

The Frightened Freshers Guide To CO320 - Introduction To Object Orientated Programming (Java)

Thomas Paul Carlson

January 10, 2006

1 Introduction

This document serves two major purposes for me: Firstly, as a tool to aid in the revision of the topic and secondly so that others may benefit from knowledge gained during the twelve weeks of doing this module. This is not a definitive guide to CO320, nor is it the perfect Java tutorial. However, it does serve well in introducing the syntax and general way in which you can do things in Java.

I will try where possible to use language that is accessible to all. If you have programmed before then a lot of this will be fairly easy with minor differences. If you havent then I'll introduce topics gently and gradually get more complex. Note that a lot of the topics are interlinked in some way and that you do need to know about some things before others.

2 Defining A Class

Heres where it begins - defining a class. Classes are the building blocks of Java, and you really cant do anything at all without defining a class (I mean it! You cant type any code!). Classes are used to make objects, so a good way of thinking about a class is like a Jelly (Or Jello, if you are American) mould that you can create lots of Jellys (Objects) from. An example of this is a car class from which you could create as many car objects as you like. As for the actual code to define a class, it is as follows:

```
public Class NameOfClass
{
    // Class Variables go here
    // Constructor goes here
    // Accessor methods go here
    // Mutator methods go here
}
```

```
}
```

...And thats it. Simple? Yep! Some notes about the Class definition: You need to state "public" before saying the name of the Class. Next, the class name itself is capitalised. The reasons for this will become clear later on, but basically its so that it is easier to distinguish between your Classes and Methods inside of the class (Which should start with a lowercase letter). Finally, you need to note that there are curly brackets around the main body of the code (The Constructor, Accessors and Mutators). This system of having curly brackets around bits of code happens quite a bit in Java, and some of the time the Java compiler might moan at you for including too many or too few brackets. To help stop this from happening, it is a good idea to make sure you indent your brackets.

The example above will compile okay, but it wont actually do anything. If you are using BlueJ, then you'll be able to create objects but they will be pretty useless...In the section after the next one, I'll show you how to do things with the Class, but first you need to know a little bit about Variables. If you've programmed in pretty much anything else ever then you'll know enough about Variables and can skip the section.

3 Class Variables

Variables store information. Well, at least that is all you need to know for now. If you think of a variable like a little box then it makes things somewhat easier :-). Variables are defined in the following way:

```
scope datatype nameOfVariable;
```

This may look a little bit odd, and you wont know much about scope until a little later on. Dont get scared! Trust me its easier than it looks. Heres a nice example of a variable for the class we defined previously.

```
public int helloWorld;
```

This is a public integer called helloWorld. Its scope is public (But you dont need to know what this means yet) and its an integer (You dont need to know what this means yet either...).

```
public class NameOfClass
{
    public int helloWorld;

    // Constructor goes here
    // Accessor methods go here
    // Mutator methods go here
}
```

These variables are known as Class Variables because they can be seen by all the different methods in your Class. There are other variables which you use in methods and I call those "Local" variables or "Temporary" variables, mainly because I cannot remember the proper name. These are used inside of methods and cannot be seen outside of them (Hence the "temporary" name).

One final thing to note is that Variable definitions are followed by a semicolon. Like with the curly brackets, the semicolon is something that you'll see a lot more of later on.

4 Constructors

Now that you know how to define a mould for your objects (The Class) and how to define some variables that will be visible to the whole class, why not start by doing something with these variables as soon as the object is created from the class? Things that happen as soon as the object is created are done in the Constructor. In the example in the previous section, we defined a new Class variables called "helloWorld". So what if we wanted this to be set to a certain value when the Object was created? Well, its as easy as:

```
public class NameOfClass
{
    public int helloWorld;

    NameOfClass
    {
        helloWorld = 1;
    }
    // Other stuff goes in here
}
```

So, this will set the variable named helloWorld to equal 1. You dont know much about variable types right now so that will be in the next section. Read on, or forever be left wondering what on earth an "int" is!

5 Types

Java, like almost every other programming language ever invented, has a great deal of different types that its variables can be. If you are thinking about variables like boxes, then their type is what you are allowed to put into the box. There are a great deal of variable types and a few of the major ones are listed in the table below:

Type	Description
int	Stores whole numbers
double	Stores decimals
boolean	Stores true or false
char	Stores single characters
String	Stores lots of chars

Examples of those above could be 14 (int), 3.14 (double), true (boolean), "h" (char) and "Hello World" (String).

