PHAS3459: Scientific Programming Using Object Oriented Languages

Module 1: Introduction and Basic Concepts

Dr. B. Waugh and Dr. S. Jolly
Department of Physics & Astronomy, UCL

Contents

1. Aims of Module 1

- Become familiar with using the Eclipse package to develop basic Java programs.
- Understand some of the basic concepts behind object-oriented programming.
- Be able to write a simple Java program.
- Understand the basic elements of the Java programming language: data types, functions, algorithm control.

2. What is Object Oriented Programming?

First, recall the basic aim of a computer program. It is to:

- perform calculations,
- read, manipulate and store data, or
- simulate some aspect of real-life (sometimes called “abstraction”).

In order to achieve this, all programs have *data* (to describe/store the state of a system) and *functions* (to manipulate/change the state of the system). How the data and functions are organised becomes particularly important when we start to develop complex programs, and this is where object-oriented programming comes in. Theoretically, there is nothing that an object-oriented (“OO”) computer program (for example, written in Java) can compute that a non-object oriented computer program (for example, written in Fortran or Basic or C) cannot. It's just that programs written in an OO language lend themselves to being organised in much more natural and maintainable ways, enabling much more complex programming tasks to be readily undertaken.

2.1 A Simple Example

Suppose we wrote a simple program to simulate an oscilloscope. Amongst other things, the program would probably contain the following elements:

- *data*: input voltage (Volts). Variable = `iv`.
- *data*: scope calibrations (cm/Volt). Variable = `sc`.
- *data*: signal size on scope (cm). Variable = `ss`.
- *function*: calculate the ouput signal. `ss = sc*iv`.

This would be an easy program to build and maintain. But what if the situation became more complex, for example:

- The scope calibration is not a constant, but depends on the input voltage itself (the scope is “non-linear”).
- We want to model other aspects of oscilloscope behaviour — for example, how the scope responds to rapidly changing inputs.
- We want to be able to simulate oscilloscopes with different properties in the same program.
- This code is embedded in a much larger program that might also use variables called `iv`, `sc` or `ss`.

What we need is a stronger coupling between data and functions. We need program elements that more naturally correspond to real-world elements.

2.2 Objects

Objects are bundles of *data* and the procedures, or *functions*[1], that act on that data. The data and functions define the object and are not just loosely connected pieces of code. Object-oriented programs define objects and utilise them by passing messages between them, as indicated schematically in Fig. 1.1.

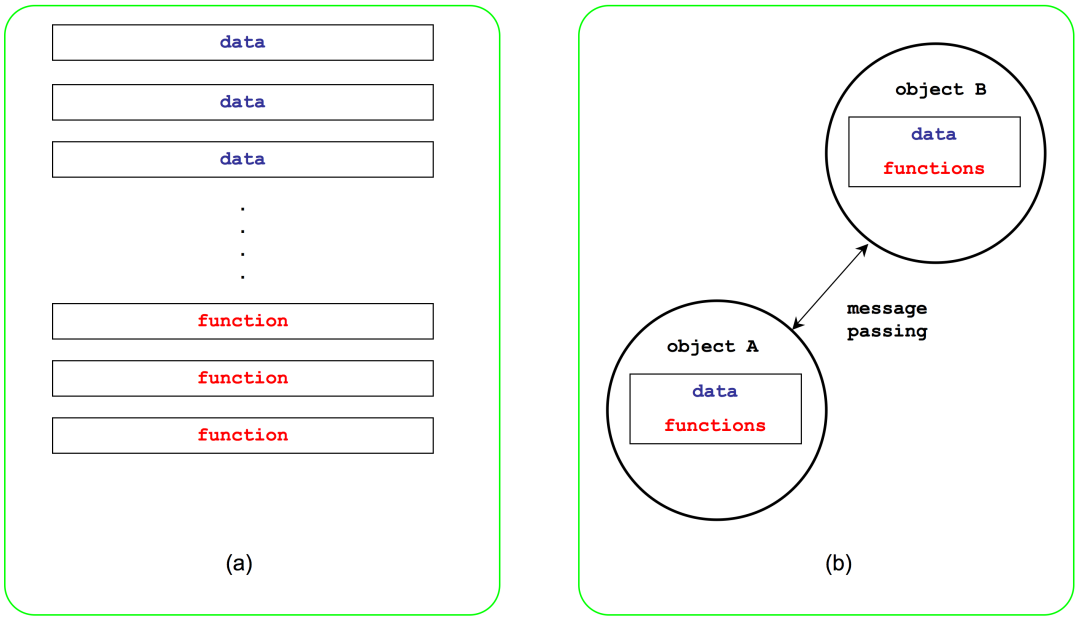

Figure 1.1: A highly schematic picture of the logical construction of a non-OO program (a) and an object-oriented program (b). In (a) the data are organised separately from the functions; program execution proceeds via functions invoking other functions, all of which may access the data. In (b) the data and functions are located together in objects; program execution proceeds through objects passing messages to one another. See text for more details.

