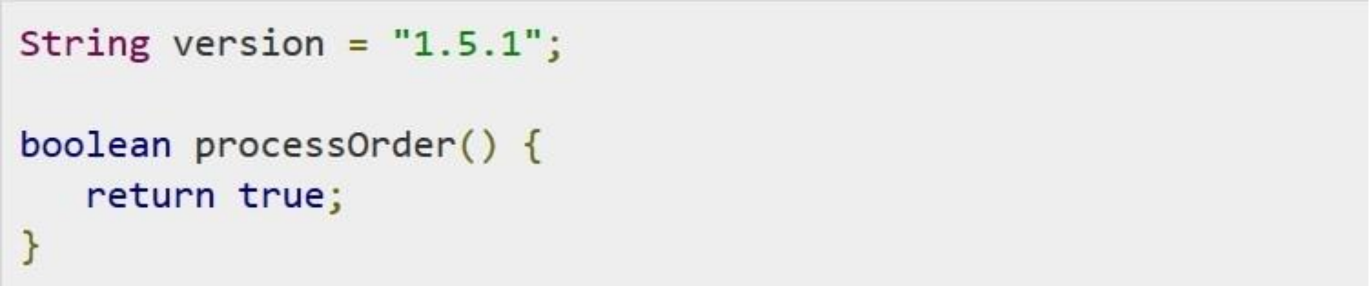
Access Modifiers

Default Access Modifier - No keyword:

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

Example:

Variables and methods can be declared without any modifiers, as in the following examples:



```
String version = "1.5.1";

boolean processOrder() {
   return true;
}
```

Access Modifiers

Private Access Modifier - private:

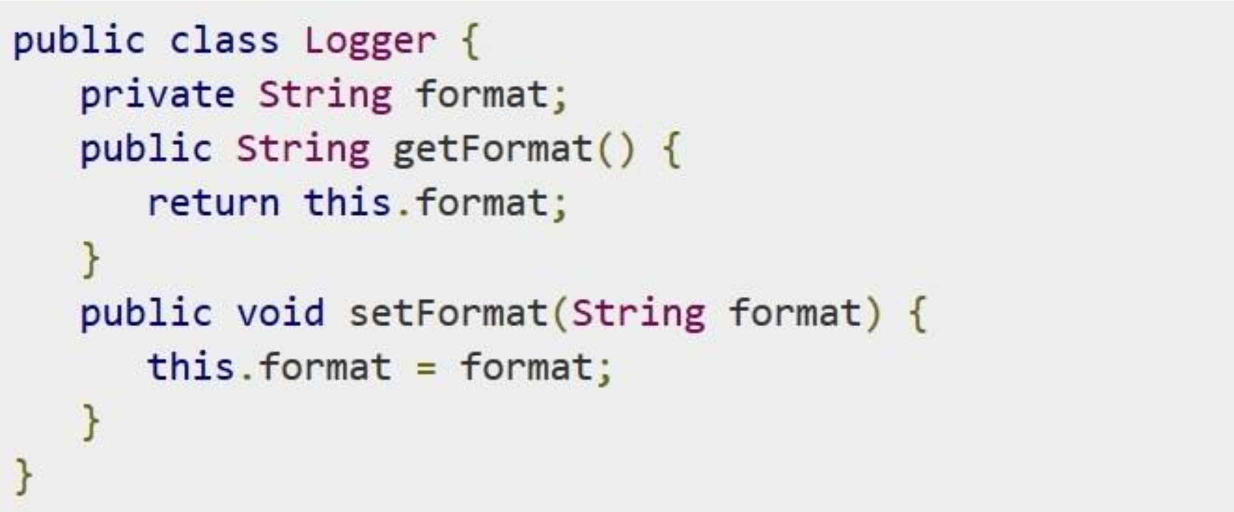
Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself.

Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

Variables that are declared private can be accessed outside the class if public getter methods are present in the class.

Using the private modifier is the main way that an object encapsulates itself and hide data from the outside world.

Example (*The following class uses private access control****):***



```
public class Logger {
   private String format;
   public String getFormat() {
      return this.format;
   }
   public void setFormat(String format) {
      this.format = format;
   }
}
```

Here, the *format* variable of the Logger class is private, so there's no way for other classes to retrieve or set its value directly. So to make this variable available to the outside world, we defined two public methods: *getFormat()*, which returns the value of format, and *setFormat(String)*, which sets its value.

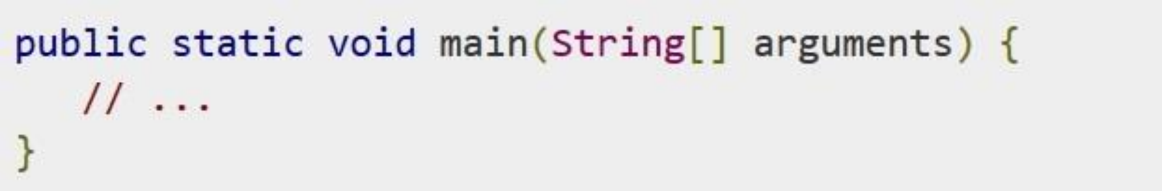
Access Modifiers

Public Access Modifier - public:

A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe. However if the public class we are trying to access is in a different package, then the public class still need to be imported.

Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

Example (The following function uses public access control)



```
public static void main(String[] arguments) {
   // ...
}
```

The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

Access Modifiers

Protected Access Modifier - protected:

Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

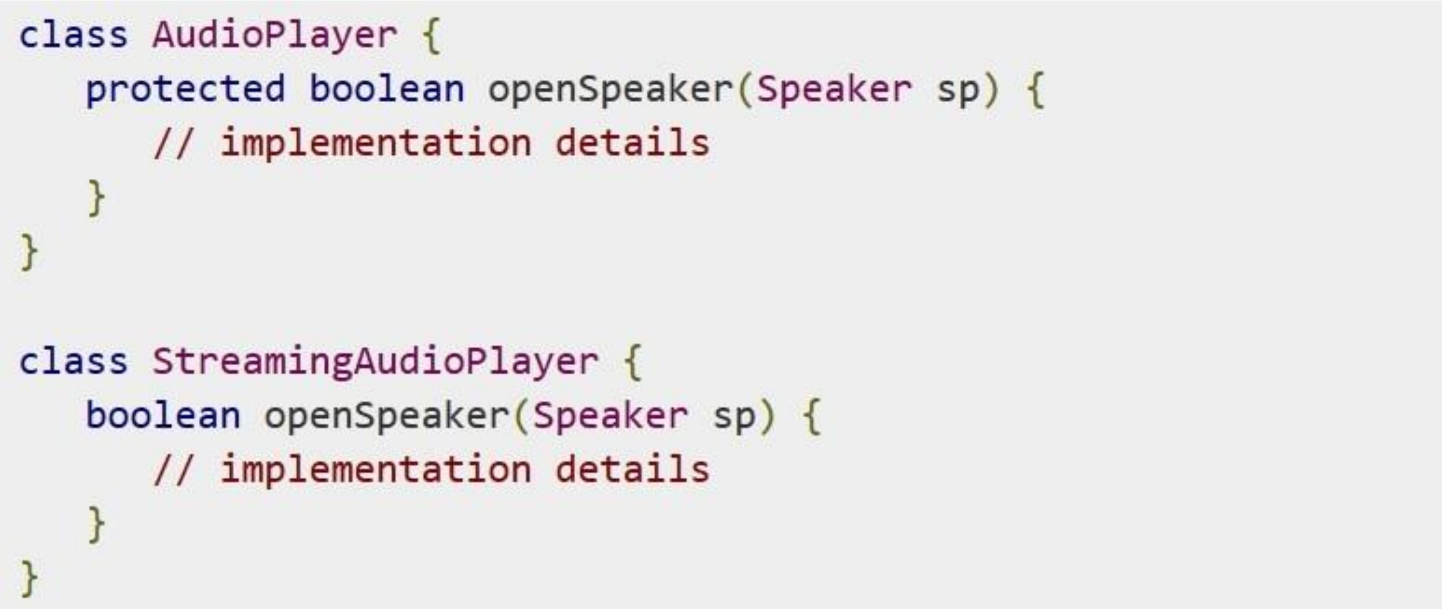
The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

Example:

The following parent class uses protected access control, to allow its child class override *openSpeaker()* method:

Here, if we define *openSpeaker()* method as private, then it would not be accessible from any other class other than *AudioPlayer*. If we define it as public, then it would become accessible to all the outside world. But our intension is to expose this method to its subclass only, thats why we used *protected* modifier.



```
class AudioPlayer {
   protected boolean openSpeaker(Speaker sp) {
      // implementation details
   }
}

class StreamingAudioPlayer {
   boolean openSpeaker(Speaker sp) {
      // implementation details
   }
}
```

Access Control and Inheritance:

The following rules for inherited methods are enforced:

- Methods declared public in a superclass also must be public in all subclasses.
- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
- Methods declared without access control (no modifier was used) can be declared more private in subclasses.
- Methods declared private are not inherited at all, so there is no rule for them.

http://www.tutorialspoint.com/java/java_access_modifiers.htm

Java Non Access Modifiers

Java provides a number of non-access modifiers to achieve many other functionality.

- The *static* modifier for creating class methods and variables
- The *final* modifier for finalizing the implementations of classes, methods, and variables.
- The *abstract* modifier for creating abstract classes and methods.
- The *synchronized* and *volatile* modifiers, which are used for threads.

The static Modifier:

Static Variables:

The *static* key word is used to create variables that will exist independently of any instances created for the class. Only one copy of the static variable exists regardless of the number of instances of the class.

Static variables are also known as class variables. Local variables cannot be declared static.

Static Methods:

The static key word is used to create methods that will exist independently of any instances created for the class.

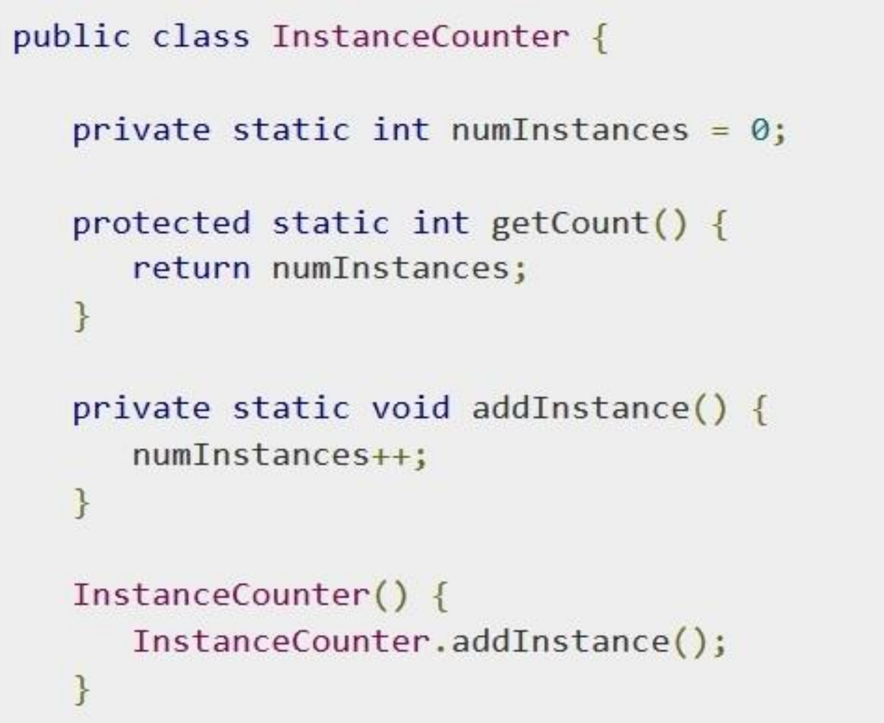
Static methods do not use any instance variables of any object of the class they are defined in. Static methods take all the data from parameters and compute something from those parameters, with no reference to variables.