We had the example in the previous section of setting the variable named helloWorld to equal 1. Here are some examples for the datatypes mentioned above. First a note about these variables I've declared them without a public or private on the front which means that they are variables for inclusion in a method, but this comes a tiny bit later on. Dont worry about it :-).

```
int helloWorld;
helloWorld = 59;
```

```
double mmmPi;
mmmPi = 3.14159265358;
// (Thanks to Oliver Firth for pi)
```

```
boolean isGoatDead;
isGoatDead = true;
```

```
char justASingleLetter;
justASingleLetter = "o";
```

```
String howLongsAPieceOfString;
howLongsAPieceOfString = "Twice_half_its_length!";
```

You'll note that these all take the same form of variableName = someValueHere;, this is important to remember! Use a single equals sign for variable assignment. You'll find out the significance of this soon enough...

6 Methods

So, its all well and good being able to define some variables and do stuff in the constructor with them, but what about after the object has been created? Well, for this you need methods. Methods in a Class take the following form:

```
scope returnType helloWorld(parameterDatatype parameterName)
{
// Do shiny things here
}
```

This probably looks like complete gibberish right now. So heres a real example. It takes a number and doubles it, returning the number that has been doubled.

```
public int doubleNumber(int numberToDouble)
{
    return (numberToDouble * 2);
}
```

If you stick this into an example class in BlueJ and create a new object from your class, you can call the method doubleNumber by right clicking on the object on the object bench and selecting the doubleNumber method. Give it a nice integer, hit okay and see what happens. Shock, horror...it doubles your number! This is not really a great example of a method because its pretty useless, but it at least gives you an idea of what methods can do.

It is very important to note that methods like this need a return statement somewhere in them - in this case it was "return (numberToDouble * 2);". Anything typed after the return statement will not be executed and the java compiler will in fact throw out an error at you saying something along the lines of "unreachable statement".

The datatype that is returned can be any from the list above and also you can return Objects using methods. However, we havent really covered object interaction yet so that'll come a bit later on. There is a single exception to the "must have a return statement!" rule, and that is that you may specify the type to be void. This means you dont need to return anything. An example of when you might want to use a void instead of some value is for editing one of your class variables. The example below may help you to understand this:

```
public class NameOfClass
{
    public int helloWorld;

    NameOfClass()
    {
        helloWorld = 1;
    }
    // Other stuff goes in here

    public void setHelloWorldValue(int newValue)
    {
        hellowWorld = newValue;
        // Do not need to return anything - void returntype
    }

    public int setHelloWorldValueAndReturnNewValue(int newValue)
```

```

        {
            helloWorld = newValue;
            return helloWorld;
        }
    }

```

So, when you make an object with this code then when it is created it'll have a value helloWorld to 1. If you call the first method (setHelloWorldValue) then it'll set the value of helloWorld to whatever you specified. If you call the second method (setHelloWorldValueAndReturnNewValue) then the variable will be set to whatever you specify and then returned. To test this out in bluej, copy out the code above, create a shiny new object and use its methods on the object workbench. You'll see that the first one doesn't appear to do anything (But inspect the object to check, because it does!) and that the second one will give you some feedback about the number returned.

This type of method which changes a Class variable is known as a Mutator. The other kind of method is called an Accessor and this just accesses the Class variable and does not do anything else. Below is a nice example of an Accessor.

```

public class NameOfClass
{
    public int helloWorld;

    NameOfClass ()
    {
        helloWorld = 13;
    }
    public int exampleAccessorMethod ()
    {
        return helloWorld;
    }
}

```

The example mutator method here just returns helloWorld and this is all. Nothing more. No editing. So it's an Accessor.

7 if Statements

So, now you can create methods and do things with variables. That's all well and good, but what if you want to check for some conditions? In Java, you'll just use an if statement. These take the form:

```

if (someCondition)
{
    //Do stuff if the condition was true
}

```

```

}
else
{
    //Do other stuff if the condition was false
}

```

Now, this may look a little scary right now, but its far simpler than it looks. Below is an example of a method that restricts the input of the user so that they cannot enter a number less than 0.

```

public void youCantEnterANegativeNumber (int newNumber)
{
    if (newNumber < 0)
    {
        System.out.println("You cant enter a negative number!");
    }
    else
    {
        System.out.println("Number entered okay!");
    }
}

```

Aieee new things! The new things you see before you are the condition in the if statement (`newNumber < 0`) and the `System.out.println("stuff");`. The first thing is the condition for the if statement. If the condition evaluates to true then the first bit (Before the else) is executed. If it evaluates to false then the else section is executed. Note that never will both the if and else be executed! The other new thing is the `System.out.println("Stuff");`. What this does is actually pretty easy, it prints out to the java console. If you entered this code into a new Class in BlueJ then you'd see it printing to console.

