Object
Oriented programming

A guide for the beginner -
from Modula-2 to Java.

Introduction

This tutorial aims to teach you the basics of Object Oriented Programming. It is not tailored to any one specific language, although examples in C++ and Java will be given. Examples using Modula-2 will also be used, although obviously not for the purposes of demonstrating OOP techniques!

There are many good tutorials out there that deal with this topic in a more thorough way. I just intend to provide a small outline of the topic. The two best references that I can provide are:

- The Java Tutorial - available from the Java Documentation Page.
- The Java White Paper - Chapter 3

Both of these concentrate on Java, but the tutorials on objects etc. are valid for any OOP language. You should download these, and use up any remaining print quota printing them out! They are both very useful resources.

Many languages claim to have "Support for Object Technologies!", while in reality, very few actually fully support all the OOP principles. Java is probably the only modern language to fully support object-oriented principles. C++ is a language that is very similar to C, with OOP spliced in. It is not recommended for learning to use OOP techniques. I would advise anyone serious about trying to learn OOP techniques to download the Java Development Kit and try out this language.

Probably the most important thing I would like you to take away from this tutorial is the idea that programming in an object oriented language is more than just learning new functions, syntax, etc. For example, when we studied COBOL it was another procedural language - we could apply the knowledge we had of Modula-2 to COBOL quite easily. This is impossible when dealing with OO languages, since OOP is more than learning a new language; it requires *a new way of thinking*. We must no longer think in terms of data structures - we must think in terms of *objects*.

Objects

Objects are the central idea behind OOP. The idea is quite simple.

An *object* is a bundle of *variables* and related *methods*.

A *method* is similar to a procedure; we'll come back to these later.

The basic idea behind an object is that of *simulation*. Most programs are written with very little reference to the real world objects the program is designed to work with; in object oriented methodology, a program should be written to simulate the states and activities of real world objects. This means that apart from looking at data structures when modelling an object, we must also look at *methods* associated with that object, in other words, functions that modify the object attributes.

A few examples should help explain this concept. First, we turn to one of my favourite pastimes...

Drink!

Say we want to write a program about a pint of beer. If we were writing this program in Modula-2, we could write something like this:

```
TYPE BeerType = RECORD
                    BeerName:       STRING;
                    VolumeInPints:  REAL;
                    Colour:         ColourType;
                    Proof:          REAL;
                    PintsNeededToGetYouFull: CARDINAL;
                    ...
                END;
```

Now lets say we want to initialise a pint of beer, and take a sip from it. In Modula-2, we might code this as:

```
VAR MyPint: BeerType;

BEGIN
    ...
    (* Initialise (i.e. buy) a pint: *)
    MyPint.BeerName := "Harp";
    MyPint.VolumeInPints := 1.00;
    ...
    ...
    (* Take a sip *)
    MyPint.VolumeInPints := MyPint.VolumeInPints - 0.1;
    ...
```

We have constructed this entire model based entirely on *data types*, that is we defined `BeerType` as a record structure, and gave that structure various names, e.g. `Name`. This is the norm for procedural programming.

This is however, **not** how we look at things when we want to program using objects. If you remember how we defined an object at the start of this section, you will remember that we must not only deal with data types, but we must also deal with *methods*.

> **A *method* is an operation which can modify an objects behaviour. In other words, it is something that will change an object by manipulating it's variables.**

This means that when we take a real world object, in this case a pint of beer, when we want to model it using computational objects, we not only look at the data structure that it consists of, but also *all possible operations that we might want to perform on that data*. For our example, we should also define the following methods associated with the `BeerType` object:

- `InitialiseBeer` - this should allow us to give our beer a name, a volume, etc.
- `GetVolume` - to see how much beer we have left!
- `Take_A_Sip` - for lunchtime pints...
- `Take_A_Gulp` - for Lavery's pints...
- `Sink_Pint` - for post exam pints...

There are loads more methods we could define - we might want a function `GetBeerName` to help us order another pint for example. Now, some definitions. An *object variable* is a single variable from an object's data structure, for example `BeerName` is one of `BeerType`'s object variables. Now the important bit from this section:

Only an object's methods can modify it's variables

There are a few exceptions, but we'll cover them much later. What this means in our example is that unlike the Modula code, we cannot directly modify `BeerType`'s variables - we cannot set `BeerName` to "Tennents" directly. We must use the object's methods to do this. In practice, what this means is that we must think very carefully when we define methods. Say in the above example we discover when writing the main program that we need to be able to take a drink of arbitrary size; we cannot do this with the above definition, we can only take a sip, a gulp etc. We must go back and define a new method associated with `BeerType`, say `Take_Drink` which will take a parameter representing the amount of beer we wish to drink.