There are numerous benefits of using objects, including:

- We can define our own data types (called *classes*). We aren't limited to simple `integer`, `real`, ... variables. Instead we can construct programs that use variables of type `oscilloscope`, `electron`, `FourVector`, etc.
- This in turn makes it much easier to model real-life situations by having program elements that correspond more closely to real-world entities ("abstraction").
- Code is more easily re-used. For example, I can take an `electron` class initially developed for the simulation of a particle detector and I might be able to re-use it in the simulation of a fusion experiment. The properties and behaviour of electrons are invariant and so a well developed electron class can find a place in many different applications.
- Large programs are more scalable and maintainable. This is because the development and maintenance of the participating objects can take place to some

extent independent of the program as a whole. This requires good program architecture. Indeed the majority of the object-oriented programmer's time should be spent in *design* rather than *implementation*.

2.3 Some Concepts of Object Oriented Programming

Here are some of the most important aspects of OO programming that you will encounter:

1. Encapsulation. The data and internal workings are "hidden" inside the object. The object is to some extent a black-box. Other objects interact with the given object by passing messages through a well defined *interface*. For example, there may be an `account` object in a piece of financial software. Another object can call a function of that object called `calculateInterest` and expect to be returned to it the interest accrued on the account. The exact details of how the interest is calculated are, however, unimportant to all users apart from the developer of the `account` class itself.
2. Inheritance. An `apple` is a type of `fruit`. An `electron` is a type of `lepton`. Object-oriented languages provide a formal structure for representing these "is-a-type-of" or "is-a-kind-of" relationships. An `electron` type definition can *extend* the `lepton` type definition. Exploiting inheritance properly is critical to good OO program design.
3. Polymorphism. Literally, "one entity has many-forms". A different meaning or usage can be assigned to something depending on the context. For example a different function (but with the same name) can be invoked, depending on the type of the variables passed to it, or two objects may respond in different ways to the same message from another object.

These OO concepts have been around since the 1960's but their use has only become widespread in the last 20 years or so. OO languages include C++, Ruby, Eiffel and SmallTalk. Java was released in 1995. Java and C++ are, by far, the languages most widely used in scientific and business applications.

We will meet these OO concepts again several times in the rest of the course, but first we have to get to grips with the nuts and bolts of the Java programming language.

---

[1] We frequently use the term *method* to denote a function associated with an object or a class. ↩

---

3. Your First Java Program

3.1 Part 1: Skeleton Code

The following code fragment represents the simplest possible executable Java program. What it all means will become clearer as the course goes on.

```
package module1;

public class MainProgram{
  public static void main (String args[]){

  // PLACE YOUR CODE HERE

  }
}
```

A few things to note already:

- Code inserted in the `main` method (i.e. at the location indicated by the comment "`PLACE YOUR CODE HERE`") gets executed when you "run" the code.
- Comment lines (i.e. those lines that will be ignored by the Java compiler and will not be executed) start with a double forward slash "`//`". It is important to comment your code well: concisely and clearly (this is dealt with in more detail in the section on Writing Readable Code).
- Every opening curly bracket "`{`" has an associated closing curly bracket "`}`".

3.2 Part 2: Make It Do Something

Add the following line after the "`package`" line and before the "`public class MainProgram`" line:

```
import java.util.Date;
```

This makes sure that the program "knows about" the built-in Java `Date` class. Then, add these lines in the "`main`" method:

```
System.out.println("Program starting");
Date myDate = new Date();
System.out.println(myDate);
System.out.println("Program finished");
```

At this point your code should look something like this:

```
package module1;
import java.util.Date;

public class MainProgram{
  public static void main (String args[]){

    System.out.println("Program starting");
    Date myDate = new Date();
    System.out.println(myDate);
    System.out.println("Program finished");

  }
}
```

3.3 Part 3: Run The Code

When you press the Eclipse "Run" button, Java tries to execute all the code that is between the curly brackets after the statement:

```
public static void main (String args[]){
```

If your code does not contain this line then it can only be compiled but cannot be run/executed. The compiled code can however be invoked/called from code that is within the "`main`" method. Without the "`main`" function an attempt to run the code will produce this error:

```
java.lang.NoSuchMethodError: main
Exception in thread "main"
```

Try it: remove the "`main`" method and attempt to re-run the code.