There are a few little oddities about if statements, firstly you can omit the `and` when you only have one statement in your if. Also, you do not necessarily need an else statement if you do not want something to happen if your condition is not true. An example of a method like this is:

```

public void itIsNothing (int someNumber)
{
    if (someNumber == 0)
        System.out.println(" Hello World!");
}

```

So, this method will print out "Hello World!" if the number you feed it is 0. But wait just a second, theres something new here! There is a HUGE difference between `=` and `==`. And thats the topic of the next section.

So whats with the `=` and `==`? Well, you saw at the start of this document that `=` was used to assign a value to something.

```
someString = "Hello";
```

...but also that you can use the double equals to compare things:

```
if(someString == "Hello")
```

Note that these two are very, very different. So never ever have a single equals in your if statements, and never ever have two equals signs in your variable assignment. Just dont do it. Its bad. The difference between the two is that one = sign will simply set the variable on the left hand side to the value on the right hand side. Double = signs means that java will do a comparison of the thing on the left hand side and the thing on the right hand side (And so if they are the same, true, otherwise false).

Some more interesting things to note about if statements in java including using multiple conditions. To do this, you just need to join up the different conditions with two & symbols:

```
if((someThing=1) && (someThingElse=2))
{
    // do stuff
}
```

You may have noticed that I added lots of brackets. If in doubt, add brackets around things! Anyways, to show how this works, heres a nice little example thing.

If someThing was 1 and someThingElse was 2, then you would have an if statement that (effectively) looked like this:

```
if((true) && (true))
```

Now, the double & symbol means that the whole thing will only be true if and only if (you may see this written as iff) both conditions in the statement are true. Okay, so effectively we end up with if(true), so the //dostuff bit of code will be executed. What about if someThing was 0 and someThingElse was 2? Well, you'd have false and true so it would overall be false. The table below shows all the different possibilities for the && operator.

First Variable	Second Variable	Evaulates To...
true	true	true
true	false	false
false	true	false
false	false	false

You have already seen an example of an operator when using the double = signs in the if statements, and a list of some more operators can be found at <http://java.sun.com/docs/books/tutorial/java/nutsandbolts/opssummary.html> if you are interested or want to know more.

8 Arrays

And now for something completely different. So far you have learned about how to declare variables, store things into them, use if statements and do several other things with java. But what if you want to store more than 1 of something? Say you have a jar of sweets that can hold a certain number of sweets in little tiny boxes, wouldnt it be useful to be able to see what is inside each? Crazy kind of analogy, I know, but instead of declaring as many variables as there are boxes to hold sweets it is better to create an array.

An array is something that holds multiple values in a single variable. Yeah, sounds complex but I'll get there in a bit. Heres how you declare your array.

```
datatype [] nameOfArray = new datatype [numberOfElements];
```

A few things to note about this. It looks similar to a variable declaration you've seen before, but instead it has a few new things tacked on. You have two square brackets after the datatype to tell Java that you want an array. Once you have got a name, you need to tell Java just how many things you want in your array. Another important thing is the new keyword.

Well, thats all fine and good, but what practical use does this have? Well, lets take another example. This time of a car. Our car has five seats, and these seats can be taken up by people who have names. From this, you know that you want an array and the names of people can be taken up using Strings. So, the code to declare your "car" would be:

```
String [] car = new String [5];
```

Okay, so now we have a car! So how do we put people into the car? Well, its fairly similar to the variable assignment for ints, Strings and so on.

```
nameOfArray [elementOfArray] = valueToStore;
```

So if one of the people wanted to sit in seat 2 then you could say:

```
car [1] = "John_Smith";
```

But wait, why did I put 1 and not 2? I said I wanted them to sit in seat 2! Well, this is something very very very important to do with Arrays. They start at 0. Yes, zero. Zilch. Nada. Nothing. If you tried to store something at position 5 in the array (car[5] = "whatever) then you would get an `indexOutOfBoundsException` thrown by the Java compiler.

So remember this well...arrays start at the number 0 not the number 1!

