Programming in Java (23/24) – Day 1: Here it all begins. . .

1 Introduction

1.1 Motivation

Computing increasingly permeates everyday life, from the apps on our phones to the social networks that connect us. At the same time, computing is a general tool that we can use for solving real-world problems. For you to get a computer to do something, you have to tell it what you want it to do. You give it a series of instructions in the form of a *program*. The computer then helps us by carrying out these instructions very fast, much faster and more reliably than a human. In this sense, a program is a description of how to carry out a process automatically.

Often programs that others have written will do the job. However, we don't want to be limited to the programs that other people have created. We want to tailor our program to the needs of the problem that we have at hand. Writing programs is about expressing our ideas for solving a problem clearly and unambiguously. This is a skill increasingly sought also outside of the software industry. The programs that you can write may then help solve problems in areas such as business, data science, engineering, and medicine.

1.2 Programming language

You write a program in a *programming language*. Many programming languages have been devised over the decades. The language we will use in this document is named *Java*.

Programming languages vary a lot, but an instruction in a typical programming language might contain some English words (such as `while` or `return`), perhaps a mathematical expression (such as `x + 2`) and some punctuation marks used in a special way.

Programs can vary in length from a few lines to thousands of lines. Here is a complete, short program written in Java:

```
 1  public class Example1 {
 2      public static void main(String[] args) {
 3          System.out.println("Given a series of words, each on a separate line,");
 4          System.out.println("this program finds the length of the longest word.");
 5          System.out.println("Please enter several sentences, one per line.");
 6          System.out.println("Finish with a blank line.");
 7          int maxi = 0;
 8          String word = ".";
 9          java.util.Scanner scan = new java.util.Scanner(System.in);
10          while (word.length() > 0) {
11              word = scan.nextLine();
12              if (word.length() > maxi) {
13                  maxi = word.length();
14              }
15          }
16          if (maxi == 0) {
17              System.out.println("There were no words.");
18          } else {
19              System.out.println("The longest sentence was " + maxi + " characters long.");
20          }
21      }
22  }
```

Listing 1: Example1.java

Here I have added line numbers on the left of the program text so that we can easily talk about specific lines. These line numbers are not part of the program text itself.

When you write a program, you have to be very careful to keep to the syntax of the programming language, i.e., to the grammar rules of the language. For example, the above Java program would behave incorrectly if in line 16 we wrote

```
        if (maxi = 0) {
```

instead of

```
        if (maxi == 0) {
```

or if in line 12 we put a semi-colon immediately after

```
            if (word.length() > maxi)
```

The computer would refuse to accept the program if we added dots or semi-colons in strange places or if any of the parentheses (or dots) were missing, or if we wrote `len` instead of `length()` or even `Length()` instead of `length()`.

When you look at the program, you may notice that it looks perhaps a bit more complicated than one would expect from other programming languages, such as Python. Before the first instruction

```
        System.out.println("Given a series of words, each on a separate line,");
```

that actually tells the computer to do something (here, print a message on the screen), there are some extra words.

On this and the following days, your Java programs will always look roughly like this:

```
public class Birkbeck {
    public static void main(String[] args) {
        ...
    }
}
```

We need the first line to declare a *class*, here with the name Birkbeck.[1] Now you may ask, what on Earth is a "class"? For our purposes at the moment, this is not so important – for now, let us pretend that "class" means the same as "program".[2] The second line introduces a *method* called main. Roughly, a method in Java is an independent part of a program that can be run on its own. Inside a method, we can write the actual instructions that the computer is supposed to execute one by one. Running a Java program usually starts with the first instruction of a method that has the above shape and name (i.e., it is called main, and also the other bits are as above).

In our example, the name of the file in which you store your program must be Birkbeck.java. This means that the filename must be the same as the name of the class.

Over the next weeks, you will learn what all those words in the first two lines of our example mean. Today, let us use these two lines "as is" (changing the name of the program as needed) and let us focus on some of the instructions that we can use in the place of the "...".

1.3 Input and output

Almost all programs need to communicate with the outside world to be useful. In particular, they need to read input and to remember it. They also need to respond with an answer.

To get the computer to take some input as text and to store it in its memory, in Java we first write:

```
        java.util.Scanner scan = new java.util.Scanner(System.in);
```

This line creates a new "Scanner" *object*[3] that reads from the keyboard[4] and puts a "handle" to it into a box in the computer's memory with a name that I have chosen to be scan. In Java, the word *Scanner* has no special meaning as such, so here I have written java.util.Scanner to tell Java that it can find the definition of Scanner that I mean in the "package" java.util.[5]

[1] I have chosen this name to make clear that the name of the class can be (almost) anything.

[2] This is not the full story – classes will become important later also to let us create our own data types.

[3] We will learn more about objects soon.

[4] In Java, the "input stream" for the key presses by the user on the keyboard is called System.in, pronounced *system dot in*.

[5] Java programs usually consist of many files of Java source code, i.e., program text. Packages are a way of organising these files, similar to folders or directories in the file system of a computer. We will learn more about working with packages soon.

After we have created our Scanner object, we can ask the Scanner object for the next line from the keyboard, by writing:

```
            word = scan.nextLine();
```

The phrase `scan.nextLine()` (pronounced *scan dot next line*) has a special meaning to the computer. The combination of these instructions means: “Take some input which is a sequence of characters and put it into the computer’s memory.”[6]

`word` by contrast, is a word that I (the programmer) have chosen. I could have used `characters` or `thingy` or `breakfast` or just `s` or almost any word I wanted. (There are some restrictions which I will deal with in the next section.) For most instructions, Java expects a semi-colon at the end so that it knows that the instruction is complete.

Computers can take in all sorts of things as input — numbers, letters, words, records and so on — but, to begin with, we will write programs that generally handle text as sequences (or *strings*) of characters (like `"I"`, `"hi"`, `"My mother has 3 cats"`, or `"This is AWESOME!"`). We’ll also assume that the computer is taking its input from the keyboard, i.e., when the program is executed, you key in one or more words at the keyboard and these are the characters that the computer puts into its memory.

You can imagine the memory of the computer as consisting of lots of little boxes. Programmers can reserve some of these boxes for use by their programs and they refer to these boxes by giving them names.

```
            word = scan.nextLine();
```

means “Take a string of characters input using the keyboard and terminated when the user presses RETURN, and then put this string into the box called word.”[7] When the program runs and this instruction gets executed, the computer will take the words which you type at the keyboard (whatever you like) and will put them into the box which has the name `word`.