3.4 Part 4: Printing to the Screen

To develop your programming skills you need to be able to print results to the screen. We will use "`System.out.println`" as in the following code fragment. Note that this is already OO code but we will defer exactly what it means until Module 2:

```
int x = 100; int y = 200;

// Print the value of x
// to the screen
System.out.println(x);

// Prefix the printing of the
// value with some descriptive
// information:
System.out.println("The value of x is "+x);

// Note the "" for any words.
// The + means append to the
// output going to the screen
System.out.println(" x = "+x+" and y = "+y);

// We can have any number of
// things appended using a '+'
```

Try putting this code in your "`main`" method and observe the resulting screen output.

4. Basic Java Language Elements

For the remainder of this module, we will learn about the basic Java language elements. Note that these are common (in one form or another) to essentially all programming languages, both OO and non-OO. We will give examples using Java syntax, but the same constructs in C++ would be very nearly identical.

4.1 Built-in Types and Variables

Built-in types are “intrinsic” to the language. What this means is that the compiler already understands how to handle them, which functions can be applied to them etc.without these having to be specified by the programmer. Some important things to remember:

- Built-in types are the basic entities in which we store the state of the program.
- They have built-in functions/operations associated with them, for example `+`, `-`, `*`, `/`, `=`
- Different types are needed in different programming situations. They are stored differently in the computer's memory and the computer needs to invoke the correct detailed instructions for operations on a given type.
- Variables must be defined before they are used.

4.1.1 Integer and Floating Point Numbers

Suppose we need to write code to solve the following equations:

$$E^2 = m^2c^4 + p^2c^2$$

$$\sin\phi = n\lambda$$

Then we need to represent E, m, c, p, ϕ and λ as real numbers. However we might choose to represent n as an integer, to ensure that it only takes on whole-number values. Here is an example code fragment showing how built-in types are declared:

```
// built-in variable
// type declaration
int n;   // an integer
float e; // a 32 bit real number
float E; // "e" and "E" are
         // different variables

// Remember - Java is case sensitive!

// meaningful names make
// code easier to read
float mass, momentum;
```

Here is a list of built-in integer and floating point data types. Also indicated are the numerical ranges available to variables of the different types:

```
// built-in integer data types

            // range     ([MIN,MAX])
byte   ib;  // 8-bits    ([-128,+127])
short  is;  // 16-bits   ([-32768,+32767])
int    ii;  // 32-bits   ([~-2.1e9,+2.1e9])
long   il;  // 64-bits   ([~-9e18,+9e18])

// built-in floating point (real) data types

            // range     +/- ([SMALLEST,LARGEST])
float  fw;  // 32-bits   +/- ([1.4e-45,3.4e38])
double dw;  // 64-bits   +/- ([4.9e-324,1.8e308])
```

The type that you use depends on the application. There are also speed and storage considerations depending on the type of computer that the code will be run on. For this course, the suggestion is:

- to represent integers, use `int`, unless dealing with numbers that may be large enough to require a `long`.
- to represent real numbers, use `double`, unless there are reasons of performance or limited storage in which case `float` may be more appropriate.

Note that you can only perform arithmetic operations on `int`, `long`, `float` and `double`. By contrast `byte` and `short` are generally only used to store or transmit data, for example from a file.

4.1.2 Assigning Values to Built-in Types

We use the assignment operator "=" as in the following code fragment:

```
// The assignment operator = assigns
// a value to a variable, or assigns
// the value of one variable to another

double xd = 100.234;
int    ix = 100;
int    ia = ix;
```

One particular pitfall to be aware of, is attempting to assign a value outside the range for that type. See what happens when you print out the values of the variables assigned in the following code fragment:

```
int ix = 1234567890;
int iy = 10*ix;     // What happens?

long ixl = 1234567890;
long iyl = 10*ixl;  // What happens?
```

4.1.3 Type Conversions and Casts

Another subtle point to be aware of is exactly how *conversions* between different types take place. We often want to convert between the built-in types, for example if we realise we need a variable with a larger number of bits to precisely represent our value within some calculation, or if we are storing data and wish to save space. There are therefore two types of conversion:

- A *widening* conversion increases the range of possible values, e.g. from `int` to `long` or `float`.
- A *narrowing* conversion reduces the range of possible values, e.g. from `double` to `float` or `int`.