Java Non Access Modifiers

Class variables and methods can be accessed using the class name followed by a dot and the name of the variable or method.

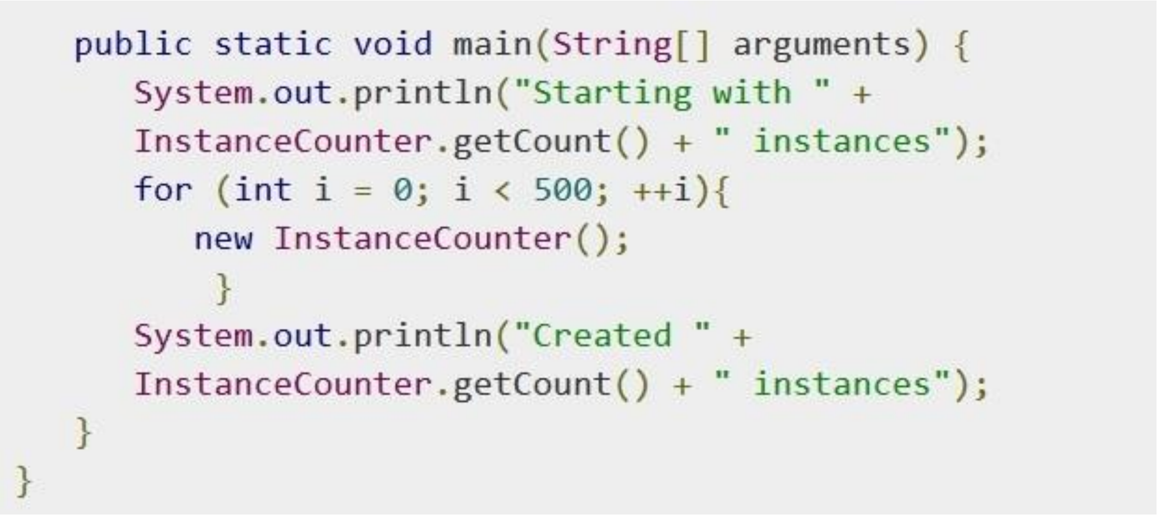
Example:

The static modifier is used to create class methods and variables, as in the following example:



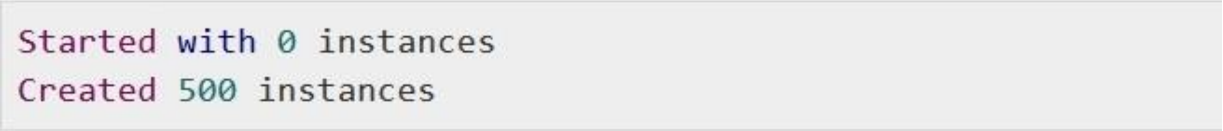
```java
public class InstanceCounter {

   private static int numInstances = 0;

   protected static int getCount() {
      return numInstances;
   }

   private static void addInstance() {
      numInstances++;
   }

   InstanceCounter() {
      InstanceCounter.addInstance();
   }
```

Java Non Access Modifiers



```
   public static void main(String[] arguments) {
      System.out.println("Starting with " +
      InstanceCounter.getCount() + " instances");
      for (int i = 0; i < 500; ++i){
         new InstanceCounter();
          }
      System.out.println("Created " +
      InstanceCounter.getCount() + " instances");
   }
}
```

This would produce the following result:



```
Started with 0 instances
Created 500 instances
```

Java Non Access Modifiers

The final Modifier:

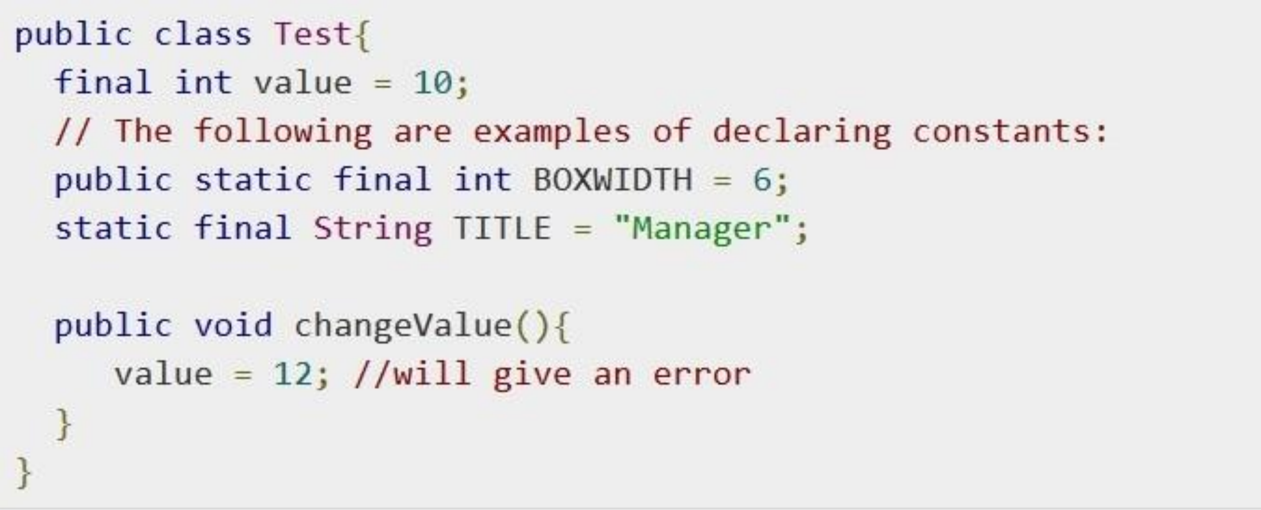
final Variables:

A final variable can be explicitly initialized only once. A reference variable declared final can never be reassigned to refer to an different object.