Each of these boxes in the computer’s memory can hold only one string at a time. If a box is holding, say, `"hello"`, and you put `"Good bye!"` into it, the `"Good bye!"` replaces the `"hello"`. In computer parlance these boxes are called *variables* because the content inside the box can vary over time; you can put an `"I"` in it to start with and later change it to a `"you"` and change it again to `"That was a hurricane"` and so on as often as you want. Your program can have as many variables as you want.

In Java, you have to tell the computer that you want to use a variable with a certain name before you use it. You can’t just pitch in using `word` without telling the computer what `word` is. Most of the time you also have to tell the computer what *type* of variable this is, i.e., what sort of thing you are going to put into it.[8] In this case we are going to put strings of characters into `word`. Strings of characters,

[6]More precisely, it means “Ask the object with the name `scan` to read a line of character input” (from the keyboard, not the screen because the user cannot type a line onto the screen... or could not before they invented touchscreens!).

[7]In general, a `Scanner` may also read from other sources, such as text files on your hard disk. However, here we have created a `Scanner` object that reads from `System.in`, i.e., the keyboard.

[8]Many other programming languages like Python or JavaScript do not expect the programmer to provide this information. However, this extra type information is very helpful to make it easier to write robust large programs with few(er) errors.

or simply *strings* as they are called in programming, are known in Java as String, with capital S. To tell the computer that we want to use a variable of type String called word, we write in our program:

```
        String word;
```

If we wanted to declare more than one variable, we could use two lines:

```
        String myName;
        String yourName;
```

or we could declare them both on one line, separated by a comma:

```
        String myName, yourName;
```

In Java (and many other programming languages), you have to make sure that you have put something into a variable before you read from it, e.g., to print its contents. If you try to take something out of a box into which you have not yet put anything, it is not clear what that "something" should be. This is why Java will complain if you write a program that reads from a variable into which nothing has yet been put.

If you want the computer to display (on the screen) the contents of one of its boxes, you use System.out.println(thingy);[9] where instead of thingy you write the name of the box. For example, we can print the contents of word by writing:

```
      System.out.println(word);
```

If the contents of the box called word happened to be "Karl", then when the computer came to execute the instruction System.out.println(word); the word "Karl" would appear on the screen.

So we can now write a program in Java (not a very exciting program, but it's a start):

```java
public class Example2 {
    public static void main(String[] args) {
        java.util.Scanner scan = new java.util.Scanner(System.in);
        String word;
        word = scan.nextLine();
        System.out.println(word);
    }
}
```

Listing 2: Example2.java

This program takes some text as input from the keyboard and displays it back on the screen.

In Java, it is customary to lay out programs like this or in our first example with each instruction on a line of its own. The indentation, if there is any, is just for the benefit of human readers, not for the computer which ignores any indentations.

[9]More precisely, this means "Take the current system – this computer, take its standard output stream which goes to the screen, then print a line to this output stream"

This is different from, e.g., Python, where indentation is used as a part of the language to structure the programs.

Instead of indentation, Java uses curly braces "{" and "}" to structure the program. The curly braces come in pairs – for each `{` used to open a structure element, there is eventually a companion `}` to close the element.

So we could have written our program `Example2.java` with the same words and symbols in a single very long line. But that would not be very readable for other programmers. A convention among Java programmers is that whenever we write a `{`, we indent the following lines by 4 more spaces, until we reach the matching `}`. The Java programs that we have seen so far use this convention for indentation. This makes Java programs much easier to read.

1.4 Running a program

You can learn the rudiments of Java from these notes just by doing the exercises with pencil and paper. However, it is a good idea to test your programs using a computer after you have written them so you can validate that they do what you intended. If you have a computer and are wondering how you run your programs, you will need to know the following, and, even if you don't have a computer, it will help if you have some idea of how it's done.

First of all, you type your program into the computer using a text editor. Then, before you can run the program, you have to *compile* it. This means that you pass it through a piece of software called a *compiler*. The compiler checks whether your program is acceptable according to the syntax of Java. If it isn't, the compiler issues one or more error messages telling you what it objects to in your program and where the problem lies. You try to see what the problem is, correct it, and try again. You keep doing this until the program compiles successfully, without further objections by the compiler. You now have an *executable* version of your program, i.e., your program has been translated into the internal machine instructions of the computer and the computer can run your program.

Then, to run your program, you use a piece of software called an *interpreter*. To make the computer run your program using the interpreter, you need to issue a command. You can issue commands[10] from the command prompt in Windows (you can find the command prompt under *Start* → *Accessories*), or from the terminal in Linux and Mac OS/X.

If you are lucky, your program does what you expect it to do first time. Often it doesn't. You look at what your program is doing, look again at your program and try to see why it is not doing what you intended. You correct the program and run it again. You might have to do this many times before the program behaves in the way you wanted. This is normal and nothing to worry about at this stage.

As I said earlier, you can study this introduction without running your programs on a computer. However, it's possible that you have a PC with a Java compiler and will try to run some of the programs given in these notes. If you have a PC but you don't have a Java compiler, I attach a few notes telling you how you can obtain one (see Section A).

[10] In a modern operating system, you can click on an icon to execute a program. However, this only makes sense for graphical applications and not for the simple programs that you will write at first.

1.5 Outputting words and ends of lines

Let's suppose that you manage to compile your program and that you then run it. Your running of the above program would produce something like this on the screen if you typed in the word Tom followed by RETURN:

```
Tom
Tom
```

The first line is the result of you keying in the word Tom. The system "echoes" the keystrokes to the screen, in the usual way. When you hit RETURN, the computer executes the

```
        word = scan.nextLine();
```

instruction, i.e., it reads the word or words that have been input. Then it executes the

```
        System.out.println(word);
```

instruction and the word or words that were input appear on the screen again.

We can also get the computer to display additional words by putting them in quotes in the parentheses after the `System.out.println`, for example:

```
        System.out.println("Hello");
```

We can use this to make the above program more user-friendly:[11]

```java
public class Example3 {
    public static void main(String[] args) {
        java.util.Scanner scan = new java.util.Scanner(System.in);
        String word;
        System.out.println("Please key in a word:");
        word = scan.nextLine();
        System.out.println("The word was:");
        System.out.println(word);
    }
}
```

Listing 3: Example3.java

An execution, or "run", of this program might appear on the screen thus:

```
Please key in a word:
Tom
The word was:
Tom
```

[11]In general, it is polite and sensible to tell the user whenever your program needs the user to do something.

Here each time the computer executed the System.out.println instruction, it also went into a new line. If we use System.out.print instead of System.out.println then no extra line is printed after the value.[12] So we can improve the formatting of the output of our program on the screen:

```
 1  public class Example3a {
 2      public static void main(String[] args) {
 3          java.util.Scanner scan = new java.util.Scanner(System.in);
 4          String word;
 5          System.out.print("Please key in a word: ");
 6          word = scan.nextLine();
 7          System.out.print("The word was: ");
 8          System.out.println(word);
 9      }
10  }
```

Listing 4: Example3a.java

Now a "run" of this program might appear on the screen thus:

```
Please key in a word: Tom
The word was: Tom
```

Note the spaces in lines 5 and 7 of the program after `word:` and `was:`. This is so that what appears on the screen is `word: Tom` and `was: Tom` rather than `word:Tom` and `was:Tom`.