What if you wanted to fill the car with people called bob? Well, you could do the same as above on elements 0 through 4, but there is an easier way. This is where a loop comes in handy.

9 While loops

This is new and shiny. You may want to read it more than once to completely understand it, or if you have programmed before then it is probably not necessary to read anything other than how while looks works.

Why use a while loop? Well, if you want to do something many times over then a while loop is probably what you are looking for. Heres a generic example of a while loop.

```
while(someBooleanCondition)
{
  //do stuff
}
```

Well that doesnt make much sense on its own. Heres a slightly more clear version of the same sort of thing.

```
while(loopyVariable < 5)
{
    //do stuff
    loopyVariable++;
}
```

This makes a little more sense. This will carry on executing whatever you put inside the body of the loop (Between the `and`) until `loopyVariable` is less than 5. There are a couple of nice new things you need to know about. Firstly, the `<` sign means less than. `>` is greater than and you may also need to know `>=` (Greater than or equal to) and `<=` (Less than or equal to). Secondly, the `loopyVariable++;` bit is new. This just means add one to the current value of `loopyVariable`. Its the same thing as writing `loopyVariable = loopyVariable + 1;` only in a shorthand form. Finally, you must note that the `loopyVariable` is incremented! If this didnt happen then the loop would carry on forever. Thats a BAD THING and you should not let it happen. Its also one of the reasons for loops are better than while loops for this type of thing (And we will do for loops after while loops.).

Going back to the example of the car with passengers, if we wanted to fill the car with people called bob then maybe this is how we would do it using the while loop.

```
while( i < 5)
{
    car [ i ] = "bob" ;
    i++;
}
```

This is how the computer would execute it (Assuming `i` is 0 and `car` is an array of Strings with length 5)

i is 0 which is less than 5, store "bob" into car[0], increase i from 0 to 1.
i is 1 which is less than 5, store "bob" into car[1], increase i from 1 to 2.
i is 2 which is less than 5, store "bob" into car[2], increase i from 2 to 3.
i is 3 which is less than 5, store "bob" into car[3], increase i from 3 to 4.
i is 4 which is less than 5, store "bob" into car[4], increase i from 4 to 5.
i is 5 which is not less than 5, skip over the stuff inside the while

So by the end of that while loop the array of Strings called car would be full up with people called "bob". However, a for loop is much nicer for this kind of thing...

10 For loops

Well you have seen how a while loop works, now for the better version of it. The for loop. The general way of declaring a for loop is slightly more complex its fine once you have done it a couple of times.

```
for (i=0; i < 5; i++)  
{  
    //do stuff here  
}
```

Erk. Looks kinda nasty. Its pretty easy really! The first bit, i=0, tells Java that you want to start counting at the number zero. The second bit is just like your boolean condition in the while loop. The last bit is what you want to do every time the first bit of the for loop gets executed. So, using the same example as the while loop, here it is in for loop form.

```
for (i=0; i < 5; i++)  
{  
    car [ i ] = "bob" ;  
}
```

There are times that you will want to use a for loop instead of a while loop and vice verca although you probably wont realise until you actually try writing some code and realise that one or the other is neater or does what you want better than the other.

11 Multidimensional Arrays

We covered arrays that had a single dimension. If you think of the array that we declared as being along a line, then multidimensional arrays are like squares, cubes and so on. This is possibly a little bit confusing, so at this point I'll just refer you to the excellent lecture slides of M. Capcarrere (Link: <http://www.cs.kent.ac.uk/teaching/05/modules/CO/3/20/Lectures/Lecture12.ppt>).

The slides in this link explain very well how a multidimensional array works and what you may want to use it for.

For completeness, heres how you declare a multidimensional array.

```
String multiDimensionalArrayName [] [] [] = new String [] [] []
```

As you can see, its fairly similar to the array declaration you are used to. The amount of pairs of square brackets used corresponds to how many dimensions you have for this variable.

Accessing elements in the array is just the same as with a single dimensional array. If we take the example of a 2 dimensional map, then the array entries could represent gridsquares. The example below shows accessing the grid refrence 0, 1.

```
String dataYouGetOut;  
String twoDMap [] [] = new String [] [];  
dataYouGetOut = twoDMap [0] [1];
```

Of course, that code does not really do too much by itself as dataYouGetOut would always return an empty String. Storing information inside of a multidimensional array similar in form to storing in a single dimensional array.

```
String twoDMap [] [] = new String [] [];  
twoDMap [0] [1] = "Beach";
```

As with single dimensional arrays, you must remember that multidimensional arrays start at the index zero.