However the data within the object can be changed. So the state of the object can be changed but not the reference.

With variables, the *final* modifier often is used with *static* to make the constant a class variable.

Example:



```
public class Test{
  final int value = 10;
  // The following are examples of declaring constants:
  public static final int BOXWIDTH = 6;
  static final String TITLE = "Manager";

  public void changeValue(){
     value = 12; //will give an error
  }
}
```

Java Non Access Modifiers

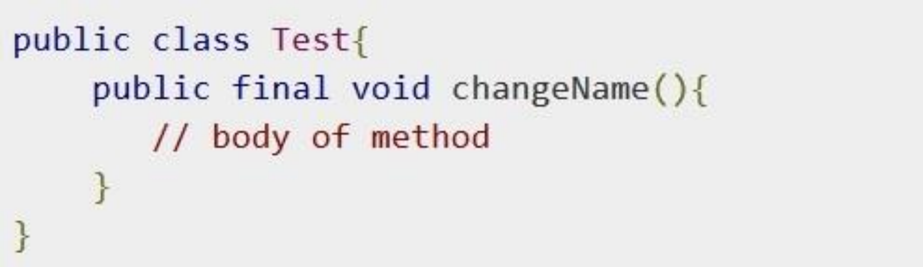
final Methods:

A final method cannot be overridden by any subclasses. As mentioned previously the final modifier prevents a method from being modified in a subclass.

The main intention of making a method final would be that the content of the method should not be changed by any outsider.

Example:

You declare methods using the *final* modifier in the class declaration, as in the following example:



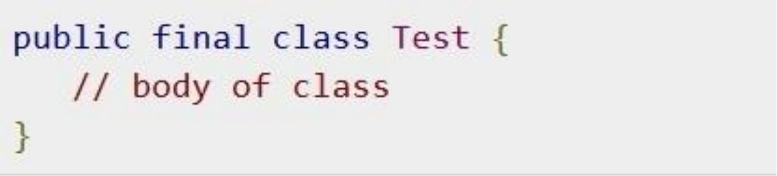
```
public class Test{
    public final void changeName(){
       // body of method
    }
}
```

Java Non Access Modifiers

final Classes:

The main purpose of using a class being declared as *final* is to prevent the class from being subclassed. If a class is marked as final then no class can inherit any feature from the final class.

Example:



```
public final class Test {
   // body of class
}
```

Java Non Access Modifiers

The abstract Modifier:

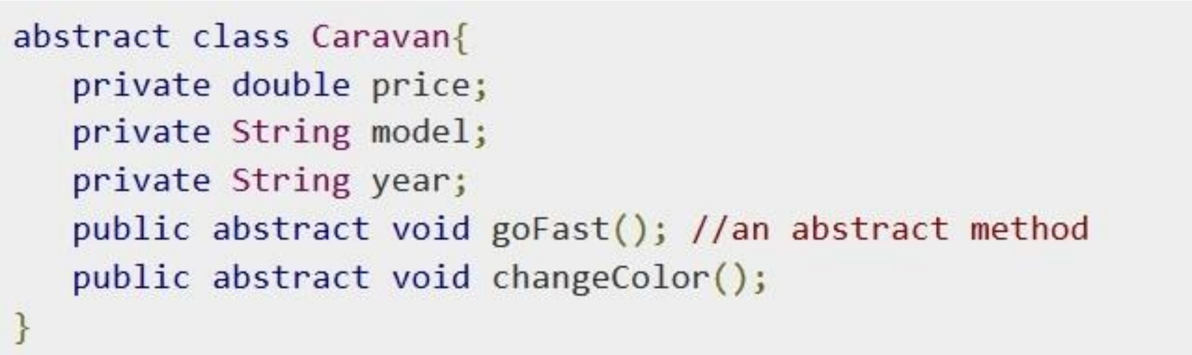
abstract Class:

An abstract class can never be instantiated. If a class is declared as abstract then the sole purpose is for the class to be extended.

A class cannot be both abstract and final. (since a final class cannot be extended). If a class contains abstract methods then the class should be declared abstract. Otherwise a compile error will be thrown.

An abstract class may contain both abstract methods as well normal methods.

Example:



```
abstract class Caravan{
   private double price;
   private String model;
   private String year;
   public abstract void goFast(); //an abstract method
   public abstract void changeColor();
}
```

Java Non Access Modifiers

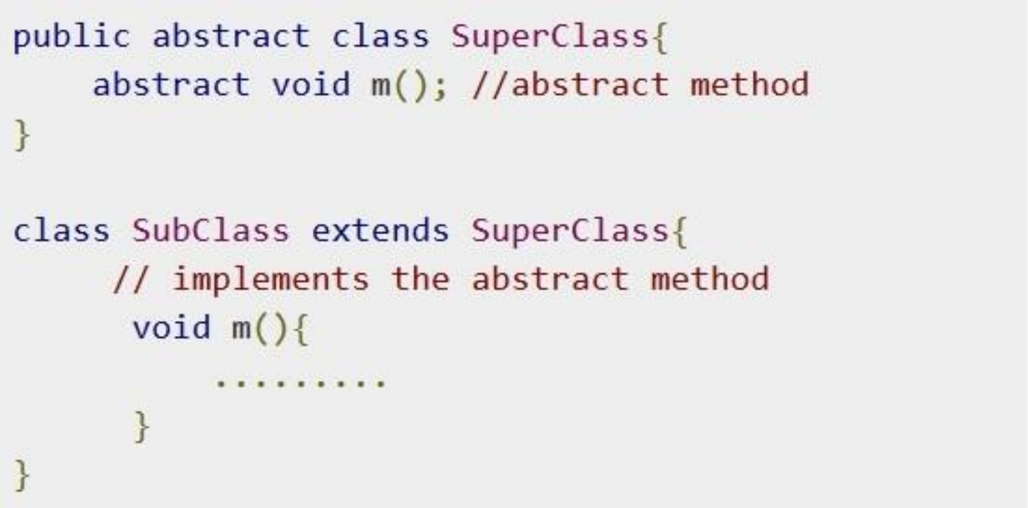
abstract Methods:

An abstract method is a method declared with out any implementation. The methods body(implementation) is provided by the subclass. Abstract methods can never be final or strict.Any class that extends an abstract class must implement all the abstract methods of the super class unless the subclass is also an abstract class.