Another Example

We'll now deal with a real-life example which will help us understand some more object concepts. We will design an object to emulate a counter.

A counter is a variable in a program that is used to hold a value. If you don't know that then you shouldn't be reading this! To make things very simple, we'll assume that our counter has only three operations associated with it:

- Initialising the counter to a value
- Incrementing the counter by one
- Getting the current value of the counter

So, when we come to implement the above using objects we will define three methods that do the above.

You may be thinking that we could implement this very simply in Modula-2 using definition and implementation modules obtaining the same results as if we used an object oriented language. Well, we *nearly* can:

```
DEFINITION MODULE Counter;

PROCEDURE InitialiseCounter(InitialValue: INTEGER);

PROCEDURE IncrementCounter;

PROCEDURE GetCounterValue(): INTEGER;

END Counter.


IMPLEMENTATION MODULE Counter;

VAR MyCounter: INTEGER;

PROCEDURE InitialiseCounter(InitialValue: INTEGER);
BEGIN
    MyCounter := InitialValue;
END InitialiseCounter;

PROCEDURE IncrementCounter;
BEGIN
    INC(MyCounter);
END IncrementCounter;

PROCEDURE GetCounterValue(): INTEGER;

BEGIN
```

```
    RETURN MyCounter;
END GetCounterValue;

BEGIN
    MyCounter := 0;
END Counter.
```

Because Modula-2 is not object oriented, this will only satisfy *one* of the requirements for an object oriented language - *encapsulation*. This has been covered before; it simply means that we have implemented *information hiding*, i.e. we cannot directly access `MyCounter` from any module that imports `Counter`. But being object oriented means a lot more than just encapsulation, as we'll see next...

Classes

Say we wanted to extend the counter example discussed previously. Perhaps in our Modula-2 program we need three counters. We could define an array of `MyCounter` and work through that. Or say we needed up to 1000 counters. Then we could also declare an array, but that would waste a lot of memory if we only used a few counters. Perhaps if we needed an infinite amount of counters we could put them in a linked list and allocate memory as required.

The point of all this is that we are now talking in terms of data structures; all of the above discussion has nothing to do with the behaviour of the counter itself. When programming with objects we can ignore anything not directly concerning the behaviour or state of an object; we instead turn our attention to *classes*.

A class is a blueprint for an object.

What this basically means is that we provide a blueprint, or an outline of an object. This blueprint is valid whether we have one or one thousand such objects. A class **does not** represent an object; it represents all the information a typical object should have as well as all the methods it should have. A class can be considered to be an extremely extended `TYPE` declaration, since not only are variables held but methods too.

C++

As an example, lets give the C++ class definition for our counter object.

```
class Counter  {
    private:
        int MyCounter;

    public:
        Counter()  {
            MyCounter = 0;
        }

        void InitialiseCounter(int value)  {
            MyCounter = value;
        }

        void IncrementCounter(void)  {
            MyCounter++;
        }

        int GetCounterValue(void)  {
            return (MyCounter);
        }
}
```

So, a lot to go through for this little example. You really need to understand the fundamentals of C before the example will make any sense.

- In the `private` section, all the object's variables should be placed. These define the state of the object. As the name suggests, the variables are going to be private, that is they cannot be accessed from outside the class declaration. This is encapsulation.
- The `public` section contains all the object's methods. These methods, as the name suggests, can be accessed outside the class declaration. The methods are the only means of communication with the object.
- The methods are implemented as C functions or procedures; the three methods should be easy to understand.
- All class definitions must also have one public method that has the same name as the class itself, in this case `Counter`. This method is called the *class constructor*, and will be explained soon.
- Functions and procedures can also be placed in the `private` section; these will not be accessible to the outside world but only within the class declaration. This can be used to provide support routines to the public routines.

