Java Just in Time:
Collected concepts after chapter 05

John Latham, School of Computer Science, Manchester University, UK.

April 15, 2011

Contents

1 Computer basics

1.1 Computer basics: hardware (page 3)

The physical parts of a computer are known as **hardware**. You can see them, and touch them.

1.2 Computer basics: hardware: processor (page 3)

The **central processing unit** (**CPU**) is the part of the **hardware** that actually obeys instructions. It does this dumbly – computers are not inherently intelligent.

1.3 Computer basics: hardware: memory (page 3)

The **computer memory** is part of the computer which is capable of storing and retrieving **data** for short term use. This includes the **machine code** instructions that the **central processing unit** is obeying, and any other data that the computer is currently working with. For example, it is likely that an image from a digital camera is stored in the computer memory while you are editing or displaying it, as are the machine code instructions for the image editing program.

The computer memory requires electrical power in order to remember its data – it is **volatile memory** and will forget its contents when the power is turned off.

An important feature of computer memory is that its contents can be accessed and changed in any order required. This is known as **random access** and such memory is called **random access memory** or just **RAM**.

1.4 Computer basics: hardware: persistent storage (page 3)

For longer term storage of **data**, computers use **persistent storage** devices such as **hard disc**s and **DVD ROM**s. These are capable of holding much more information than **computer memory**, and are persistent in that they do not need power to remember the information stored on them. However, the time taken to store and retrieve data is *much* longer than for computer memory. Also, these devices cannot as easily be accessed in a random order.

1.5 Computer basics: hardware: input and output devices (page 3)

Some parts of the **hardware** are dedicated to receiving input from or producing output to the outside world. Keyboards and mice are examples of **input device**s. Displays and printers are

examples of **output device**s.

1.6   Computer basics: software (page 3)

One part of a computer you cannot see is its **software**. This is stored on **computer media**, such as **DVD ROM**s, and ultimately inside the computer, as lots of numbers. It is the instructions that the computer will obey. The closest you get to seeing it might be if you look at the silver surface of a DVD ROM with a powerful magnifying glass!

1.7   Computer basics: software: machine code (page 3)

The instructions that the **central processing unit** obeys are expressed in a language known as **machine code**. This is a very **low level language**, meaning that each instruction gets the computer to do only a very simple thing, such as the **addition** of two numbers, or sending a **byte** to a printer.

1.8   Computer basics: software: operating system (page 4)

A collection of **software** which is dedicated to making the computer generally usable, rather than being able to solve a *particular* task, is known as an **operating system**. The most popular examples for modern personal computers are Microsoft Windows, Mac OS X and Linux. The latter two are implementations of Unix, which was first conceived in the early 1970s. The fact it is still in widespread use today, especially by computer professionals, is proof that it is a thoroughly stable and well **design**ed and integrated platform for the expert (or budding expert) computer scientist.

1.9   Computer basics: software: application program (page 4)

A piece of **software** which is dedicated to solving a particular task, or application, is known as an **application program**. For example, an image editing program.

1.10   Computer basics: data (page 3)

Another part of the computer that you cannot see is its **data**. Like **software** it is stored as lots of numbers. Computers are processing and producing data all the time. For example, an image from a digital camera is data. You can only see the picture when you display it using

some image displaying or editing software, but even this isn't showing you the actual data that makes up the picture. The names and addresses of your friends is another example of data.

1.11   Computer basics: data: files (page 5)

When **data** is stored in **persistent storage**, such as on a **hard disc**, it is organized into chunks of related information known as **file**s. Files have names and can be accessed by the computer through the **operating system**. For example, the image from a digital camera would probably be stored in a jpeg file, which is a particular type of image file, and the name of this file would probably end in `.jpg` or `.jpeg`.

1.12   Computer basics: data: files: text files (page 5)

A **text file** is a type of **file** that contains **data** stored directly as **character**s in a human readable form. This means if you were to send the raw contents directly to the printer, you would (for most printers) be immediately able to read it. Examples of text files include `README.txt` that sometimes comes with **software** you are installing, or source text for a document to be processed by the LATEX[6] document processing system, such as the ones used to produce this book (prior to publication). As you will see shortly, a more interesting example for you, is computer program **source code** files.

