Chapter 1

Week 1: Here it all begins...

1.1 Introduction

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*. You write a program in a *programming language*. Many programming languages have been devised; well known ones include Fortran, BASIC, Cobol, Algol, Lisp, Ada, C++, Python, Ruby, Scala, and Java.

The language we will use in this document is named *Groovy*, and it is *very similar to the Java programming language but with a simplified syntax* that better fits the purposes of this introduction.

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. The following three lines are in BASIC, Cobol and C++ respectively:

```
FOR I% = 1 TO 100 STEP 5 DO

PERFORM RECORD-TRANSACTION UNTIL ALL-DONE = "T"

if (x > 100) cout << "Big"; else cout << "Small";
```

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

Example

```
println "Given a series of words, each on a separate line,"
println "this program finds the length of the longest word."
println "Please enter several sentences, one per line. "
println "Finish with a blank line."
max = 0
String str = "."
while (str.length() > 0) {
```

```
  str = System.console().readLine()
  if (str.length() > max) {
    max = str.length()
  }
}
if (max == 0) {
    println "There were no words."
} else {
    println "The longest sentence was " + max + " characters long."
}
```

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 Groovy program would behave incorrectly if in the 13^{th} line we wrote

```
if (max = 0)
```

instead of

```
if (max == 0)
```

or if in the 9^{th} line we put a semi-colon immediately after

```
if (s.length() > max)
```

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 curly braces) were missing, or if we wrote `leng` instead of `length()` or even `Length()` instead of `length()`.

1.1.1 Input and output

To get the computer to take some input and to store it in its memory, we write in Groovy:

```
str = System.console().readLine()
```

`System.console().readLine()` (pronounced *system dot console dot read line*) is a phrase which has a special meaning to the computer. The combination of these instructions means[1] "Take some input which is a sequence of characters and put it into the computer's memory."

[1]More precisely, it means "Take the current system — this computer, take its console — its keyboard and screen, and read a line of character input from it as entered by the user" (from the keyboard, not the screen because the user cannot type a line onto the screen... or could not before they invented touchscreens!)).

`str` by contrast, is a word that I (the programmer) have chosen. I could have used `characters` or `thingy` or `foodle` or just `s` or almost any word I wanted. (There are some restrictions which I will deal with in the next section.)

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 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.

```
str = System.console().readLine()
```

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 `str`." 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 `str`.

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; 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.

In Groovy, 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 `str` without telling the computer what `str` 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. In this case we are going to put strings of characters into `str`. Strings of characters, or simply *strings* as they are called in programming, are known in Groovy as `String`, with capital S. To tell the computer that we want to use a variable of type `String` called `str`, we write in our program:

```
String str
```

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
```

Your program can have as many variables as you want. In Groovy you don't have to declare all your variables at the start of the program, as is the case in some other languages. You can declare them in the middle of a program, but you mustn't try to use a variable before you've declared it.

If you want the computer to display (on the screen) the contents of one of its boxes, you use `print` followed by the name of the box. For example, we can print the contents of `str` by writing:

```
print str
```

If the contents of the box called `str` happened to be "Karl", then when the computer came to execute the instruction `print str` the word "Karl" would appear on the screen.

If we use `println` instead of `print` then an extra line is printed after the value.

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

Example

```
String str
str = System.console().readLine()
println str
```

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

It is customary to lay out programs like this 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 ingnores any indentations.

1.1.2 Compiling and running a program

You can learn the rudiments of Groovy from these notes just by doing the exercises with pencil and paper. It is not essential to run your programs on a computer. However, if you have a computer and are wondering how you run the 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 Groovy. 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. 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.

Now you issue a command and the computer executes your program. You can issue commands[2] from the command prompt in Windows (you can find the command prompt under *Start* → *Accesories*), 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, recompile and run it again. You might have to do this many times before the program behaves in the way you wanted.

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 Groovy 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 Groovy compiler, I attach a few notes telling you how you can obtain one (see Section A).

1.1.3 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 `str = System.console().readLine()` instruction, i.e., it reads the word or words that have been input. Then it executes the `println` 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 after the `println`, for example:

```
println "Hello"
```

We can use this to improve the above program:

Example

```
String str
print "Please key in a word: "
str = System.console().readLine()
print "The word was: "
println str
```

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

[2] 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.

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

Note the spaces in lines 2 and 4 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 `print` instruction. For example, we could combine the last two lines of the above program into one:

```
println "The word was " + str
```

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:

Example

```
String str
print "Please key in a word: "
str = System.console().readLine()
print "The word was: " + str
print "Now please key in another: "
str = System.console().readLine()
print "And this one was:" + str
```

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

```
Please key in a word: Tom
The word was TomNow please key in another: Jack
And this one was Jack
```

which is probably not what we wanted. If we want a new line after the the first word is printed, we need to use the `println` instruction we mentioned earlier:

Example

```
String str
print "Please key in a word: "
str = System.console().readLine()
println "The word was: " + str
print "Now please key in another: "
str = System.console().readLine()
print "And this one was:" + str
```

Now we would get:

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

Exercise A

Now pause and see if you can write:

1. a Groovy 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
   ```

1.1.4 Assignment and initialisation

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

```
str = "Some text"
```

This has the effect of putting the string "Some text" into the `str` box. Whatever content was in `str` 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 `str`, or that the variable `str` takes the value "Some text". The "=" symbol is the assignment operator in Groovy. We are not testing whether `str` has the value "Some text" or not, nor are we stating that `str` has the value "Some text"; we are *giving* the value "Some text" to `str`.

An assignment instruction such as this:

```
str = str + " and some more"
```

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

Using undefined values is a common cause of program bugs. 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 empty, you should declare it thus:

```
String str = ""
```

This is very like assignment since we are giving a value to `str`. But this is a special case where `str` 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 `println str` and `str = 5` It seems a natural word to use since we are giving the computer instructions. But the correct word is actually "statement". `println str` is an output statement, and `str = "Hello"` is an assignment statement. The lines in which we tell the computer about the variables we intend to use, such as `String str` or `String str = ""` are called variable *definitions.* They are also referred to as variable *declarations.* When you learn more about Groovy 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 B

Now see if you can write a program in Groovy 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
```

1.2 Variables, identifiers and expressions

In this section we will discuss integer variables, arithmetic expressions, identifiers, comments, and string variables.

1.2.1 Integer variables

As well as variables of type `String` we can have variables of type `int`. We call them integer variables or just *integers.* Integers can contain any[3] integer value like 0, -1, -200, or 1966.

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

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

```
int i
```

and also initialise it if we wanted to:

```
int i = 0
```

We can change the value of an integer with an assignment:

```
i = 1
```

or, if we had two integers `i` and `j`:

```
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 variables 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 str = "My name is " + "Inigo Montoya"
```

results in `str` 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 and string variables when using "+" ? In that case, Groovy converts the integers to strings and performs concatenation. For example, the small program

```
String str = "My name is " + "Inigo Montoya"
int n = 10 + 7
String text = str + " and I am " + n
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 integer variables.

1.2.2 Reading integers from the keyboard

In the last section, we saw how we can read a string of characters from the keyboard, using `System.console().readLine()`. 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. We can do this easily by *parsing* it using the command `Integer.parseInt()`:

Example

```
print "Please introduce a number: "
String str = System.console().readLine()
int n = Integer.parseInt(str)
println "The number was " + n
println "The next number is " + (n + 1)
```

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 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 be unpredictable. 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:

Example

```
print "Please introduce a number: "
String str = System.console().readLine()
println "The number was " + str
println "The next number is " + (str + 1)
```

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

1.2.3 Operator precedence

Groovy 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? Groovy, 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 brackets 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 `int` variables can only store integer values. If you try to store the result of a division in an `int` variable, you will store only the integer part and lose the decimal part. Try to understand what the following program does:

Example

```
int num = 5
println num
num = num + 2
println num
num = num / 3 * 6   Why is this outputting 13?
println num
println 7 + 15 % 4
num = 24 / 3 / 4
println num
num = 24 / (num / 4)
println num
```

A computer would get into difficulty if it tried to divide by zero. 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?

1.2.4 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. Groovy 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 `int` as the name of a variable, for example. There are many keywords but most of them are words that you are unlikely to choose. Ones that you might accidentally hit upon are `break`, `case`, `catch`, `class`, `const`, `continue`, `double`, `final`, `finally`, `float`, `import`, `long`, `new`, `return`, `short`, `switch`, `this`, `throw` and `try`. You should also avoid using words which, though not technically keywords, have special significance in the language, such as `println` and `String`.

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 `max` rather than `x` and `y`.