Instantiation

This is an awful big word for a powerfully simple concept. All we have done so far is to create a class, i.e. a specification for our object; we have not created our object yet. To create an object that simulates a counter in C++ then, all we have to do is declare in our main program:

```
Counter i;
```

Although this seems just like an ordinary variable declaration, this is much more. The variable `i` now represents *an instance of the counter type*; a counter object. We can now communicate with this object by calling it's methods, for example we can set the counter to the value '50' by calling `i.InitialiseCounter(50);`. We can increment the counter - `i.IncrementCounter();` - and we can get the counter value - `value = i.GetCounterValue();`.

When we first instantiate an object (i.e. when we first declare, or create it), the *class constructor* is called. The class constructor is a method with the same name as the class definition. This method should contain any start-up code for the object; any initialisation of object variables should appear within this method. In the counter example, whenever we create a new counter object, the first thing that happens to the object is that the variable `MyCounter` is initialised to zero.

Remember the question posed at the very start? The power of objects starts to kick in now. Say we require another counter within our program. All we have to do is declare a new object, say:

```
Counter j;
```

The new counter object will have nothing to do with the previous object. What this means is that `i` and `j` are two distinct objects, each with their own separate values. We can increment them independently, for example. Should we need 1000 counter objects we could declare an array of counter objects:

```
Counter loads[1000];
```

and then increment one of them using a call such as `loads[321].InitialiseCounter();`.

Java

The equivalent Java class definition for the counter example follows. It is remarkably similar to the C++ definition, and differs only in syntax.

```
class Counter extends Object  {

    private int MyCounter;

    Counter()  {
        MyCounter = 0;
    }

    public void InitialiseCounter(int value)  {
        MyCounter = value;
    }

    public void IncrementCounter(void)  {
        MyCounter++;
    }

    public int GetCounterValue(void)  {
        return (MyCounter);
    }
}
```

A few brief notes about the differences:

- All new classes must be defined with the extension `extends Object`. This defines the *superclass*; this will be dealt with in the next section.
- There are no `public` or `private` sections, instead all variables and methods are prefixed with the appropriate qualifier.
- The class constructor definition remains the same.

Instantiating objects in Java is slightly different; the designers knew that the C++ method of declaring a new object was far too similar to how new variables are declared, so objects are declared differently:

```
Counter i;

i = new Counter();
```

Basically we define a variable to reference the object in the first line. Then we actually create an instance of the object by a call to `new` in the second line. Accessing object methods is done in the exact same way in Java as in C++.

So which...?

A quick diversion from OOP here! At this point you might think it doesn't matter whether you use C++ or Java, they both implement object oriented technology. Well, C++ can be used to design programs *without* implementing any objects; C++ can be used as an extended C. In Java, you *must* implement any non-trivial program using objects. This is because Java has no support for structures (record types) or pointers; all these must be replaced by object variables and methods. So, if you are using Java, you need to understand object methodology; with C++ this is optional.

Basically, both these languages have hundreds of other features that I don't have time even to begin to explain; as long as you have a basic understanding of object technologies and the C language, you should find both rather easy to learn.

Why Bother?

The process of designing and programming objects seems very cumbersome, so why bother? Well, it's difficult to see from such a small example, but for larger projects, OOP techniques allow unlimited flexibility. Objects are used because:

- Encapsulation; in our example we cannot alter the value of the counter other than by incrementing it or setting it to a initial value. This reduces possible bugs.
- Modularity; Different programmers or teams can work on different independent objects.
- Inheritance; this is covered in the next section.

Basically, objects provide a secure and easily upgradable path for program developers. Already, a considerable amount of developers are moving from normal procedural design and embracing object oriented technology.

The next section should be easy to follow if you understood this one! By the way, the reason the next few examples are only in Java is because I don't know enough about C++ to program them!

Inheritance

Another big word for a simple concept. To help explain this, we'll go back to our beer example. Say we want to define a new class to represent a pint of an imported French beer. This class would have all the variables and methods of the normal beer class, but it would have the following additional information:

- A variable representing the price of the beer
- Two methods to set and get the price of the beer

(We need this information because we are students; everyone knows the price of Harp, but we would like to know the price of this expensive beer before we order it!)

It would be rather tedious to define a new class, `FrenchBeerType` which had all the variables and methods of `BeerType` plus a few more. Instead, we can define `FrenchBeerType` to be a *subclass* of `BeerType`.