It's possible to output more than one item with a single System.out.println instruction. For example, we could combine the last two lines of the above program into one:

```
        System.out.println("The word was: " + word);
```

and the output would be exactly the same. The symbol "+" does not represent addition in this context, it represents concatenation, i.e., writing one string after the other. We will look at addition of numbers in the next section.

Let's suppose that we now added three lines to the end of our program, thus:

```
public class Example4 {
    public static void main(String[] args) {
        java.util.Scanner scan = new java.util.Scanner(System.in);
        String word;
        System.out.print("Please key in a word: ");
        word = scan.nextLine();
        System.out.println("The word was: " + word);
        System.out.print("Now please key in another: ");
        word = scan.nextLine();
        System.out.println("And this one was: " + word);
    }
}
```

[12] println is short for "print line".

Listing 5: Example4.java

After running this program, the screen would look something like this:

```
Please key in a word: Tom
The word was: Tom
Now please key in another: Jane
And this one was: Jane
```

Exercise A

Now pause and see if you can write:

1. a Java instruction which would output a blank line.
2. an instruction which would output

   ```
   Hickory, Dickory, Dock
   ```

3. a program which reads in two words, one after the other, and then displays them in reverse order. For example, if the input was

   ```
   First
   Second
   ```

   the output should be

   ```
   Second
   First
   ```

1.6 Assignment and initialisation

There is another way to get a string into a box apart from using `scan.nextLine();`. We can write, for instance:

```
    word = "Some text";
```

This has the effect of putting the string `"Some text"` into the word box. Whatever content was in word before is obliterated; the new text replaces the old one.

In programming, this is called *assignment*. We say that the value `"Some text"` is assigned to the variable word, or that the variable word takes the value `"Some text"`. The "=" symbol is the assignment operator in Java. We are not testing whether word has the value `"Some text"` or not, nor are we stating that word currently has the value `"Some text"`; we are *giving* the value `"Some text"` to word.

An assignment instruction such as this:

```
    word = word + " and some more";
```

looks a little strange at first but makes perfectly good sense. Let's suppose the current value of word (the contents of the box) is "Some text". The instruction says, "Take the value of word ("Some text"), add " and some more" to it (obtaining "Some text and some more") and put the result into word". So the effect is to put "Some text and some more" into word in place of the earlier "Some text".

The = operator is also used to "initialise" variables.

When the computer allocates a portion of memory to store one of your variables, it does not clear it for you; the variable holds whatever value this portion of memory happened to have the last time it was used. Its value is said to be *undefined.*

To prevent yourself from using undefined values, you can give a variable an initial value when you declare it. For example, if you wanted `str` to begin with the empty string, you should declare it thus:

```
      String word = "";
```

This is very much like assignment since we are giving a value to word. But this is a special case where word did not have any defined value before, so it is known as *initialisation.*

Finally a word about terminology. I have used the word "instruction" to refer to lines such as `System.out.println(word);` and `word = "Some text";`. It seems a natural word to use since we are giving the computer instructions. But the correct word is actually "statement". `System.out.println(word);` is an output statement, and `word = "Hello";` is an assignment statement. The lines in which we tell the computer about the variables we intend to use, such as `String word;` or `String word = "";` are called variable *definitions*. They are also referred to as variable *declarations*. When you learn more about Java, you will find that you can have declarations which are not definitions, but the ones in these introductory notes are both definitions and declarations.

Exercise C

Now see if you can write a program in Java that takes two words from the keyboard and outputs one after the other on the same line. E.g., if you keyed in "Humpty" and "Dumpty" it would reply with "Humpty Dumpty" (note the space in between). A run of the program should look like this:

```
    Please key in a word: Humpty
    And now key in another: Dumpty
    You have typed: Humpty Dumpty
```

2 Variables, identifiers and expressions

In this section, we will look at the fundamentals of how Java stores and manipulates information. To this end, we will discuss integer numbers, arithmetic expressions, identifiers, comments, and strings.

2.1 Integer variables and numbers

As well as strings, we can also have integer numbers (often just *integers*) as values: whole numbers. So variables can contain not only strings, but also any[13] integer value like 0, -1, -200, or 1966.

We would declare an **int** variable called `i` as follows:

```
        int i;
```

We can assign an integer value, say, `0`, to a variable, say, `i`:

```
        i = 0;
```

or, if we had two variables `i` and `j`, we could assign:

```
        i = j;
```

We can even say that `i` takes the result of adding two numbers together:

```
        i = 2 + 2;
```

which results in `i` having the value 4.

Integers and strings

You have probably noticed that, when dealing with integer values the symbol "+" represents addition of numbers, while – as we saw in the last section – when dealing with strings the same symbol represents concatenation. Therefore, the statement

```
        String word = "My name is " + "Inigo Montoya";
```

results in word having the value `"My name is Inigo Montoya"`, while the statement

```
        int n = 10 + 7;
```

results in n having the value 17 (not 107). What happens if we mix integer values and string values when using "+"?

[13]Actually, a computer's memory is finite and therefore there are limits to the possible values for an integer variable, but they are big enough for most programs.

In that case, Java converts the integers to strings and perform concatenation. For example, the program snippet (also called program fragment)[14]

```
        String word = "My name is " + "Inigo Montoya";
        int n = 10 + 7;
        String text = word + " and I am " + n;
        System.out.println(text);
```

will print on the screen "My name is Inigo Montoya and I am 17". It is important to know that the same symbol can be used for different things, but we will come to this later again; now let's go back to writing programs with a bit of maths in them using integers.