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 compiler, 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
```

Comments 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 all but ignores 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 Groovy and, if not, why not:

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

1.2.5 A bit more on String variables

We have done already many things with string variables: 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 so with strings.

For example, if you want to know how long a string is, you can find out with the `length()` *method*[4]. If `str` is the string, then `str.length()` gives its length. Do not forget the dot or the brackets: methods are always prefixed with a dot and followed by brackets[5] (sometimes empty, sometimes not). You could say, for example:

```
print "Please enter some text: "
String str = System.console().readLine()
int len = str.length()
println "The string " + str + " has " + len + " characters."
```

You can obtain a substring of a string with the `substring` method. For example if the string `s` has the value "`Silverdale`", then `s.substring(0,6)` will give you the first six letters, i.e., the string "`Silver`". The first number in brackets after the `substring` says where you want the substring to begin, and the second number says where you want it to end. *Note that the initial character of the string is treated as being at position 0, not position 1.* 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 `print` or assign them to other strings or combine them with the "+" operator. For example:

Example

```
String s = "software services"
s = s.substring(0,4) + s.substring(8,9) + s.substring(13)
println s
```

will output "soft ices".

[4] 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.

[5] Now you can see that console() and readLine() are also methods.

Exercise C

Say what the output of the following program fragment would be:

Example

```
String s = "artificial reality"
println s.substring(11,15) + " " + s.substring(0,3)
println s.substring(11).length()
```

Then run the program and see if you were right.

1.3 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 Groovy with the keyword `if`. We can write, for example:

Example

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

When the computer executes this, it first sees 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 `println` line and goes on to the next line.

Note that the conditional expression (`num == 180`) has to be in brackets, and that we use curly brackets 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 is a serious nuisance in Groovy, 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:

Example

```
int num1, num2
String s
print "Please key in a number: "
s = System.console().readLine()
num1 = Integer.parseInt(s)
print "And another: "
s = System.console().readLine()
```

```
num2 = Integer.parseInt(s)
if (num1 == num2) {
    println "They are the same"
}
```

1.3.1 Conditional expressions

Conditional expressions — the kind that follow an `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, but they can also be used with strings. Here their meaning corresponds to something like alphabetical order. For instance, `if (s < t)`, 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 < t)` 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[6] consists of a repertoire of more than 109,000 characters covering 93 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.

[6]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 < t` 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's College"`	`"King Kong"`

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

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

Example

```
int examMark
String s
print "Please key in your exam mark: "
s = System.console().readLine()
examMark = Integer.parseInt(s)
if (examMark >= 50) {
    println "A satisfactory result!"
}
```

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 "I'm afraid 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`, thus:

Example

```
if (exammark >= 50) {
    println "A satisfactory result!"
} else {
    println "I'm afraid you have failed."
}
```

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 inside them into a programming unit called a *block*[7]. Look at this example:

[7] A block consisting only of one statement does not need curly brackets. However, I recommend to use them always for clarity.

Example

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

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 "I'm afraid" 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:

Example

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

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:

Example

```
if (examMark >= 70) {
    println "An exceptional result!"
    println "We expect a first-class project from you."
}
if (examMark >= 50) {
    println "A satisfactory result"
    println "You may proceed with your project."
}
else {
    println "I'm afraid you have failed."
}
```

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 then 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`:

Example

```
if (examMark >= 70) {
    println "An exceptional result!"
    println "We expect a first-class project from you."
} else if (examMark >= 50) {
    println "A satisfactory result"
    println "You may proceed with your project."
} else {
    println "I'm afraid you have failed."
}
```

Exercise B

Write a program that takes two numbers, one representing a husband's salary and the other representing the wife's, and tells them 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 "I'm afraid 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. 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.

1.4 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:

Example

```
String s
int num1, num2, num3, num4

println "Please key in four numbers: "
print "> "
s = System.console().readLine()
num1 = Integer.parseInt(s)
print "> "
s = System.console().readLine()
num2 = Integer.parseInt(s)
print "> "
s = System.console().readLine()
num3 = Integer.parseInt(s)
print "> "
s = System.console().readLine()
num4 = Integer.parseInt(s)
int total = num1 + num2 + num3 + num4
println "Total: " + total
```

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:

Example