Widening is simple and is done automatically when we compile the code:

```
byte x = 100;
// size = 8 bits
// in memory: x = 01100100

short y = 0;
// size = 16 bits
// in memory: y = 0000000000000000

y = x;
// in memory: y = 0000000001100100
```

However, you can still lose precision in some widening conversions, as illustrated in the following example:

```
int ix = 123456789;
int iy = 123456788;
System.out.println(ix-iy);

// Widening conversions
float fx = ix;
float fy = iy;

System.out.println(fx-fy);
// What happens when you make
// these doubles instead of floats?
```

By contrast, narrowing must be done explicitly (via a so-called *cast* operation). The cast operation has the general form:

```
type x = (type) y;
```

Take care, strange things can happen! You can see this in the following code fragment:

```
short x = 1000;
// size = 16 bits
// in memory: x = 0000001111101000

byte y;
// size = 8 bits
// in memory: y = 00000000

y = x;
// ILLEGAL: COMPILE ERROR (try it and see)

y = (byte) x;
// in memory: y = 11101000 (= -24!)
```

Note the compilation error if you attempt to assign the value of `x` to `y` without an explicit cast. Importantly, note that even though the cast on the last line is a perfectly legitimate piece of code that will compile and run without error, it nevertheless produces an unexpected result[2].

When we are using numbers explicitly (i.e. typing them into the code — we say they are "literals") then we must again be careful as evidenced in the following code fragment. To understand the code you need to remember some assumptions made by Java:

- A number with no decimal point is assumed to be an `int`.
- A number with a decimal point is assumed to be a `double`.

```
// This is not allowed since it is
// an attempt to do a "narrowing"
// without a cast, since
// LHS = float; RHS = double
float x = 100.0;

// This is allowed:
double xd = 100.0;

// We can explicitly "cast" a
// literal in two ways:

// State the (cast) in front
// of the number:
float xx = (float) 100.0;

// Add f to the end of the number:
float xy = 100.0f;

// For integers we do not have the
// problem if we only use int &
// long since a literal is an int:
int  ix = 100;  // OK
long iy = 100;  // OK: widening
long iz = 100L; // OK: specify long by
                //   adding L on the end
```

4.1.4 Other Built-in Data Types

The `char` data type:

- represents a single (unicode) character, e.g. ‘a’, ‘f’, ‘?’, ‘3’, etc.,
- is stored internally as a number in the range 0-65535 (an unsigned 16-bit integer). Therefore there are 65,535 single characters that the `char` type can hold. This can be used to encode every possible single character from all languages, not just those used in English.

Later, we will see how to use the Java string class to manipulate multi-character “words”.

The `boolean` data type:

- represents the logical condition “true” or “false”,
- is stored internally as a single bit: 0 (false), 1 (true).

The following code fragment illustrates some of their methods of assignment:

```
char SingleCharacter = 'b';

// This is not allowed:
char TwoCharacters = 'bc';

// A boolean variable can only be
// assigned a value of true or false:
boolean SystemIsOK = false;
boolean SystemIsBroken = true;
```

4.1.5 Built-in Operators

The built-in types have associated built-in operators. Not all operators can be used with all types. Arithmetic operators include:

```
// int, long, float and double types
// support all the standard
// arithmetic operations.
//
// (short, byte do not unless
// they are converted to int)

// addition (+),
// subtraction (-)
int ia,ib,ic,id;
ib = 2;
id = 5;
ia = ib + 4;
ic = id - 2;

// multiplication (*),
// division (/),
// remainder (%)
ia = ib / 2;
ic = id * 3;

// increment (++),
// decrement (--)
ia++;  // same as ia = ia + 1;
ib--;  // same as ib = ib - 1;
```

Note that the **order of precedence** for arithmetic operators is the familiar one: multiplication and division (including modulus) are done before addition and subtraction, otherwise the ordering is from left to right.

