The this Keyword:

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the this keyword.

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 Keyword:

A field can be declared as final. Doing so prevents its contents 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 situations, 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 finalizer to a class, you simply define the finalize( ) method. The Java run time calls that method whenever

it is about to recycle an object of that class. Right before an asset is freed, the Java run time calls the finalize( )

method on the object.

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

Constructors:

Once defined, the constructor is automatically called when the object is created, before the new operator completes.

Constructors look a little strange because they have no return type, not even void.

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

In the line

```
Box mybox1 = new Box();
```

new Box( ) is calling the Box( ) constructor.

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 following program given below is:

```
Base Class Constructor Called
Derived Class Constructor Called
```

```
 // filename: Main.java
class Base {
  Base() {
    System.out.println("Base Class Constructor Called ");
  }
}

class Derived extends Base {
  Derived() {
    System.out.println("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 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 parameterised 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 constructor,

instead it has a parameterised constructor, then the derived class constructor should call explicitly call the

parameterised super class constructor.