If a class contains one or more abstract methods then the class must be declared abstract. An abstract class does not need to contain abstract methods.

The abstract method ends with a semicolon. Example: public abstract sample();

Example:



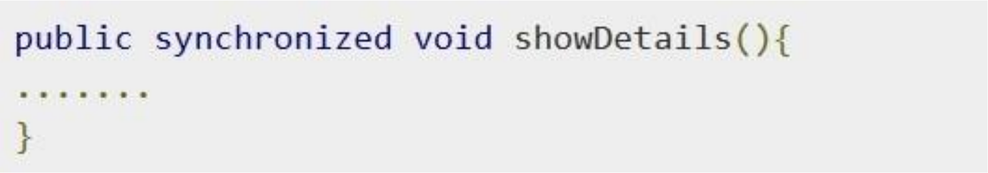
```
public abstract class SuperClass{
    abstract void m(); //abstract method
}

class SubClass extends SuperClass{
     // implements the abstract method
      void m(){
          .........
      }
}
```

Java Non Access Modifiers

The synchronized Modifier:

The synchronized key word used to indicate that a method can be accessed by only one thread at a time. The synchronized modifier can be applied with any of the four access level modifiers.

Example:



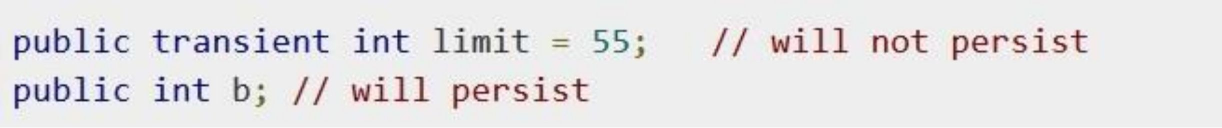
```
public synchronized void showDetails(){
.......
}
```

The transient Modifier:

An instance variable is marked transient to indicate the JVM to skip the particular variable when serializing the object containing it.

This modifier is included in the statement that creates the variable, preceding the class or data type of the variable.

Example:



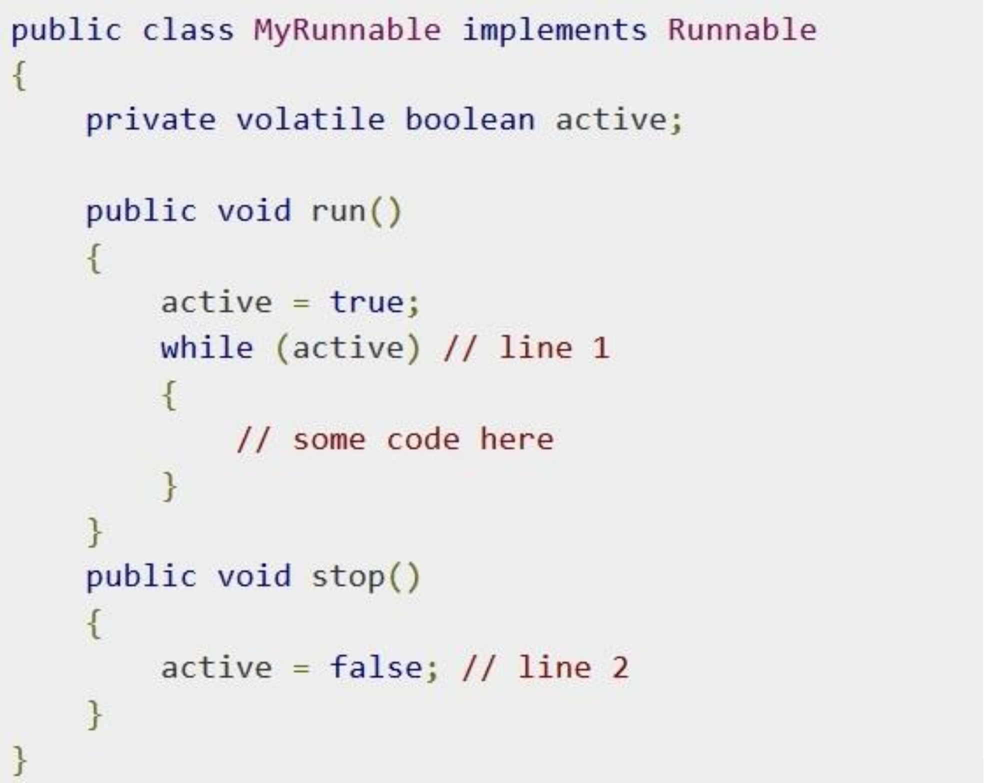
```
public transient int limit = 55;   // will not persist
public int b; // will persist
```

Java Non Access Modifiers

The volatile Modifier:

The volatile is used to let the JVM know that a thread accessing the variable must always merge its own private copy of the variable with the master copy in the memory. Accessing a volatile variable synchronizes all the cached copied of the variables in the main memory. Volatile can only be applied to instance variables, which are of type object or private. A volatile object reference can be null.