12 The Java API

Perhaps the most important thing is the Java API. You may not know it but a lot of the things featured in this document come from the java API and I have not shown you how to import packages so if you just typed them out then they would not work.

The java API can be found at the Sun site, <http://java.sun.com/j2se/1.4.2/docs/api/>. There is a local copy of the Java API here too <http://www.cs.kent.ac.uk/java/api/>, although be warned that the local copy is Java 5 and not Java 1.4, which is what we are taught. Either way, looking at the Java API page you can see that it is split into three frames. Top left is the list of packages and clicking on one will cause the second frame to change to just the Classes in that package. Go ahead and click on java. Click on the java.util link, and all the Classes from java.util will be displayed.

Importing a package for use in a program is easy!

```
import java . packagename . subpackagename ;
```

An example of this is importing the ArrayList Class

```
import java.util.ArrayList;
```

If you click on one of the Classes from the package in the second frame then a larger view will open up in the main frame telling you lots of important information about this Class. This is called the Interface to the Class. Click on the Random link, and guess what. New topic.

13 java.util.Random

Say you wanted to add some randomness into your application, how would you do it? Well, by using the java.util.Random package! First of all, you will need to import the package into your project.

```
import java.util.Random;
```

This tells java that when its compiling it should include the code from the package java.util.Random into it. If you dont include this and try and write code with Random bits in, then it actually wont compile. Looking back at the Java API for Random, you can see that its constructor has no parameters (Or a seed with datatype long). So to create a nice Random object from which you can spawn Random numbers you can do the following.

```
Random itsARandomThing = new Random();
```

You can see quite clearly now that Random is an object (It uses the new keyword). To get random numbers out of this thing you just spawned you can use the method nextInt. As with when calling things not in the Class you are in now, the dot notation must be used.

```
int someRandomNumber = itsARandomThing.nextInt(6);
```

The parameter in brackets, if you read the API, is the upper limit value that the Random object can generate for you. So this would be almost perfect for a dice although the Random starts at 0 and ends at whatever is given in brackets. Easy enough to fix:

```
int someRandomNumber = 1 + (itsARandomThing.nextInt(5));
```

This way you can be sure that the value you will get back in someRandomNumber is between 1 and 6.

Random also generates other random things, if you check the API you will see that there are methods for getting Booleans, Doubles, Integers and other things out.

14 HashMaps

Despite their name, HashMaps have absolutely nothing in common with Cannabis. Which is a damn shame, because I am pretty certain that

Cannabis would make HashMaps a bit more interesting. Anyways, the idea of a HashMap is that you can store things into it and the HashMap will index it and so searching for stuff will be far faster than using something like an ArrayList and iterating through the ArrayList. If you like, the ArrayList is like a Cassette tape. You have to go through all the stuff beforehand to get to the track you want to listen to. The HashMap is more like a CD in that you can skip straight to elements in the HashMap.

Heres some code for declaring a new HashMap. Remember you will need to import the HashMap from java.util at the top of any Class that uses HashMaps.

```
HashMap exampleMap = new HashMap ();
```

And now heres adding a pair of values (Key first, then Value)

```
exampleMap.put("Twig" , "01227_874123");  
exampleMap.put("Someone" , "01227_641325");
```

And heres how you can get the value which is associated with the key "Twig"

```
String twigsFakeNumber = (String) exampleMap.get("Twig");
```

You must note that you need to typecast the reply from the HashMap when using get because it is like the ArrayList in that it forgets what type it was you put in to the HashMap.

The easiest way of thinking about HashMaps is to imagine a database with primary keys and a single field. The first value you enter is the key and the second is the data you want associated with the key. In the example above, the key is the name of a person and their telephone number is the data associated with the key. It would not make much sense to remember peoples telephone number to get their name!

One extremely important point is that, like ArrayLists, HashMaps store objects and not primitive types like char, int, boolean and so on.

15 More Fun With The Java API And Strings

Although we have been using Strings quite a bit in this little tutorial, I have not introduced the Java API for Strings. Strings have many methods that you can execute on them many of which you will need to explore and look at yourself to see exactly how they work. There is one VERY useful thing in the Java API for Strings which is the split method.