Comparison operators yield a `boolean` type as the result:

```
// comparison
// == : is equal to
// != : is not equal to
//  < : is less than
// <= : is less than or equal to
//  > : is greater than
// >= : is greater than or equal to

boolean isSameAs;
int ia = 4; int ib = 4; int ic = 5;
isSameAs = (ia == ib); // True;
isSameAs = (ia == ic); // False;
isSamaAs = (ia != ic); // True;
```

Bitwise operators operate on both `integer` and `boolean` types. They operate on the individual bits which, for `boolean` variables represented by a single bit, is straightforward to understand. For `integer` variables you need to be familiar with the binary representation of numbers (e.g. 3 in binary is “11”; 15 is “1111”; 8 is “100” etc.):

```
// AND (&), OR (|), NOT (~),
// XOR (^), left shift (<<),
// signed right shift (>>),
// unsigned right shift (>>>)

int ia = 60;   // 0011 1100 in binary
int ib = 13;   // 0000 1101 in binary
int ic;
ic = ia | ib; // Sets value of ic = 61
ic = ia & ib; // Sets value of ic = 12
ic = ia ^ ib; // Sets value of ic = 49

ic = ia << 2; // Sets value of ic = 240

boolean ab = true;
boolean bc = false;
boolean cd = ab | bc; // Sets cd to be true
```

The reason that the `int` examples above work in the way they do is because `int` variables are treated as arrays of `boolean` true/false values (`0` or `1`) which then obey boolean logic in the following way:

- **`&` (Bitwise AND)**: copies a bit to the result if it exists in both operands. So `0011 1100 & 0000 1101 = 0000 1100` = 12.
- **`|` (Bitwise OR)**: copies a bit if it exists in either operand. So `0011 1100 | 0000 1101 = 0011 1101` = 61.
- **`^` (Bitwise XOR)**: copies the bit if it is set in one operand but not both. So `0011 1100 ^ 0000 1101 = 0011 0001` = 49.
- **`<<` (Left shift)**: shifts the left operands value to the left by the number of bits specified by the right operand. So `0011 1100 << 2 = 1111 0000` = 240.

When dealing with `boolean` variables it is more common to use the "conditional" operators, which don't evaluate the second input if the answer is already known from the first:

```
// conditional AND (&&)
// conditional OR (||)

// Doesn't check a against c
// if a is not greater than b:
boolean biggest = (a > b) && (a > c);

// Doesn't check a against c
// if a is less than b:
boolean notBiggest = (a < b) || (a < c);
```

Important note: we can use the same variable either side of assignment operators. This is extremely common and you must get used to this concept:

```
int ib = 1;

// At this point in the code
// ib has the value 1
ib = ib + 8;
// ib now has the value 9

float d = 8.0f;
float e = 3.0f;
d = d*e;
d = d/e;
```

You can combine certain arithmetic and assignment operators. The resulting code is less verbose although can be less easy to understand:

```
int ia = 2; int ib = 1;
ia += ib; // same as: ia = ia + ib;
ia -= ib; // same as: ia = ia - ib;

float d = 2.0f; float e = 1.5f;

d *= e; // same as: d = d*e;
```

Summary of Data-Types and Operators:

```
// Built-in types are:

// byte, short
// int, long
// float, double
// char
// boolean

// Built-in operators are:

// +, -, *, /, %
// &, |, ~, <<, >>, >>>
// ==, !=, <, >, <=, >=
```

4.2 Functions

Now we've covered the basic types of **data**. The other essential element of all programming is the use of **functions**. We use functions to manipulate data and to replace repetitive pieces of code. We encapsulate the code to perform a certain function in one place, greatly aiding code maintainability and readability.

Here is an example of a function to calculate the y-value for the quadratic function $y = 2x^2+3x+5$:

```
// The function declaration is of the form:

<return type> FunctionName (<arguments>)

// where:
// <return type> : the type of the result
//                 returned by the function
//                 (e.g. int, double or a
//                  class type).
//
// FunctionName  : programmer defined function
//                 name.
//
// <arguments>   : list of variables with their
//                 types passed to the function,
//                 and used in the function code.

double quadratic (double x) {

  double y;
  double a = 2.0;
  double b = 3.0;
  double c = 5.0;

  y = a*x*x + b*x + c;

  // The following line is how we
  // send the value of the function
  // back to the code that called
  // the function:

  return y;

}
```