Example:



```
public class MyRunnable implements Runnable
{
    private volatile boolean active;

    public void run()
    {
        active = true;
        while (active) // line 1
        {
            // some code here
        }
    }
    public void stop()
    {
        active = false; // line 2
    }
}
```

Usually, run() is called in one thread (the one you start using the Runnable), and stop() is called from another thread. If in line 1 the cached value of active is used, the loop may not stop when you set active to false in line 2. That's when you want to use *volatile*.

http://www.tutorialspoint.com/java/java_nonaccess_modifiers.htm

Java Constructor

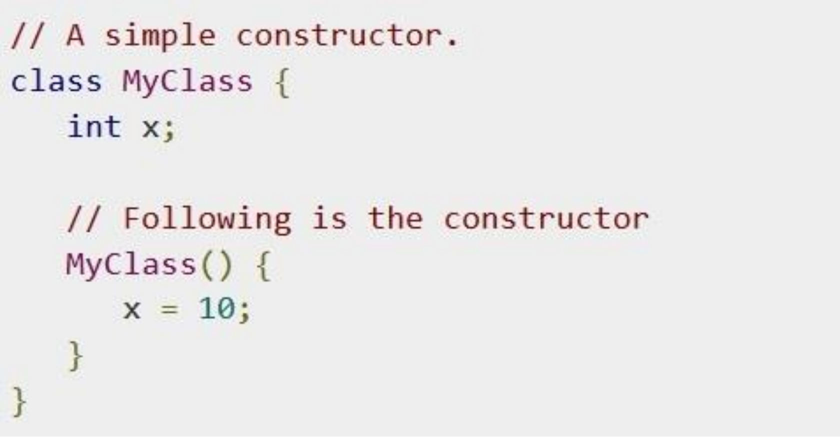
The Constructors:

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

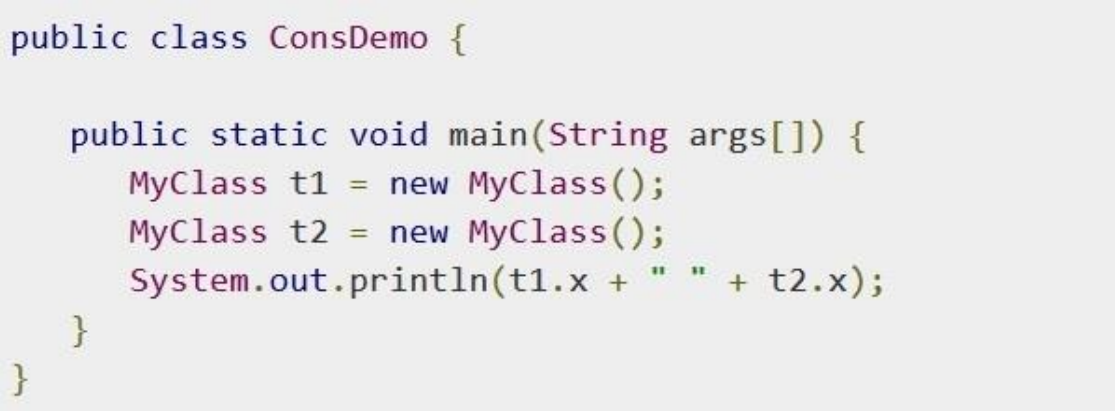
Here is a example that uses a constructor:



```
// A simple constructor.
class MyClass {
   int x;

   // Following is the constructor
   MyClass() {
      x = 10;
   }
}
```

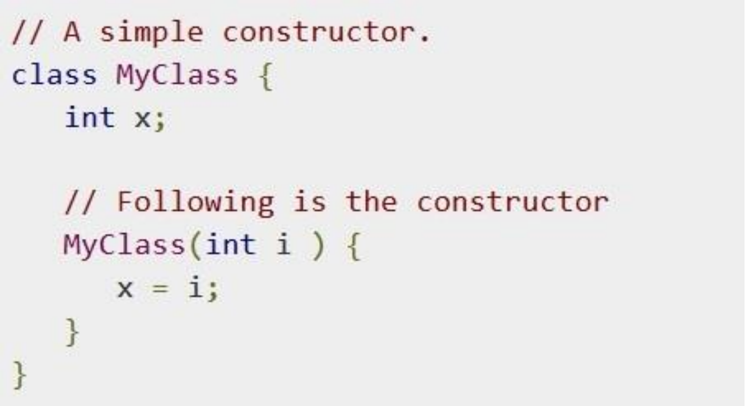
Java Constructor

You would call constructor to initialize objects as follows:



```
public class ConsDemo {

   public static void main(String args[]) {
      MyClass t1 = new MyClass();
      MyClass t2 = new MyClass();
      System.out.println(t1.x + " " + t2.x);
   }
}
```

Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

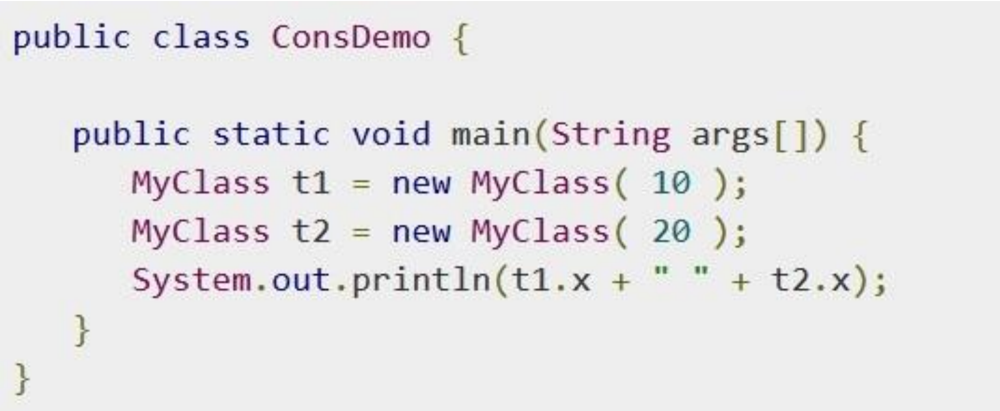


```
// A simple constructor.
class MyClass {
   int x;

   // Following is the constructor
   MyClass(int i ) {
      x = i;
   }
}
```

Here is a simple example that uses a constructor:

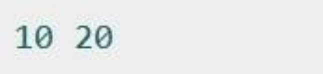
Java Constructor

You would call constructor to initialize objects as follows:



```java
public class ConsDemo {

   public static void main(String args[]) {
      MyClass t1 = new MyClass( 10 );
      MyClass t2 = new MyClass( 20 );
      System.out.println(t1.x + " " + t2.x);
   }
}
```

This would produce the following result:



```
10 20
```

http://www.tutorialspoint.com/java/java_methods.htm

Using **this** keyword in Java

Within an instance method or a constructor, this is a reference to the *current object* — the object whose method or constructor is being called. You can refer to any member of the current object from within an instance method or a constructor by using this.

