The This key word:

sometimes a method will need to refer to the object that invoked it. To allow this java defines the 'this' key word.
"this" can be used inside any method to refer to the current object. That is, 'this' is always a reference to the object on which the method was invoked.

Final key word

A field can be declared as final. Doing so prevents its content from being modified making it, essentially a constant.
This means that you must initialize a final field when it is declared.

It is a common coding convention to choose all uppercase identifiers for final fields.

```
final int FILE_OPEN = 2;
```

Unfortunately, final guarantees immutability only when instance variables are primitive types, not reference types.
If an instance variable of a reference type has the final modifier, the value of that instance variable (the reference to an object) will never change - it will always refer to the same object, but the value of the object itself can change.

The finalize() Method:

Sometimes an object will need to perform some action when it is destroyed.
To handle such situation, Java provides a mechanism called finalization. By using finalization you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
To add a ~~finalize~~ finalizer to a class you simply define the finalize() method. The Java run time calls the finalize() method on the object.

```
protected void finalize(){
    //finalization code here
}
```

Constructors:

Once defined, the Constructors is automatically called when ~~an~~ the Object is created, before the new operator completes. Constructors look a little strange because they have no ~~[illegible]~~ return type, not even void.

This is because the implicit return type of a class' constructor is a class type itself.

In the line

~~myBox1~~

```
Box myBox1 = new Box();
```

new Box() is calling the ~~constructor~~ Constructor Box();

Inheritance and Constructors in Java

In Java, Constructor of base class with no argument gets automatically called in derived class Constructor.
For example, output of the following program given below is.
Base class Constructor called
Derived class constructor called.

```
//file name Main.Java
class Base {
    Base () {
        Sout ("Base class Constructor called");
    }
}
class Derived extends Base {
    Derived () {
        Sout (" Derived class constructor called");
    }
}
```

```
public class Main {
    public static void main (String [] args) {
        Derived d = new Derived ();
    }
}
```

Any class will have a default constructor does not matter if we declare it in the class or not

If we inherit a class, then the derived ~~class~~ class must call its super class constructor. It is done by default in derived class.

If it does not have a default constructor in the derived class, the JVM will invoke its default constructor and call the super class constructor by default.

If we have a parameterized constructor in the derived class still it calls the default super class constructor by default.

In this case, if the super class does not have a default constructer, instead, it has a parameterized constructer, then the derived class constructor should explicitly call the parameterized super class constructer.

Packages

Packages are containers for classes. They are used to keep the class name space compartmentalized.

For example - a package allows you to create a class named list, which can store in your own package without concern that it will collide with some other class named list stored elsewhere. Packages are stored in a hierarchical manner and are ~~explicitly~~ explicitly imported into new class definition.

The package is both a naming and a visibility control mechanism.

The following statement creates a package called mypackage:
package myPackage;

Java uses file system directories to store packages for example, the .class files for any classes you declare to be part of mypackage must be stored in a directory called mypackage. Remember that case is significant, and the directory name must match the package name exactly.

A package hierarchy must be reflected in the file system of your Java development system. For example a package ~~[illegible]~~ declared as
package java.awt.image.

needs to be stored in java\awt\image in a Windows environment.
Be sure to ~~use~~ choose your package names carefully.

You cannot rename a package without renaming the directory in which the classes are stored.

How does the Java run-time System know where to look for packages that you create?

The answer has 3 Parts ->

- First - by default, the Java run time System uses the current working directory as it's starting point. Thus if your package is in a subdirectory of the current directory it will be found.

- Second - you can Specify a directory path or paths by setting the ~~Class path to~~ CLASSPATH environmental variables

- Third. You can use the - classpath option with Java and javac to specify the path to your classes

When a package is imported , only those items within the package declared as public be available to non-subclass in the importing code.

Static :

When a Member is declared Static , it can be accessed before any object of its class are created and without any reference to any object.

You can declare both methods and variables to be static.

The most common example of Static is main (),
Main () is declared as static because it must be called before any object exist.

Static method in Java is a method which belong to the class and not to the object.

A static method can access only static data.
It cannot access non-static data
(instance variable)

A non static member belongs to an Instance.
Its ~~meaningling~~ meaningless without somehow
resolving which instance of a ~~g~~ class you
are talking about. In a static context, you
don't have an instance, that's why you can't
access a non static member without
explicitly mentioning an object reference

In fact, you can access a non static member
in a static context by specifying the object
reference explicitly:

```
public class Human {
        String message = "Hello world";
        public static void display (Human human) {
            System.out.println(human.message);
        }
        public static void main (String [] args) {
            Human kunal = new Human();
            kunal.message = "Kunal's message.";
            Human.display (kunal);

        }
```

Output = "Kunal's message"

- A static method can call only other static methods and cannot call a non-static method from it.
- A static method can be accessed directly by the class name and doesn't need any object
- A static method cannot refer to "this" or "super" keywords in anyway.

If you need to do computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded.

```
// Demonstrate static variables, methods and blocks
class UseStatic {
      static int a = 3;
      static int b;
      static void meth (int x) {
            Sop ("x = " + x);
            Sop ("a = " + a);
            Sop ("b = " + b);
   }
static {
    sout ("static block initialized");
    b = a*4;
}
public static void main (String args[]) {
    meth (42);
}
}
```

As soon as the usestatic class is loaded all the static ~~elements~~ ~~are~~ statements are run, First a is set to 3, then the static block executes, which prints a message and then initializes b to a*4, or 12. Then main() is called, which calls meth(), passing 42 to x. The three println() statements refer to the two static variables a and b, as well as the local variable x

<u>output</u>

```
=> Static block initialized   x=42
a=3
b=12
```

Note: main Method is static, Since it Must be accessible for an application to run before any instantiation takes place.

Note- only nested ~~static~~ Classes can be static

Note- Static inner classes can have static variables

You can't override the inherited static methods, in Java overriding takes place by resolving the type of object at run-time and ~~compile~~ compile time and then calling the respective method.

Static methods are class level methods, so it is always resolved during compile time.
Static Interface Methods are not inherited by either an implementing class or a sub-interface.

Note:

```
public class Static {
    // class test // ERROR
    static class Test {
        String name;
    }
        public Test (String name) {
            this. name = name;
        }
    }
    public static void main (String args []) {
        Test a = new Test ("Kunal");
        Test b = new Test ("Rahul");

        System. out. println (a.name); // Kunal
        System. out. println (b.name); // Rahul
    }
```

Because :

The static keyword may modify the declaration of a member type C within the body of a non-inner class or interface T.

Its effect is to declare that C is not an inner class.

Just as a static method of T has no current instance of T in its body C also has no current instance of T, nor does it have any Lexically enclosing instances.

Here, Test does not have any instance of its outer class Static. Neither does main.

But main & Test can have instance of each other.