This function must be written outside of the “main” method and then called from it, as follows:

```
package module1;
public class Simple {
  public Simple() { }

  public double quadratic(double x) {
    double y;
    double a = 2.0;
    double b = 3.0;
    double c = 5.0;
    y = a*x*x + b*x + c;
    return y;
  }

  public static void main (String[] args) {
    // Instantiate Simple object:
    // explained in module 2
    Simple myS = new Simple();

    double xx = 2.0;
    double yy = myS.quadratic(xx);

    System.out.println("xx = "+xx);
    System.out.println("yy = "+yy);
  }
}
```

Functions can also be called from inside functions:

```
double function1( double x1 ) {

  double x3 = function2( x1/10.0 + 2 );
  double x2 = x1 * x3;

  return x2;
}
```

but `function2` must be defined somewhere, otherwise you will get a compilation error (try it).

Mathematical Functions

Of course it's useful to be able to use more sophisticated mathematical functions than just `+`, `-`, `*` and `/`. Java comes with a library of around 40 common mathematical functions. These functions are static methods of the class `java.lang.Math` and can be used as follows:

```
double cos_value = Math.cos(1.0);
```

When using Eclipse, if you type "`Math.`" the editor will then present to you a list of all the possible mathematical functions. You can also find documentation under "Resources" on the course web-page. Here is how "`Simple`" will look when you have your own user function and you also use a built-in Java math function:

```
package module1;
import java.lang.Math.*;

public class Simple {

 public Simple() { }

 public double sin2Theta (double theta){
  double y;

  // Convert input from degrees
  double arad = Math.toRadians(theta);

  // Example use of a
  // Java math function:
  y = 2.0*Math.sin(arad)*Math.cos(arad);

  return y;
 }

 public static void main (String[] args){
   Simple ss = new Simple();
   double angle = 45.0;

   // Call our function:
   double sin2t = ss.sin2Theta(angle);
 }
}
```

4.3 Algorithm Control

In this section we will cover the basic constructs necessary to add control to a program. All high level programming languages (both OO and non-OO) contain these constructs in one form or another. They are needed in almost every program and you cannot write useful code without mastering them.

4.3.1 `if` Statements

These statements are used to take a different action or direct the program flow elsewhere, depending on the logical result of some test. The keywords are `if`, `else if` and `else` as illustrated in this example:

```
// Example use of "if; else if; else"
if (a > b) {
   // Code written here is executed if:
   // (1) a is greater than b
}
else if ( a > c ) {
   // Code written here is executed if:
   // (1) the block above was NOT executed,
   //       and
   // (2) a is greater than c
}
else if ( a > d ) {
   // Code written here is executed if:
   // (1) Neither of the blocks above
   //     was executed,
   //       and
   // (2) a is greater than d
}
else {
   // Code written here is executed if:
   // (1) None of the blocks above
   //     was executed.
}
```

It's important to fully understand the logic above. Note that the successive `if` and `else if` tests are evaluated in order, and only the code in the block corresponding to the first successful test will be executed. The order is therefore important, and it is impossible for more than one block to be executed at a time. You can have any number of `else if` clauses. The final `else` is the default or "catch-all" case which is executed when none of the previous tests are successful. However it is entirely optional; if it is missing, and if all the previous tests fail, then none of the code in the `if` construct will be executed.

Any logical expression can appear in `if` tests. For example:

```
// greater than or less than
if (a > b) { }
if (a < b) { }

// greater/less than or equal to
if (a >= b) { }
if (a <= b) { }

// equality
if (a == b) {}
if (a != b) {}

// you can AND and OR tests
if ( (a > b) && (b > c) ) {}
if ( (a > b) || (b > d) ) {}

// etc.
```

When a test is comprised of a complicated logical expression we have to take into account the relevant **precedence** rules:

- brackets `()` take the highest precedence. Sometimes it's useful to use brackets even if

they are logically redundant in order to make the code easier to read by humans,

- the “AND” operator (`&&`) is next,
- the “OR” operator (`||`) takes the lowest precedence.