Using this with a Field

The most common reason for using the this keyword is because a field is shadowed by a method or constructor parameter.

For example, the Point class was written like this

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

but it could have been written like this:

Using **this** keyword in Java

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Each argument to the constructor shadows one of the object's fields — inside the constructor **x** is a local copy of the constructor's first argument. To refer to the Point field **x**, the constructor must use **this.x**.

Using this with a Constructor

From within a constructor, you can also use the this keyword to call another constructor in the same class. Doing so is called an *explicit constructor invocation*. Here's another Rectangle class, with a different implementation from the one in the Objects section.

https://docs.oracle.com/javase/tutorial/java/javaOO/thiskey.html

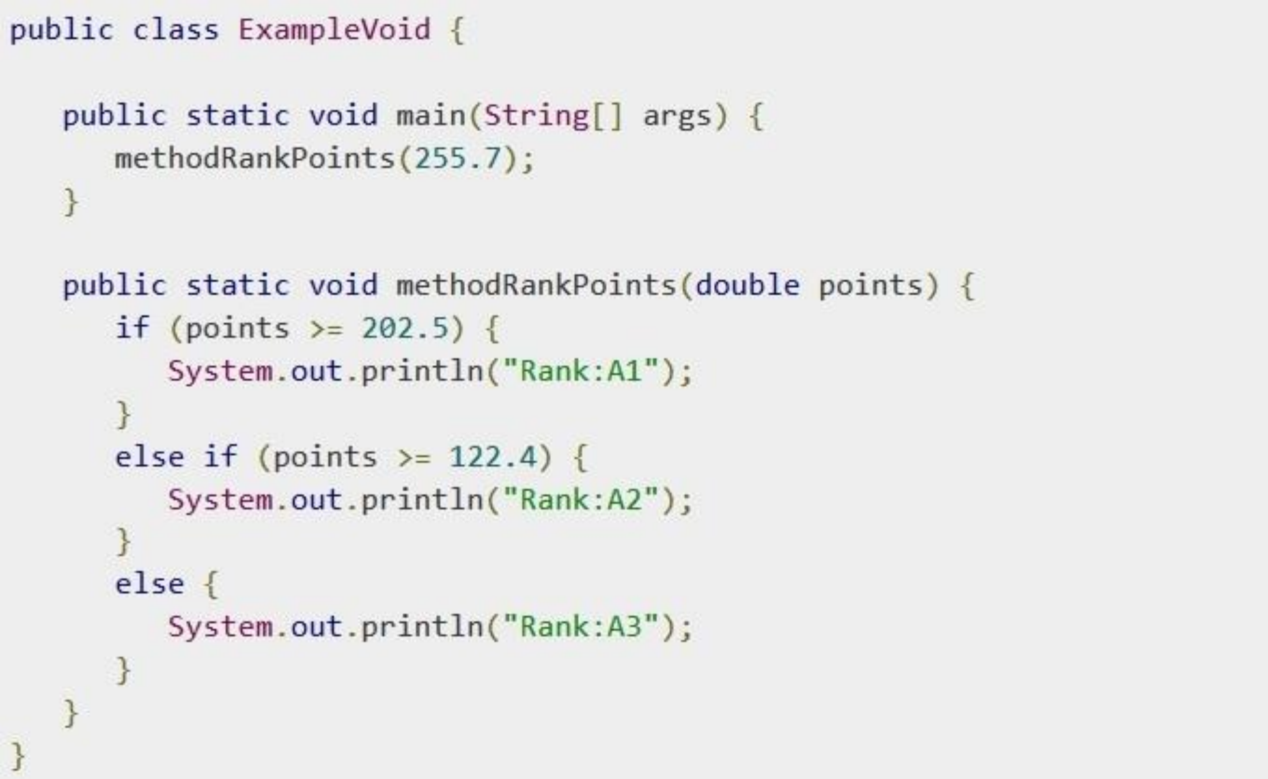
Using **this** keyword in Java

```
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0, 1, 1);
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
}
```

This class contains a set of constructors. Each constructor initializes some or all of the rectangle's member variables. The constructors provide a default value for any member variable whose initial value is not provided by an argument. For example, the no-argument constructor creates a 1x1 Rectangle at coordinates 0,0. The two-argument constructor calls the four-argument constructor, passing in the width and height but always using the 0,0 coordinates. As before, the compiler determines which constructor to call, based on the number and the type of arguments. If present, the invocation of another constructor must be the first line in the constructor.

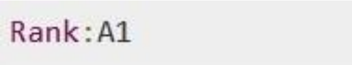
Using **void** keyword in Java

The void keyword allows us to create methods which do not return a value. Here, in the following example we're considering a void method *methodRankPoints*. This method is a void method which does not return any value. Call to a void method must be a statement i.e. *methodRankPoints(255.7);*. It is a Java statement which ends with a semicolon as shown below.



```
public class ExampleVoid {

   public static void main(String[] args) {
      methodRankPoints(255.7);
   }

   public static void methodRankPoints(double points) {
      if (points >= 202.5) {
         System.out.println("Rank:A1");
      }
      else if (points >= 122.4) {
         System.out.println("Rank:A2");
      }
      else {
         System.out.println("Rank:A3");
      }
   }
}
```

This would produce the following result:



```
Rank:A1
```

http://www.tutorialspoint.com/java/java_methods.htm

Using **return** keyword in Java

A method returns to the code that invoked it when it

- completes all the statements in the method,
- reaches a return statement, or
- throws an exception (covered later),

whichever occurs first.