```
int num = 0
while (num < 100) {
    num = num + 5
    println num
}
```

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 initialized 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 and goes on to the line after the end of the loop. (In the case of this program, there are no more lines, so it finishes.) The output of this program is the numbers 5, 10, 15, 20 and so on up to 95, 100.

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.

1.4.1 Booleans (true/false expressions)

So far we have just used integer and string variables. But we can have variables of other types and, specifically, we can have *boolean* variables[8]. In Groovy, these are variables of type `boolean`. A variable of type `boolean` does not hold numbers; it can hold just the values `true` and `false`. We might declare and initialize a boolean variable thus:

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

```
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"
```

`stringvar` is a String variable which is being initialized with the four-character string "`true`"; we could assign any other string to `stringvar`. 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 and they are the sort of expression you have in brackets 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. And of course an integer value is exactly the right kind of thing to assign to an integer variable.

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. And of course a boolean value is exactly the right kind of thing to assign to a boolean variable. 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)
```

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) == "A" || name.substring(0,1) == "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 B

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 and s.length() == 8)
(s.substring(1,3) == "Bir" || x / y > 0)
```

False
True
True
False

1.4.2 Back to loops

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

Example

```
int total = 0
boolean finished = false
while (!finished) {
    println "Please enter a number (end with 0):"
```

```
    String s = System.console().readLine()
    int num = Integer.parseInt(s)
    if (num != 0) {
        total = total + num
    } else {
        finished = true
    }
}
println "Total is " + total
```

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 `boolean` variable `finished` is being used to help us detect when there are no more numbers. It is initialized 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 initialized 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 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:

Example

```
int total
boolean finished = false
while (!finished) {
    total = 0
    println "Please enter a number (end with 0):"
    String s = System.console().readLine()
    int num = Integer.parseInt(s)
    if (num != 0) {
```

```
        total = total + num
    } else {
        finished = true
    }
}
println "Total is " + total
```

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:

Example

```
int total = 0
boolean finished = false
while (!finished) {
    println "Please enter a number (end with 0):"
    String s = System.console().readLine()
    int num = Integer.parseInt(s)
    if (num != 0) {
        total = total + num
        println "Total is " + total
    } else {
        finished = true
    }
}
```

This one has the `print` 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 C

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.".

1.5 More on branches

You can write a lot of different programs with the little `if...else` and `while` constructs you have already learned and practiced, for executing different branches in your program and for running loops. There are, however, other constructs that you can use to make your programs clearer. In this section we will learn about the `do...while` loops, the

`switch...case` multiple branching feature, and the so-called ternary operator. In later chapters, we will also see how to use the very useful `for` and *for each* loops.

1.5.1 Multiple branching

Look at the following code, that loosely resembles an automatic phone-answering program:

Example

```
println "Please choose an option:"
println "For 'Checking you balance', please enter 1"
println "For 'Purchases', please enter 2"
println "For 'Refunds', please enter 3"
println "For 'Queries about the warranty', please enter 4"
println "For 'Returning faulty goods', please enter 5"
println "For any other query, please enter 0"
String s = System.console().readLine()
Integer choice = Integer.parseInt(s)
if (choice == 1) {
  // go and check balance
} else if (choice == 2) {
  // go and purchase something
} else if (choice == 3) {
  // go and process refunds
} else if (choice == 4) {
  // go and answer queries
} else if (choice == 5) {
  // return faulty goods
} else {
  // go and talk with a human operator
}
```

Such a long list of `if` branches can be a bit confusing for future readers. When there are many options, and especially if the options are finite and known in advance, it is better to make the multiple branching clearer by using a `switch...case` construct, as in the following example:

Example

```
println "Please choose an option:"
println "For 'Checking you balance', please enter 1"
println "For 'Purchases', please enter 2"
println "For 'Refunds', please enter 3"
println "For 'Queries about the warranty', please enter 4"
println "For 'Returning faulty goods', please enter 5"
println "For any other query, please enter 0"
String s = System.console().readLine()
```