So for example, the following tests yield different results:

```
if (x == 1 || y == 2 && z == 3)
// is different from:
if ((x == 1 || y == 2) && z == 3)
```

You can see that the tests in `if` statements evaluate to a `boolean`; that is, the result of the test is simply “true” or “false”. Alternatively, we can use `boolean` variables directly in the tests as follows:

```
boolean isValid;

// Some code which sets "isValid"
// to be either true or false...

if (isValid) {
  // This code executes if
  // "isValid" is true
}

if (!isValid) {
  // This code executes if
  // "isValid" is false
}
```

4.3.2 `switch/case` Statements

This is a glorified `if`-statement. It is of limited use since it is restricted to testing the value of an integer. It can be used in certain circumstances to make the code more readable however:

```
int month = 2;

// Note the syntax of the "case"
// statement including the colon.
//
// Note the required "break"
// after the code for each case,
// which jumps to the code
// immediately following the final
// "}" at the end of the switch
// construction.

switch (month) {
  case 1:
    System.out.println("Jan");
    break;
  case 2:
    System.out.println("Feb");
    break;
  case 3:
    System.out.println("Mar");
    break;
  default:
    System.out.println("???");
    break;
}
```

4.3.3 `for` Loops

These are used to repeat a piece of code:

```
// The general format is:

for (<initial action>; <condition>; <loop action>) { }

// where:
//   <initial action> :
//     action performed before
//     looping starts. For
//     example, a loop counter
//     is initialised.
//
//   <logical condition> :
//     the test performed at
//     the start of each loop.
//     Looping continues until
//     the test fails, at which
//     point execution jumps to
//     the final "}".
//
//   <loop action> :
//     action performed at the
//     end of each loop. For
//     example, a loop counter
//     is incremented.

int i = 0;
int maxN = 10;

for ( i = 0 ; i < maxN ; i++) {

 System.out.println(" i = "+i);
 // execute code ...

}
```

4.3.4 `while` Loops

`while` is simpler than `if` but achieves the same thing — looping continues until a condition is no longer satisfied:

```
boolean funValue;
float x = 10;
funValue = true;

while (funValue) {

 x++;
 funValue = someFunction(x);

}
```

A variation on this is the *do-while* loop, which always executes the contained code at least once before checking the condition:

```
do {
  x++;
  funValue = someFunction(x);
} while (funValue);
```

---

[2] To understand why the answer is -24, you will need to understand the binary representation of negative numbers. For example see http://en.wikipedia.org /wiki/Two's_complement. ↩

---

5. Writing Readable Code

All the previous sections have dealt with the syntactic structure of your Java program. However, just as important is to ensure that your code is properly indented and commented. While you are writing code simply for yourself, the necessity of adding comments or correctly indenting your code may seem pointless: after all, most programmers understand the code they've just written. The key, however, is what happens when you pass on your code to someone else (such as the course demonstrators for marking...), or when you go back to code that you have not used for some time. Code that is badly laid out and lacking in comments is extremely difficult to read. Well-indented code makes it clear which sections of code are grouped together and therefore have some kind of connected functionality. Good comments allow the programmer to work out what each section of code is doing without having to go through the code line by line.

5.1 Code Indentation

You will have seen from all the previous code examples in this module that some lines of code start with a different spacing from the left-hand margin than others. This practice is called *Indentation* and provides a quick and easy way to see which sections of code are grouped together. In some programming languages, such as Python, code indentation actually has a programmatic meaning: the contents of a loop, for example, are indicated by an indent. In Java however, indenting code has no effect on the functionality: it is merely an extremely useful tool for making the code more readable.

As an example, which of the following code snippets seem more readable to you? Which makes it easier to determine the functionality and programmatic flow of each section?

```
package module1;
import java.lang.Math.*;

public class Simple {

  public Simple() { }

  public double sin2Theta(double theta) {
    double doubAng;
    double arad = Math.toRadians(theta);

    doubAng = 2.0*Math.sin(arad)*Math.cos(arad);

    return doubAng;
  }

  public static void main(String[] args) {
    Simple ss = new Simple();
    double angle = 45.0;

    double sin2t = ss.sin2Theta(angle);
  }

}
```

```
package module1;
import java.lang.Math.*;

public class Simple {
public Simple() { }
public double sin2Theta (double theta){
double doubAng;
double arad = Math.toRadians(theta);
y = 2.0*Math.sin(arad)*Math.cos(arad);
return doubAng;
}
public static void main (String[] args){
Simple ss = new Simple();
double angle = 45.0;
double sin2t = ss.sin2Theta(angle);
}
}
```

The answer should be obvious! The first example makes it much easier to see where a method begins and ends and the code it contains. The only way of working this out from the second example is to go through it line by line and count the opening and closing curly braces "`{}`".