> **A subclass is a class definition which takes functionality from a previous class definition.**

What this means is that we only define the *additional information* that the `FrenchBeerType` class has.

Informally then, we would create a new class, `FrenchBeerType`, and tell our compiler that it is a subclass of `BeerType`. In the class definition, we would include *only* the following information:

- A variable `BeerPrice`
- A method `SetBeerPrice`
- A method `GetBeerPrice`

We do not need to include any information about `BeerName` for example; all this is automatically *inherited*. This means that `FrenchBeerType` has all the attributes of `BeerType` plus a few additional ones. All this talk of beer is making me mad for a pint...

Counters, Counters, Counters...

Back to the counter example then! The counter we had in the last section is fine for most counting purposes. But say in a program we require a counter that can not only be incremented, but can be decremented too. Since this new counter is so similar in behaviour to our previous counter, it would be mad to define a brand new class with everything that `Counter` has plus a new method. Instead, we'll define a new class `ReverseCounter` that is a subclass of `Counter`. We'll do this in Java.

```
class ReverseCounter extends Counter
{
    public void DecrementCounter(void)  {
        MyCounter--;
    }
}
```

The `extends` clause indicates the *superclass* of a class definition. A superclass is the "parent" of a subclass; in our beer analogy, `BeerType` is the superclass of `FrenchBeerType`, so if we were defining this in Java we would use `class FrenchBeerType extends BeerType`. Basically, we are just saying that we want `ReverseCounter` to be a subclass of `Counter`. When we define a brand new class that is not a subclass of anything (as we did when we defined `Counter`) we use the superclass `Object` to indicate we want the default superclass.

We have defined `ReverseCounter` to be a subclass of `Counter`. This means that if we instantiate a `ReverseCounter` object, we can use any method that the class `Counter` provided, as well as the new methods provided. For example, if `i` is an object of the `ReverseCounter` class, then we can both increment it and decrement it; `i.IncrementCounter();` and `i.DecrementCounter;` respectively.

Inheritance is a powerful tool. Unlike our simple example, inheritance can be passed on from generation to generation; we could define a class `SuperDuperReverseCounter` for example, that is a subclass of `ReverseCounter` which could provide added variables or methods.

Bugs, bugs, bugs...

If you tried to compile the above example and found it wasn't compiling, don't worry! There is a semi-deliberate mistake left in the code, which I am very usefully going to use to stress a point.

When defining a class you must consider any possible subclass.

When we defined the `Counter` class we didn't even know what a subclass was, so we could be forgiven for breaking this rule then, but not from now on! If we go back to how the class was defined:

```
class Counter extends Object  {
    private int MyCounter;

    ...
    ...
}
```

We can see that the variable `MyCounter` is defined to be of type `private`. In Java, this means that the variable becomes very, very private indeed; in fact, it is only accessible from inside the class from which it is defined. It is not available to any other class, including it's subclasses. So when we reference `MyCounter` from inside `ReverseCounter` the Java compiler will kick up a fuss, since we are outside the scope of the variable.

So, we should have realised at the time of writing the `Counter` class that subclasses might need to get at this variable too. To fix this, all we have to do is change the qualifier of `MyCounter` to:

```
    protected int MyCounter;
```

A variable with a `protected` qualifier means that it can only be accessed from within the class in which it is defined, as well as all subclasses of this class. This seems appropriate for our purposes.

That's It!

That's about as far as I can go without delving too deep into any particular programming language. What I recommend is that you download the Java tutorial (see the Introduction) and the Java Development Kit if you have a computer of your own, and try some Java! Failing that, get some rest in, because if you need to use OOP on your year out, it isn't going to be too easy!
Seriously, good luck on your year out, and let me know if this tutorial was of any use to you!

References

Various sources were used to compile this tutorial. These include:

The Java Tutorial, by Mary Campione and Kathy Walrath
Most of the technical information was checked through using this book. Some definitions were also used.

The Java White Paper
Mainly for ideas, some definitions.

Stacks & Heaps, Computer Shopper, May '93, by Mike James
Found this article by accident! Outline of the counter example is given here, to which much has been added.

Comments on any part of this tutorial are most welcome. Most useful would be suggestions for improvements, as well as bug reports. I hope you find this a useful resource, let me know if you do.

Adrian O' Neill – aon@bigfoot.com
http://www.quiver.freeserve.co.uk
5 June 1999