From the Javadoc:

```
public String [] split (String regex)
```

What the hecks that mean?! Well, it means that the function will return an array of Strings split by the regular expression named regex. Regular expressions are very, very powerful but I do not know how to use them so I will not try. Anyways, here is a short example of using the split method.

```
String exampleString = "This is just an example";
String [] splitBySpaces = exampleString.split(" ");
```

This is basically saying that wherever there is a space, split up the String and return an array with all the split up Strings inside of it. After being executed the array splitBySpaces would contain the elements:

Array Element	Contents
splitBySpaces[0]	"This
splitBySpaces[1]	"is"
splitBySpaces[2]	"just"
splitBySpaces[3]	"an"
splitBySpaces[4]	"example"

There are some more examples of this in the lecture slides for CO320, look for lecture 14 (Week 9, Monday 21st November).

16 Writing Good Documentation

You have no doubt looked at the Java API documentation and have seen that all the Classes follow the same layout with descriptions of the Class, its methods and any attributes that it might have. Would it not be great to be able to have those for Classes that you write? Well you can! There is a program called javadoc which generates documentation for you based on your classes and comments contained within the Class.

To start including javadoc comments, you will need to do the following:

```
/** This Class is designed to show javadoc comments
 * @author Tom Carlson
 * @version 09/01/2005
 public Class someClassHere
 {
     someClassHere ()
     {
     }
 }
 }
```

That's just a very simple example of class documentation with an author and version as well as a short description of the Class. Methods should also have documentation.

```
/**
 * An example of a method – replace this comment with your own
 *
 * @param y a sample parameter for a method
 * @return the sum of x and y
 */
```

As you can see there are keywords with the at symbol before them and then data that you want in this keyword.

One major point to note about javadoc and commenting in general is that it is completely futile and useless to comment after writing a piece of code in a method at the end of a project. Do it as you go along and you will definitely save yourself a lot of trouble. Get into the habit of labelling up odd bits of code that you would not know what it does immediately with comments after the code and in the method javadoc bit too.

17 Public and Private

So far, you know that you can have public and private methods and variables. But you do not really know why you would want to use one over the other yet. Public Class Variables can be accessed and modified outside of the Class by other Classes, public methods can be called from outside of the Class by an object of that Class. On the other hand, private methods and variables are somewhat more uptight about who uses and abuses them and will only let the Class they were created in call them. They are not visible from the outside. If you remember about the Class interface in the javadoc, none of the private methods internal to the Classes are shown.

There is a very good reason for this! This tactic of Information hiding hides what the user does not need to or is not allowed to know. After all, you wrote the Class and you dont want some other evil Class to come in and muck around with your Variables! Now might be a good time to introduce coupling.

No, not the BBC TV programme that was on a while ago. Coupling refers to how tightly integrated one Class is to another. If you make a small change to one class, it should not utterly destroy the workings of another class. This is called Loose coupling and according to Matthieus lecture slides ‘Should be aimed at, even if we write both the used and the user class’.

So why use private methods at all? We have seen that they can not be called outside of the Class that created them, thats all well and good. Its part of a tactic in programming called divide and conquer. If you have some supercomplex method that you are writing it might be easier to break it down into logical steps and have each step in its own private method. This also makes programs somewhat easier to debug and I highly recommend that you break down any huge long methods into something shorter (If you are having difficulty that is. Otherwise, do so anyway because it makes code easier to read!).

You can also put things as Private methods when they would be dangerous for other Classes to access (Imagine a DeleteEverythingEver method that was public - a nasty evil Class could call this and inadvertantly delete everything!)

18 More Keywords

There are some more very important keywords that you need to know about in java.

The static keyword can be used to share a variable between all objects of the same class. This can be useful to have a field like `numberOfObjectsCreated` and then in the constructor for the Class increment this by one. It is easy to see that without the static keyword the value would always be one. (And if its not, Mr. Capcarre's slides are very good!).

Static variables save on memory space because there is not a new variable having to be stored every time a new object is created from the class. The static keyword allows information to be shared between objects and is useful to represent information which is the same for all objects of the same Class.

```
private static int numberOfObjects;
```

The final keyword is used to define constants. So if you wanted a numerical value for pi but did not want to use the inbuilt pi thing in java for some reason you could define pi:

```
private final double mmmPi = 3.14;
```

The final keyword means that the value will never be changed (If you try to change it in code then the java compiler will just eat your brains. It is a bug that never got ironed out.)

It may also be useful to have the constant the same for all objects of the same class. keywords can be combined and you can have static final variables (Constants that are the same for all objects in a class)

```
private static final double mmmPi = 3.14;
```