```
Integer choice = Integer.parseInt(s)
switch (choice) {
case 1:
  // go and check balance
  break;
case 2:
  // go and purchase something
  break;
case 3:
  // go and process refunds
  break;
case 4:
  // go and answer queries
  break;
case 5:
  // return faulty goods
  break;
default:
  // go and talk with a human operator
  break;
}
```

As you can see, the multiple branches are introduced by the keyword `switch` followed by some parameter, *choice* in this case. Then several possible cases follow, each of them starting by a semicolon. The whole list of cases is inside two curly brackets, but the code of each case does not need curly brackets.

There are two important things to note in this program. First, there is a keyword `break` at the end of each case. This is very important. Without it, Groovy will finish the execution of one case and continue with the next one. Second, there is a default case that is executed if none of the other cases matches *choice*. This is also important: always write a default case for your `switch...case` structures. They are good for detecting errors in your program.

Exercise A

Write a program like the one in the last example, but add a little code for each case (some `println` statements will do) and remove the `break` keywords. Execute the program and see what happens.

Exercise B

Write a program similar to the one in the former example, but a bit better. Make the computer ask for an option and then execute the corresponding code in a `switch...case` structure. If the user introduces an invalid option (e.g. 9 in the example above), make

the computer say "That is not a valid option, please try again", and ask again until the user enters a valid numbers. Hint: You will need to use a loop and change the default case.

Exercise C

Write a program that uses Strings instead of integers for the `switch`. What happens?

It works fine. Nothing Happens!

What types can go inside a `switch`

If you have tried all the exercises above, you have noticed that Groovy complains when you try to use Strings inside a `switch...case`. The bad news is that we cannot use any type of data, but only things like integers, characters (also known as *chars* for brevity), and enumerated types (also known as *enums*, we will come to them in a second). The good news is that this is usually enough.

Didn't get any error

Characters can be used alone in any Groovy program, can be printed on the screen, and can also be combined to form our well-known Strings, as in the following code:

Example

```
char firstChar='J'
char secondChar='i'
char thirdChar='m'
String jimName = "" + firstChar + secondChar + thirdChar
println "Name: " + jimName
println "Letters: " + firstChar + "," + secondChar + "," + thirdChar
```

Notice that single quotes are used for characters and double quotes are used for Strings: ćís the character *c*, but "c" is a String with only the character *c* in it.

What are *enums*

Enumerated types are just lists of *tags* that can be useful to make your program more legible. They can be used to specify constants that have some special meaning. Defining an *enum* in Groovy is really easy:

```
enum Day {
  MONDAY,
  TUESDAY,
  WEDNESDAY,
  THURSDAY,
  FRIDAY,
  SATURDAY,
  SUNDAY,
}
```

Now, this new enum can be used in `switch` (or `if`) statements. Compare the clarity of this code:

```
switch (day) {
  case 1:
    // do something here for this day
    break;
  case 3:
    // do something here for this day
    break;
  ...
}
```

and this code:

```
switch (day) {
  case MONDAY:
    // do something here for this day
    break;
  case WEDNESDAY:
    // do something here for this day
    break;
  ...
}
```

In the second case, it is clear what each case is for. In the first piece of code, you need to guess: look at the code for each case, look at the surrounding code... it is harder. Programming is already a hard (and fun) activity. It is wise not to make it harder than it is; it does not make it any funnier.

There is a general lesson to learn here. What you see in the code above is an example of *magic numbers*: numbers that appear in your code and have an effect, but is difficult to understand what their effect is or why they are what they are (i.e. why a 1 and not a 5?). **Magic number make code harder to read and understand. Never use magic numbers in your code.**

1.5.2 The ternary operator

There is another form of writing a branch. It is not commonly used, but you should know it exists. It is the so-called *ternary operator*.

Most operators in Groovy are either *unary* (they take one argument, like logical not "!") or *binary* (they take two arguments, like logical and "&&" or addition "+"). There is only one ternary operator, with three arguments, that is really another fancy way of writing an `if...else` clause. We will illustrate how it works with a simple example:

```
print "Enter a number: "
int i = Integer.parseInt(System.console().readLine())
String s = (i > 5)? "Greater than 5" : "Not greater than 5"
println s
```

This code behaves exactly in the same as the following code:

```
print "Enter a number: "
int i = Integer.parseInt(System.console().readLine())
String s;
if (i > 5) {
  s = "Greater than 5"
} else {
  s = "Not greater than 5"
}
println s
```

As you see, the ternary operator takes three arguments: a boolean expression and two values. If the boolean expression is true, then the second argument is returned; if it is false, it returns the third argument.

In some cases the ternary operator can make your code clearer, but these cases are few and far between. In general, it is easier to write *bona fide* `if` statements. However, you may find the ternary operator in code written by others, and it is thus important to understand how it works.

Exercise D

Write a program that read a series of numbers from the user, until the user enters something that is not a number (like "END"). The program should then print how many numbers were positive, negative, or zero. Write the program using `if...else` clauses and using the ternary operator. Is there any difference? In which case it is easier to write the program? In which case it looks clearer to you?