2.2 Reading integers from the keyboard

In the last section, we saw how we can read a string of characters from the keyboard, creating a Scanner object that we called scan and then issuing the command scan.nextLine(). We can use the same command to read a number... but the computer will not know it is a number, it will think it is a string of characters. If we want to tell the computer that a sequence of characters *is* a number, we need to convert it. This is similar to the way we needed to tell the computer that we wanted to treat a number as a character string in the previous example. We can do this easily by *parsing* it using the command Integer.parseInt():

```
public class S2Example1 {
    public static void main(String[] args) {
        java.util.Scanner scan = new java.util.Scanner(System.in);
        System.out.print("Please introduce a number: ");
        String word = scan.nextLine();
        int n = Integer.parseInt(word);
        System.out.println("The number was " + n);
        System.out.println("The next number is " + (n + 1));
    }
}
```

Listing 6: S2Example1.java

When we parse a string that contains an integer we obtain an integer with the correct value. If we try to parse a string that is not an integer (for example, the word "Tom") the program will terminate with

[14] The instructions in this snippet need to go between

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

and

```
    }
}
```

as in the earlier examples. In the following, the examples will not always be complete programs – I assume that you now know how to put them into a class with a suitable main method so that you can compile and run them, similar to the example programs that we saw earlier.

an error message on the screen. If we do not parse the string and use it as if it was an integer, the results will not be what you'd expect. This is a common source of errors in programs. You can check for yourself what happens if you do not parse the string in the former example, e.g., what happens with this program:

```java
public class S2Example1_noParsing {
    public static void main(String[] args) {
        java.util.Scanner scan = new java.util.Scanner(System.in);
        System.out.print("Please introduce a number: ");
        String word = scan.nextLine();
        System.out.println("The number was " + word);
        System.out.println("The next number is " + (word + 1));
    }
}
```

Listing 7: S2Example1_noParsing.java

We can use the command `Integer.parseInt()` to parse strings that may come directly from the keyboard, but also strings that our program has computed in several steps. If all we want is to read an integer number directly from the keyboard, there is a shortcut with the help of a Scanner:

Instead of

```java
        java.util.Scanner scan = new java.util.Scanner(System.in);
        String word = scan.nextLine();
        int n = Integer.parseInt(word);
```

we can write

```java
        java.util.Scanner scan = new java.util.Scanner(System.in);
        int n = scan.nextInt();
```

and skip the intermediate string. Here the Scanner does the parsing of the input to an integer number for us.

Now, assuming that you read your integers and always remember to parse them, what maths can you do with them?

2.3 Operator precedence

Java uses the following arithmetic operators (amongst others):

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	modulo

The last one is perhaps unfamiliar. The result of x % y ("x mod y") is the remainder that you get after dividing the integer x by the integer y. For example, 13 % 5 is 3; 18 % 4 is 2; 21 % 7 is 0, and 4 % 6 is 4 (6 into 4 won't go, remainder 4). num = 20 % 7 would assign the value 6 to num.

How does the computer evaluate an expression containing more than one operator? For example, given 2 + 3 * 4, does it do the addition first, thus getting 5 * 4, which comes to 20, or does it do the multiplication first, thus getting 2 + 12, which comes to 14? Java, in common with other programming languages and with mathematical convention in general, gives precedence to *, / and % over + and -. This means that, in the example, it does the multiplication first and gets 14.

If the arithmetic operators are at the same level of precedence, it takes them left to right. 10 - 5 - 2 comes to 3, not 7. You can always override the order of precedence by putting parentheses into the expression; (2 + 3) * 4 comes to 20, and 10 - (5 - 2) comes to 7.

Some words of warning are needed about division. First, remember that integer values are whole numbers. So the result of a division with / is only the integer part without the decimal part. Try to understand what the following program does:

```java
public class S2Example2 {
    public static void main(String[] args) {
        int num = 5;
        System.out.println(num);
        num = num + 2;
        System.out.println(num);
        num = num / 3 * 6;
        System.out.println(num);
        System.out.println(7 + 15 % 4);
        num = 24 / 3 / 4;
        System.out.println(num);
        num = 24 / (num / 4);
        System.out.println(num);
    }
}
```

Listing 8: S2Example2.java

A computer would get into difficulty if it tried to divide by zero (it is not clear what the result should be). Consequently, the system makes sure that it never does. If a program tries to get the computer to divide by zero, the program is unceremoniously terminated, usually with an error message on the screen.

Exercise A

Write down the output of the above program without executing it.

Now execute it and check if you got the right values. Did you get them all right? If you got some wrong, why was it?

2.4 Identifiers and comments

I said earlier that you could use more or less any names for your variables. I now need to qualify that.

The names that the programmer invents are called *identifiers*. The rules for forming identifiers are that the first character can be a letter (upper or lower case, usually the latter) and subsequent characters can be letters or digits or underscores. (Actually the first character can be an underscore, but identifiers beginning with an underscore are often used by system programs and are best avoided.) Other characters are not allowed. Java is case-sensitive, so `Num`, for example, is a different identifier from `num`.

The only other restriction is that you cannot use any of the language's keywords as an identifier. You couldn't use **`if`** as the name of a variable, for example.[15] There are many keywords but most of them are words that you are unlikely to choose. Ones that you might accidentally hit upon are **`assert`**, **`break`**, **`catch`**, **`class`**, **`continue`**, **`double`**, **`finally`**, **`float`**, **`import`**, **`private`**, **`public`**, **`return`**, **`static`**, **`switch`**, **`throw`** and **`try`**. You should also avoid using words which, though not technically keywords, have special significance in the language, such as `String` and `println`.

Programmers often use very short names for variables, such as `i`, `n`, or `x` for integers. There is no harm in this if the variable is used to do an obvious job, such as counting the number of times the program goes round a loop, and its purpose is immediately clear from the context. If, however, its function is not so obvious, **it should be given a name that gives a clue as to the role it plays in the program**. If a variable is holding the total of a series of integers and another is holding the largest of a series of integers, for example, then call them `total` and `maxi` rather than `x` and `y`.

Sometimes we want to choose a variable name that consists of several words, such as "exam mark" to store the result of an exam. But Java does not allow us to have a space in the name of a variable. A convention among Java programmers is to remove the space and instead use a capital letter for the letter right after the space. In our example, we would call our variable `examMark`. This is also called "Camel Case", because of the "humps" inside the words. Another convention among Java programmers is that names of variables should start with a lower-case letter.

The aim in programming is to write programs that are "self-documenting", meaning that a (human) reader can understand them without having to read any supplementary documentation. A good choice of identifiers helps to make a program self-documenting.

Comments provide another way to help the human reader to understand a program. Anything on a line after "`//`" is ignored by the interpreter, so you can use this to annotate your program. You might summarise what a chunk of program is doing:

```
        // sorts numbers into ascending order
```

or explain the purpose of an obscure bit:

```
        x = x * 100 / y;   // x as percent of y
```

[15]Note that in our examples and in many text editors for Java source code, such keywords are highlighted in a special colour. You don't need to apply the colouring manually – the editor does it for you to make the Java code more readable.