In most IDE's — including Eclipse — pressing the "tab" key will indent the code by 4 spaces: this is the convention for the amount of space an indent occupies. Therefore, a single indent is 4 spaces, a double indent 8 spaces and so on. Fortunately, the Eclipse IDE provides an automatic tool for applying the correct code indentation[3] to a whole code block: highlight the code you want to automatically indent and either select "*Correct Indentation*" from the *Source* menu or type "Ctrl-I" (or "Cmd-I" on Macs).

As such, getting the layout of your code correct is relatively straightforward, assuming that

you also break your lines in reasonably sensible places. Guidelines for correctly inserting line breaks can be found in the Google Java Style Guide: https://google.github.io /styleguide/javaguide.html (see the section on Formatting). Be aware that, when submitting code for marking, **you will lose marks if your code is not correctly indented and cleanly laid out**.

5.2 Comments

As with correct code indentation, good comments make no difference to the function of your code. However, they are just as important for making the code easy to understand. The golden rule for adding comments is: "*provide as much information as possible in as few words as possible*". This may seem paradoxical, but it is important that your comments are descriptive without being overly long. If you end up with comments that are longer that your code, you've almost certainly written too much!

There are two ways to add comments to your code:

- Using two forward slashes "`//`": Java ignores everything after these slashes on the same line. It is common to use this practice as a quick way to comment out code you want to keep but don't want to be executed just now.
- Enclosing your comments within a block that begins with "`/*`" (a forward slash and asterisk) and ends with "`*/`" (asterisk and a forward slash). Any text between these delimiters (even if it spans multiple lines) is ignored.

When it comes to writing good comments, it is important that your comments describe something about what the code is doing that makes the code easier to understand. The purpose of comments is to explain **why**, not **what**: comments shouldn't simply repeat what is already obvious about the code. You don't need to say "`// Method to ...`" at the start of a method: we already know it's a method! Take the following examples:

```
public class Simple {

  public Simple() { }

  // Calculate sin of double angle
  public double sin2Theta(double theta) {

    double doubAng;

    // Convert angle theta from degrees
    double arad = Math.toRadians(theta);

    // sin(2θ) = 2 sin(θ) cos(θ)
    doubAng = 2.0*Math.sin(arad)*Math.cos(arad);

    return doubAng;

  }

}
```

```
// Definition of Simple class
public class Simple {

  // Simple class null constructor
  public Simple() { }

  // Method to calculate sin2Theta
  public double sin2Theta(double theta) {

    // Variable doubAng
    double doubAng;

    // Calculate arad from theta
    double arad = Math.toRadians(theta);

    // Final calc
    doubAng = 2.0*Math.sin(arad)*Math.cos(arad);

    // Return calc
    return doubAng;

  }

}
```

How much information are you able to glean about what the code is doing from the second example as opposed to the first? The comments in the second example aren't actually helpful in any way: they simply restate what the code is already doing without adding anything to the explanation! The first example actually contains *fewer* comments, yet provides *more* information over and above what is already evident from the code. Of course, if your comments are overly concise you won't actually provide any extra information: comments should **always add value** to the code to make it easier to understand (without, of course, writing an entire paragraph where one line will do...). As with code indenting, **you will lose marks if your code is not properly commented**.

Finally, remember that comments should **not** be a substitute for well-written code. By using well-chosen names for variables and methods — eschewing single-letter variables (`a`, `b`, `c` etc.) unless there are good reasons for it; avoiding undescriptive method names (`method1`, `myMethod`, `doStuff` etc.) — your code becomes easier to understand even before you start adding comments. As a rule, **clean code with fewer comments is always better than bad code with lots of comments**. But well-written comments will make your code that much easier to interpret: something you will appreciate the first time you have to go back to some old code or try working your way through someone else's...

---

[3] More information on code indenting can be found on the Wikipedia page on Code Indenting: en.wikipedia.org/wiki/Indent_style. Eclipse uses the 1TBS variant of the Kernighan & Ritchie style, also known as “Egyptian Brackets”. ↩

---

6. Summary

In this module we have learned some of the concepts behind object-oriented programming. We have learned how to assemble simple Java programs. We have learned about the basic elements of the Java language including **data** types, **functions** and **algorithm control**. These concepts will become clear through examples and the more subtle aspects of object-oriented programming will be revisited in subsequent modules.