You declare a method's return type in its method declaration. Within the body of the method, you use the return statement to return the value.

Any method declared void doesn't return a value. It does not need to contain a return statement, but it may do so. In such a case, a return statement can be used to branch out of a control flow block and exit the method and is simply used like this:

```
return;
```

If you try to return a value from a method that is declared void, you will get a compiler error.

Any method that is not declared void must contain a return statement with a corresponding return value, like this:

```
return returnValue;
```

Using **return** keyword in Java

The data type of the return value must match the method's declared return type; you can't return an integer value from a method declared to return a boolean.

The `getArea()` method in the `Rectangle` `Rectangle` class that was discussed in the sections on objects returns an integer:

```
    // a method for computing the area of the rectangle
    public int getArea() {
        return width * height;
    }
```

This method returns the integer that the expression `width*height` evaluates to.

The `getArea` method returns a primitive type. A method can also return a reference type. For example, in a program to manipulate `Bicycle` objects, we might have a method like this:

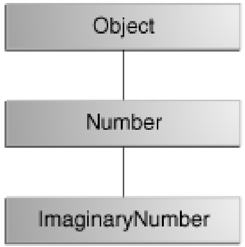
```
public Bicycle seeWhosFastest(Bicycle myBike, Bicycle yourBike,
                              Environment env) {
    Bicycle fastest;
    // code to calculate which bike is
    // faster, given each bike's gear
    // and cadence and given the
    // environment (terrain and wind)
    return fastest;                              }
```

Using **return** keyword in Java

Returning a Class or Interface

If this section confuses you, skip it and return to it after you have finished the lesson on interfaces and inheritance.

When a method uses a class name as its return type, such as `whosFastest` does, the class of the type of the returned object must be either a subclass of, or the exact class of, the return type. Suppose that you have a class hierarchy in which `ImaginaryNumber` is a subclass of `java.lang.Number`, which is in turn a subclass of `Object`, as illustrated in the following figure.



The class hierarchy for ImaginaryNumber

Now suppose that you have a method declared to return a `Number`:

```
public Number returnANumber() {
    ...
}
```

Using **return** keyword in Java

The `returnANumber` method can return an `ImaginaryNumber` but not an `Object`. `ImaginaryNumber` is a `Number` because it's a subclass of `Number`. However, an `Object` is not necessarily a `Number` — it could be a `String` or another type.

You can override a method and define it to return a subclass of the original method, like this:

```
public ImaginaryNumber returnANumber() {
    ...
}
```

This technique, called *covariant return type*, means that the return type is allowed to vary in the same direction as the subclass.

https://docs.oracle.com/javase/tutorial/java/javaOO/returnvalue.html

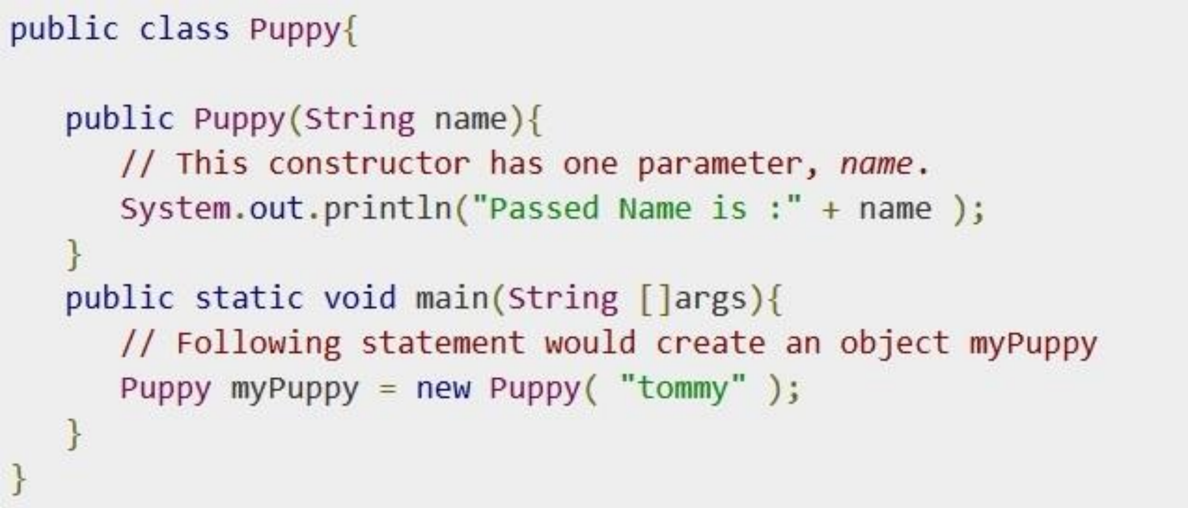
Creation of class instances (objects)

As mentioned previously, a class provides the blueprints for objects. So basically an object is created from a class. In Java, the new key word is used to create new objects.

There are three steps when creating an object from a class:

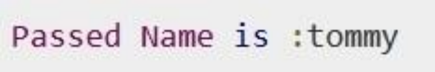
- **Declaration:** A variable declaration with a variable name with an object type.
- **Instantiation:** The 'new' key word is used to create the object.
- **Initialization:** The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Example of creating an object is given below:



```
public class Puppy{

   public Puppy(String name){
      // This constructor has one parameter, name.
      System.out.println("Passed Name is :" + name );
   }
   public static void main(String []args){
      // Following statement would create an object myPuppy
      Puppy myPuppy = new Puppy( "tommy" );
   }
}
```

If we compile and run the above program, then it would produce the following result:



```
Passed Name is :tommy
```

http://www.tutorialspoint.com/java/java_object_classes.htm