Comments inside your code should be few and helpful. Do not clutter your programs with statements of the obvious such as:

```
        num = num + 1;    // add 1 to num
```

Judicious use of comments can add greatly to a program's readability, but they are second-best to self-documentation. Note that comments are ignored by the computer: their weakness is that it is all too easy for a programmer to modify the program but forget to make any corresponding modification to the comments, so the comments no longer quite go with the program. At worst, the comments can become so out-of-date as to be positively misleading, as illustrated in this case:

```
        num = num + 1;  // decrease num
```

What is wrong here? Is the comment out of date? Is there a bug in the code and the plus should be a minus? Remember, use comments sparingly and make them matter. A good rule of thumb is that comments should explain "why" and not "how": the code already says *how* things are done!

Exercise B

Say for each of the following whether it is a valid identifier in Java and, if not, why not:

```
BBC, Python, y2k, Y2K, old, new, 3GL, a.out,
first-choice, 2nd_choice, third_choice
```

2.5 A bit more on strings

We have done already many things with strings: we have created them, printed them on the screen, even concatenated several of them together to get a longer value. But there are many other useful things we can do with strings.

For example, if you want to know how long a string is, you can find out using the *method*[16] `length()`. If `word` is the string, then `word.length()` gives its length. Do not forget the dot or the parentheses: methods are always prefixed with a dot and followed by parentheses[17] (sometimes empty, sometimes not). You could say, for example:

```
        java.util.Scanner scan = new java.util.Scanner(System.in);
        System.out.print("Please enter some text: ");
        String word = scan.nextLine();
        int length = word.length();
        System.out.println("The string " + word + " has " + length + " characters.");
```

You can obtain a substring of a string with the `substring` method. If the variable `s` has the string value `"Silverdale"`, then `s.substring(0,6)` will give you the first six letters, i.e., the string `"Silver"`. The first number in parentheses after the string says where you want the substring to begin, and the

[16]A method is a named sequence of instructions. We *call* the method using its name in order to execute its instructions. We can call a method without knowing what instructions it consists of, so long as we know what it does.

[17]Now you can see that `println()` and `nextLine()` are also methods.

second number says where you want it to end (to be precise, the second number is the position of the first character that we *don't* want in the substring).

Note that the initial character of the string is treated as being at position 0, not position 1. Similarly, `s.substring(2,6)` will give you the string `lver`.

If you leave out the second number, you get the rest of the string. For example, `s.substring(6)` would give you `"dale"`, i.e., the tail end of the string beginning at character 6 (`'d'` is character 6, not 7, because the `'S'` character is character 0).

You can output substrings with `System.out.println` or assign them to other strings or combine them with the "+" operator. For example:

```java
public class S2Example3 {
    public static void main(String[] args) {
        String s = "software services";
        s = s.substring(0,4) + s.substring(8,9) + s.substring(13);
        System.out.println(s);
    }
}
```

Listing 9: S2Example3.java

will output "soft ices".

Exercise C

Say what the output of the following program would be:

```java
public class S2Example4 {
    public static void main(String[] args) {
        String s = "artificial reality";
        System.out.println(s.substring(11,15) + " " + s.substring(0,3));
        int length = s.substring(11).length();
        System.out.println(length);
    }
}
```

Listing 10: S2Example4.java

Then run the program and see if you were right.

3 Conditionals (if statements)

To write anything more than very straightforward programs, we need some way of getting the computer to make choices. We do this in Java with the keyword **if**. We can write, for example:[18]

```
        if (num == 180) {
            System.out.println("One hundred and eighty!");
        }
```

When the computer executes this, it first checks whether the variable num currently has the value 180 or not. If it does, the computer displays its message; if it doesn't, the computer ignores the print line and goes on to the next line.

Note that the conditional expression (num == 180) has to be in parentheses, and that we use curly braces to indicate what to do if the conditional expression is true.

Note also that, to test whether num has the value 180 or not, we have to write **if** (num == 180) and not **if** (num = 180). We have to remember to hit the "=" key twice. This can be a serious nuisance in Java, especially for beginners. It comes about because the language uses the "=" operator for a different purpose, namely assignment. num = 180 does not mean "num is equal to 180", it means "Give the value 180 to num". You may feel that it is obvious that assignment is not what is intended in **if** (num = 180), but unfortunately that is how the computer will interpret it – and will complain. You have been warned.

The following program takes two numbers and displays a message if they happen to be the same:

```
public class S3Example1 {
    public static void main(String[] args) {
        java.util.Scanner scan = new java.util.Scanner(System.in);
        System.out.print("Please key in a number: ");
        int num1 = scan.nextInt();
        System.out.print("And another: ");
        int num2 = scan.nextInt();
        if (num1 == num2) {
            System.out.println("They are the same.");
        }
    }
}
```

Listing 11: S3Example1.java

[18]Here num would have been declared with the type int and given some value before these lines.

3.1 Conditional expressions

Conditional expressions (or conditions) – the kind that follow the keyword **if** – can be formed using the following operators:

==	is equal to
!=	is not equal to
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to

When these operators are used with integers, their meaning is obvious. With the help of an extra method called compareTo, they can also be used with strings. Here their meaning corresponds to something like alphabetical order (the order used for entries in a printed dictionary). For instance, `if (s.compareTo(t) < 0)`, where s and t are strings, means "If s comes before t in alphabetical order". So it would be true if s had the value `"Birkbeck"` and t had the value `"College"`. All the upper-case letters come before the lower-case, so `s.compareTo(t) < 0` would still be true if s had the value `"Zebra"` and t had the value `"antelope"` (upper-case "Z" comes before lower-case "a").

But what about strings that contain non-alphabetic characters? Would s come before t if s had the value `"#+*"` and t had the value `"$&!"`? To find the answer we have to consult the *UNICODE table* – the Universal Character Set. UNICODE defines a particular ordering of all the characters on the keyboard. (There are other orderings in use, notably EBCDIC which is used on IBM mainframe computers, and ASCII, which was adopted by PCs and became the de facto standard for English-based languages.) The UNICODE table tells us that the character "#" comes before the character "$", for instance. The latest version of Unicode[19] consists of a repertoire of more than 120,000 characters covering 129 different scripts (including Latin, Cyrillic, Arabic, all the Japanese ones, and many more). Some points worth remembering are:

- The space character comes before all the printable characters.
- Numerals come in the order you'd expect, from "0" to "9".
- Letters come in the order you'd expect, from "A" to "Z" and from "a" to "z".
- Numerals come before upper-case letters and upper-case letters come before lower-case.

Finally, if all you want is to check whether two strings are equal (and you don't care which one comes before the other if they are not), there is a slightly more convenient way of writing this using a method called equals:

```java
        if (word1.equals(word2)) {
            System.out.println("The words are equal!");
        }
```

[19]The full list can be downloaded from many places, including Wikipedia (http://en.wikipedia.org/wiki/List_of_Unicode_characters).

Exercise A

Say, for each of the following pairs of strings, whether `s.compareTo(t) < 0` would be true or false, assuming that `s` had the value on the left and `t` had the value on the right:

"A"	"9"
"Zurich"	"acapulco"
"Abba"	"ABBA"
"long_thing_with_a_$"	"long_thing_with_a_&"
"King@example.invalid"	"King Kong"

3.2 Two-way branches (`if ... else`)

The following program tells students whether they have passed their exam:

```java
 1 public class S3Example2 {
 2     public static void main(String[] args) {
 3         java.util.Scanner scan = new java.util.Scanner(System.in);
 4         System.out.print("Please key in your exam mark: ");
 5         int examMark = scan.nextInt();
 6         if (examMark >= 50) {
 7             System.out.println("A satisfactory result!");
 8         }
 9     }
10 }
```

Listing 12: S3Example2.java

What happens, in the case of this program, if a student's mark is less than 50? The program does nothing. This kind of **if** statement is a one-way branch. If the condition is true, we do something; if not, we do nothing. But in this case this seems unsatisfactory. If the exam mark is less than 50, we would like it to display "Sorry, you have failed." We could arrange this by including another test – `if (examMark < 50)` – or, better, we could do it by using the keyword **else**. So we replace lines 6 to 8 in the above program by:

```java
 6         if (examMark >= 50) {
 7             System.out.println("A satisfactory result!");
 8         } else {
 9             System.out.println("Sorry, you have failed.");
10         }
```

Listing 13: S3Example3.java

The **else** turns a one-way branch into a two-way branch. If the condition is true, do this; otherwise, do that.

A note about the curly braces. The curly braces have the effect of grouping all the statements within them into a programming unit called a *block*.[20]. Look at this example:

```java
        if (examMark >= 50) {
            System.out.println("A satisfactory result!");
            System.out.println("You may proceed with your project.");
        } else {
            System.out.println("Sorry, you have failed.");
        }
```

Listing 14: S3Example4.java

If the exam mark is greater than or equal to 50, the whole of the block is executed and both lines "A satisfactory result!" and "You may proceed with your project." will be printed. If the mark is lower than 50, the computer skips to the **else** and executes the "Sorry" line.

You will find that different programmers, and different textbooks, have different ideas about the precise placement of the curly braces. Some would set out the above fragment as:

```java
        if (examMark >= 50)
        {
            System.out.println("A satisfactory result!");
            System.out.println("You may proceed with your project.");
        }
        else
        {
            System.out.println("I'm afraid you have failed.");
        }
```

Listing 15: S3Example5.java

and there are other variations. Personally I prefer the first version because it is more compact but at the same time the structure of the program is very clear.

Suppose now that we wanted to give a different message to candidates who had done exceptionally well. Our first thought might be as follows:

```java
        if (examMark >= 70) {
            System.out.println("An exceptional result!");
            System.out.println("We expect a first-class project from you.");
        }
        if (examMark >= 50) {
            System.out.println("A satisfactory result!");
            System.out.println("You may proceed with your project.");
        } else {
            System.out.println("Sorry, you have failed.");
        }
```

[20] If we have only a single statement inside a branch of an **if** statement, making a block with the curly braces around the statement is not needed. However, I recommend always using the curly braces here, for clarity.

Listing 16: S3Example6.java

But this would not work quite right. It's OK for candidates with marks below 70, but candidates with marks greater than or equal to 70 would give the following output:

```
An exceptional result
We expect a first-class project from you.
A satisfactory result
You may proceed with your project.
```

The problem is that if a mark is greater than or equal to 70, it is also greater than 50. The first condition is true, so we get the "exceptional" part, but then the second condition is also true, so we get the "satisfactory" part. We want to proceed to the greater than 50 test only if the mark is below 70. We need another **else**:

```java
        if (examMark >= 70) {
            System.out.println("An exceptional result!");
            System.out.println("We expect a first-class project from you.");
        } else if (examMark >= 50) {
            System.out.println("A satisfactory result!");
            System.out.println("You may proceed with your project.");
        } else {
            System.out.println("Sorry, you have failed.");
        }
```

Listing 17: S3Example7.java

We can write more complex conditional expressions by joining them together using the operator ||, denoting **or**. For example, the following additional condition at the start of the last program checks if the exam mark keyed in is invalid, i.e., less than 0 or greater than 100:

```java
        if (examMark < 0 || examMark > 100) {
            System.out.println("Invalid mark entered - needs to be in the range 0..100");
        } else if (examMark >= 70) {
            System.out.println("An exceptional result!");
            System.out.println("We expect a first-class project from you.");
        } else if (examMark >= 50) {
            System.out.println("A satisfactory result!");
            System.out.println("You may proceed with your project.");
        } else {
            System.out.println("Sorry, you have failed.");
        }
```

Listing 18: S3Example8.java

Exercise B

Write a program that takes two numbers as input, one representing one partner's salary and the other representing the other partner's salary, and prints out whether or not their combined income makes them due for tax at the higher rate (i.e., it exceeds £40,000).

Exercise C

Extend the program about students' marks so that all the candidates get two lines of output, the unsuccessful ones getting "Sorry, you have failed." and "You may re-enter next year."

Exercise D

Write a program which takes two integers as input. If the first is exactly divisible by the second (such as 10 and 5 or 24 and 8, but not 10 and 3 or 24 and 7) it outputs "Yes", otherwise "No", except when the second is zero, in which case it outputs "Cannot divide by zero". Remember you can use the modulo operator ("%") to find out whether one number is divisible by another.

Exercise E

Write a program which takes an integer as its input, representing the time using the 24-hour clock. For example, 930 is 9.30 am; 2345 is 11.45 pm. Midnight is zero. The program responds with a suitable greeting for the time of day (e.g., "Good morning", "Good afternoon", or "Good evening"). If you want to make this a bit harder, make the program respond with a "?" if the time represented by the number is impossible, such as 2400, -5 or 1163.

4 Loops and booleans

How would you write a program to add up a series of numbers? If you knew that there were, say, four numbers, you might write this program:

```java
public class S4Example1 {
    public static void main(String[] args) {
        java.util.Scanner scan = new java.util.Scanner(System.in);
        int num1, num2, num3, num4;

        System.out.println("Please key in four numbers: ");
        System.out.print("> ");
        num1 = scan.nextInt();
        System.out.print("> ");
        num2 = scan.nextInt();
        System.out.print("> ");
        num3 = scan.nextInt();
        System.out.print("> ");
        num4 = scan.nextInt();
        int total = num1 + num2 + num3 + num4;
        System.out.println("Total: " + total);
    }
}
```

Listing 19: S4Example1.java

But a similar program to add up 100 numbers would be very long. More seriously, each program would need to be tailor-made for a particular number of numbers. It would be better if we could write one program to handle *any* series of numbers. We need a *loop*.

One way to create a loop is to use the keyword **while**. For example:

```java
public class S4Example2 {
    public static void main(String[] args) {
        int num = 0;
        while (num < 100) {
            num = num + 5;
            System.out.println(num);
        }
        System.out.println("Done looping!");
    }
}
```

Listing 20: S4Example2.java

It is not essential to indent the lines inside the loop, but it makes the program easier to read and it is a good habit to get into.

Having initialised the variable num to zero, the program checks whether the value of num is less than 100. It is, so it enters the loop. Inside the loop, it adds 5 to the value of num and then outputs this value (so the first thing that appears on the screen is a 5). Then it goes back to the **`while`** and checks whether the value of num is less than 100. The current value of num is 5, which is less than 100, so it enters the loop again. It adds 5 to num, so num takes the value 10, and then outputs this value. It goes back to the **`while`**, checks whether 10 is less than 100 and enters the loop again. It carries on doing this with num getting larger each time round the loop. Eventually num has the value 95. As 95 is still less than 100, it enters the loop again, adds 5 to num to make it 100 and outputs this number. Then it goes back to the **`while`** and this time sees that num is not less than 100. So it stops looping, goes on to the line after the end of the loop, and prints "Done looping!". The output of this program is the numbers 5, 10, 15, 20 and so on up to 95, 100, followed by "Done looping!".

Note the use of curly braces to mark the start and end of the loop. Each time round the loop the program does everything inside the curly braces. When it decides not to execute the loop again, it jumps to the point beyond the closing brace.

What would happen if the **`while`** line of this program was `while (num != 99)`? The value of num would eventually reach 95. The computer would decide that 95 was not equal to 99 and would go round the loop again. It would add 5 to num, making 100. It would now decide that 100 was not equal to 99 and would go round the loop again. Next time num would have the value 105, then 110, then 115 and so on. The value of num would never be equal to 99 and the computer would carry on for ever. This is an example of an *infinite loop*.

Note that the computer makes the test *before* it enters the loop. What would happen if the **`while`** line of this program was `while (num > 0)`? num begins with the value zero and the computer would first test whether this value was greater than zero. Zero is not greater than zero, so it would not enter the loop. It would skip straight to the end of the loop and finish, producing no output.

Exercise A

Write a program that outputs the squares of all the numbers from 1 to 10, i.e., the output will be the numbers 1, 4, 9, 16 and so on up to 100.

Exercise B

Write a program that asks the user to type in 5 numbers, and that outputs the largest of these numbers and the smallest of these numbers. So, for example, if the user types in the numbers 2456 457 13 999 35, the output will be as follows:

```
The largest number is 2456
The smallest number is 13
```

4.1 Booleans (true/false expressions)

So far we have just used integer and string values. But we can have values of other types and, specifically, we can have *boolean* values.[21] In Java, these are variables of type **boolean**. Its values are also called "truth values", and they are **true** and **false**.

```
        boolean positive = true;
```

Note that we do not have quote marks around the word **true**. The word **true**, without quote marks, is not a string; it's the name of a boolean value. Contrast it with:

```
        String stringVar = "true";
```

Here stringVar is assigned the four-character string "true", not the truth value **true**. By contrast, positive is a boolean variable and we cannot assign strings to it. It can hold only the values **true** or **false**.

You have already met boolean expressions. They are also called conditional expressions or conditions and they are the sort of expression you have in parentheses after **if** or **while**. When you evaluate a boolean expression, you get the value **true** or **false** as the result.

Consider the kind of integer assignment statement with which you are now familiar:

```
        num = count + 5;
```

The expression on the right-hand side, the count + 5, is an integer expression. That is, when we evaluate it, we get an integer value as the result.

Now consider a similar-looking boolean assignment statement:

```
        positive = num >= 0;
```

The expression on the right-hand side, the num >= 0, is a boolean expression. That is, when we evaluate it, we get a boolean value (**true**/**false**) as the result. You can achieve the same effect by the following more long-winded statement:

```
        if (num >= 0) {
            positive = true;
        } else {
            positive = false;
        }
```

The variable positive now stores a simple fact about the value of num at this point in the program. (The value of num might subsequently change, of course, but the value of positive will not change with it.) If, later in the program, we wish to test the value of positive, we need only write

```
        if (positive)
```

[21]The word "boolean" was coined in honour of an Irish mathematician of the nineteenth century called *George Boole*.

You can write `if (positive == true)` if you prefer, but the `== true` is redundant: `positive` itself is either true or false. We can also write

```
        if (!positive)
```

(pronounced *if not positive*) that is exactly the same as `if (positive == false)`. If `positive` is true, then `not positive` is false, and vice-versa.

Boolean variables are often called *flags*. The idea is that a flag has basically two states – either it's flying or it isn't.

So far we have constructed simple boolean expressions using the operators introduced in the last chapter – `x == y`, `s >= t` and so on – now augmented with negation (`!`). We can make more complex boolean expressions by joining simple ones with the operators *and* (`&&`) and *or* (`||`). For example, we can express "if x is a non-negative odd number" as `if (x >= 0 && x % 2 == 1)`. We can express "if the name begins with an A or an E" as:

```
if (name.substring(0,1).equals("A") || name.substring(0,1).equals("E"))
```

The rules for evaluating *and* and *or* are as follows:

	left	&&	right	left	||	right
1	true	**true**	true	true	**true**	true
2	true	**false**	false	true	**true**	false
3	false	**false**	true	false	**true**	true
4	false	**false**	false	false	**false**	false

Taking line 2 as an example, this says that, given that you have two simple boolean expressions joined by *and* and that the one on the left is true while the one on the right is false, the whole thing is false. If, however, you had the same two simple expressions joined by *or*, the whole thing would be true. As you can see, *and* is true if and only if both sides are true, otherwise it's false; *or* is false if and only if both sides are false, otherwise it's true.

Exercise D

Given that x has the value 5, y has the value 20, and s has the value `"Birkbeck"`, decide whether these expressions are true or false:

```
  (x == 5 && y == 10)
  (x < 0 || y > 15)
  (y % x == 0 && s.length() == 8)
  (s.substring(1,3).compareTo("Bir") == 0 || x / y > 0)
```

4.2 Back to loops

Returning to the problem of adding up a series of numbers, have a look at this program:

```java
public class S4Example3 {
    public static void main(String[] args) {
        java.util.Scanner scan = new java.util.Scanner(System.in);
        int total = 0;
        boolean finished = false;
        while (!finished) {
            System.out.println("Please enter a number (end with 0):");
            int num = scan.nextInt();
            if (num != 0) {
                total = total + num;
            } else {
                finished = true;
            }
        }
        System.out.println("Total is " + total);
    }
}
```

Listing 21: S4Example3.java

If we want to input a series of numbers, how will the program know when we have put them all in? That is the tricky part, which accounts for the added complexity of this program.

The variable `finished` is being used to help us detect when there are no more numbers. It is initialised to **`false`**. When the computer detects that the last number ("0") is introduced, it will be set to **`true`**. When `finished` is true, it means that we have finished reading in the input. The **`while`** loop begins by testing whether `finished` is **`true`** or not. If `finished` is not **`true`**, there is some more input to read and we enter the loop. If `finished` is **`true`**, there are no more numbers to input and we skip to the end of the loop.

The variable `total` is initialised to zero. Each time round the loop, the computer reads a new value into `num` and adds it to `total`. So `total` holds the total of all the values input so far.

Actually the program only adds `num` to `total` if a (non-zero) number has been entered. If the user enters something other than an integer – perhaps a letter or a punctuation mark – then the parsing of the input will fail (here done by the Scanner) and the program will stop, giving an error message.

A real-life program ought not to respond to a user error by aborting with a terse error message, though regrettably many of them do. However, dealing with this problem properly would make this little program more complicated than I want it to be at this stage.

You have to take some care in deciding whether a line should go in the loop or outside it. This program, for example, is only slightly different from the one above but it will perform differently:

```java
public class S4Example4 {
    public static void main(String[] args) {
        java.util.Scanner scan = new java.util.Scanner(System.in);
        int total = 0;
        boolean finished = false;
        while (!finished) {
            total = 0;
            System.out.println("Please enter a number (end with 0):");
            int num = scan.nextInt();
            if (num != 0) {
                total = total + num;
            } else {
                finished = true;
            }
        }
        System.out.println("Total is " + total);
    }
}
```

Listing 22: S4Example4.java

It resets `total` to zero *each time round the loop.* So `total` gets set to zero, has a value added to it, then gets set to zero, has another value added to it, then gets set to zero again, and so on. When the program finishes, `total` does not hold the total of all the numbers, just the value zero.

Here is another variation:

```java
public class S4Example5 {
    public static void main(String[] args) {
        java.util.Scanner scan = new java.util.Scanner(System.in);
        int total = 0;
        boolean finished = false;
        while (!finished) {
            System.out.println("Please enter a number (end with 0):");
            int num = scan.nextInt();
            if (num != 0) {
                total = total + num;
                System.out.println("Total is " + total);
            } else {
                finished = true;
            }
        }
    }
}
```

Listing 23: S4Example5.java

This one has the `println` line inside the loop, so it outputs the value of `total` each time round the loop. If you keyed in the numbers 4, 5, 6, 7 and 8, then, instead of just getting the total (30) as the output, you would get 4, 9, 15, 22 and then 30.

Exercise E

Write a program that reads a series of numbers, ending with 0, and then tells you how many numbers you have keyed in (other than the last 0). For example, if you keyed in the numbers 5, -10, 50, 22, -945, 12, 0 it would output "You have entered 6 numbers.".

A Obtaining, installing and running Java on a PC

These notes are for Java Development Kit (JDK), version 17. The JDK can run on any computer with Java installed, including the main operating systems like Windows, Linux, and Mac. You can download a free copy of the JDK from the web, e.g., at one of the following sites:

- `https://www.oracle.com/java/technologies/downloads/#java17`
- `https://jdk.java.net/java-se-ri/17`

If you do not know whether you have the JDK installed on your system, you can open a command prompt[22] and type[23] `javac -version`.

If it is installed, the result should be something similar to this:

```
> javac -version
javac 17.0.8.1
```

Note that the JDK is already installed in the computer labs at the School of Computing and Mathematical Sciences at Birkbeck (rooms 403, 404, 405, 407 in the Malet Street building).

Compiling and running Java programs from the command line

You do not need many extra tools to write, compile, and run Java programs. The Java compiler `javac` (part of the JDK) and the Java interpreter `java` (also part of the JDK) are run from the command line.

You first type your program in a text editor. Notepad or Notepad++ (for Windows), gedit or kwrite (for Linux), or TextMate (for Mac) are good options, but any simple editor will do. Note: If you use a word processor (like Microsoft Word, LibreOffice Writer, or TextEdit), make sure you save the file as plain text.

Give the filename a `.java` extension.

Open the command prompt. Go to the folder where you have saved your Java program using the command "cd" (change directory).

Suppose your program is in a file called `Myprog.java` in the current folder. You can compile your program by typing

```
javac Myprog.java
```

If the compiler is successful, it will create a file `Myprog.class` in the same folder. You can then type

```
java Myprog
```

to run your compiled program with the Java interpreter. You will see the result of running your program on the screen.

If however the compiler finds compilation errors, you will get error messages and you need to go back to the text editor, correct the program and save it again, then recompile it.

[22] You can find it in Windows in "Accessories"
[23] In some systems, you can type "-v" instead of "-version".

Compiling and running Java programs with an IDE

In principle, a text editor, `javac`, and `java` are enough to write, compile, and run Java programs on a computer.

In practice, software developers use additional specialised software to make developing and testing Java programs more comfortable and productive. This includes *Integrated Development Environments (IDEs)*. IDEs are essentially specialised text editors with extra support for the programming language that let you to compile and run your programs from within the IDE. They usually provide highlighting for keywords of the language (like **`public`** or **`class`**)[24] and will point out programming mistakes that the Java compiler would find in the editor part of the IDE as you type.

For Java, one of the most widely used IDEs is called IntelliJ. It is already installed on the computers at the School of Computing and Mathematical Sciences at Birkbeck. You can access it via the Windows Start Menu → JetBrains → IntelliJ IDEA 2021.2.1 (Ultimate 2021.2).[25]

IntelliJ is available for download here:

`https://www.jetbrains.com/idea/download/`

There is both a Community Edition of IntelliJ (free for everyone) and an Ultimate Edition (free trial for 30 days; students can apply for a free non-commercial educational licence at `https://www.jetbrains.com/community/education/#students`). For the *Programming in Java* module, the Community Edition is perfectly sufficient.

In class, we will often use IntelliJ to write our Java programs. A downside of IDEs is that they offer a *lot* of functionality, with many different buttons to click. This can be a bit overwhelming, especially at the beginning. It is up to you whether you use IntelliJ, a different IDE (like Eclipse or BlueJ), or a simple text editor to write your Java programs. If you feel that an IDE like IntelliJ is “too much” at the beginning, Notepad++ (also installed in the lab at the School) can be a good compromise between functionality and simplicity: the editor provides syntax highlighting for Java, but does not have integration with the Java compiler.

[24] In our notes for the *Programming in Java* module, we will use similar *syntax highlighting*.
[25] The name and version number of the menu entry may be different.