1.13   Computer basics: data: files: binary files (page 5)

A **binary file** is another kind of **file** in which **data** is stored as **binary** (base 2) numbers, and so is not human readable. For example, the image from a digital camera is probably stored as a jpeg file, and if you were to look directly at its contents, rather than use some **application program** to display it, you would see what appears to be nonsense! An interesting example of a binary file is the **machine code** instructions of a program.

2   Java tools

2.1   Java tools: text editor (page 5)

A **text editor** is a program that allows the user to type and edit **text file**s. You may well have used `notepad` under Microsoft Windows; that is a text editor. More likely you have used `Microsoft Word`. If you have, you should note that it is not a text editor, it is a **word processor**. Although you can save your documents as text files, it is more common to save

them as `.doc` **file**s, which is actually a **binary file** format. `Microsoft Word` is not a good tool to use for creating program **source code** files.

If you are using an **integrated development environment** to support your programming, then the text editor will be built in to it. If not, there are a plethora of text editors available which are suited to Java programming.

2.2 Java tools: javac compiler (page 9)

The Java **compiler** is called `javac`. Java program source is saved by the programmer in a **text file** that has the suffix `.java`. For example, the text file `HelloWorld.java` might contain the source text of a program that prints `Hello world!` on the **standard output**. This text file can then be **compile**d by the Java compiler, by giving its name as a **command line argument**. Thus the command

```
javac HelloWorld.java
```

will produce the **byte code** version of it in the **file** `HelloWorld.class`. Like **machine code** files, byte code is stored in **binary file**s as numbers, and so is not human readable.

2.3 Java tools: java interpreter (page 9)

When the end user wants to run a Java program, he or she invokes the `java` **interpreter** with the name of the program as its **command line argument**. The program must, of course, have been **compile**d first! For example, to run the `HelloWorld` program we would issue the following command.

```
java HelloWorld
```

This makes the **central processing unit** run the interpreter or **virtual machine** `java`, which itself then **execute**s the program named as its first argument. Notice that the suffix `.java` is needed when compiling the program, but no suffix is used when **run**ning it. In our example here, the virtual machine finds the **byte code** for the program in the **file** `HelloWorld.class` which must have been previously produced by the **compiler**.

3 Operating environment

3.1 Operating environment: programs are commands (page 7)

When a program is **execute**d, the name of it is passed to the **operating system** which finds and loads the **file** of that name, and then starts the program. This might be hidden from you if you are used to starting programs from a menu or browser interface, but it happens nevertheless.

3.2 Operating environment: standard output (page 7)

When programs **execute**, they have something called the **standard output** in which they can produce text results. If they are **run** from some kind of **command line interface**, such as a Unix **shell** or a Microsoft Windows **Command Prompt**, then this output appears in that interface while the program is running. (If they are invoked through some **integrated development environment**, browser, or menu, then this output might get displayed in some pop-up box, or special console window.)

3.3 Operating environment: command line arguments (page 8)

Programs can be, and often are, given **command line argument**s to vary their behaviour.

4 Class

4.1 Class: programs are divided into classes (page 16)

In Java, the source text for a program is separated into pieces called **class**es. The source text for each class is (usually) stored in a separate **file**. Classes have a name, and if the name is `HelloWorld` then the text for the class is saved by the programmer in the **text file** `HelloWorld.java`.

One reason for dividing programs into pieces is to make them easier to manage – programs to perform complex tasks typically contain thousands of lines of text. Another reason is to make it easier to share the pieces between more than one program – such **software reuse** is beneficial to programmer productivity.

Every program has at least one class. The name of this class shall reflect the intention of the program. By convention, class names start with an upper case letter.

4.2   Class: public class (page 16)

A **class** can be declared as being **public**, which means it can be accessed from anywhere in the running Java environment; in particular the **virtual machine** itself can access it. The source text for a public class definition starts with the **reserved word `public`**. A reserved word is one which is part of the Java language, rather than a word chosen by the programmer for use as, say, the name of a program.

4.3   Class: definition (page 16)

After stating whether it has **public** access, a **class** next has the **reserved word `class`**, then its name, then a left brace ({), its body of text and finally a closing right brace (}).

```
public class MyFabulousProgram
{
  ... Lots of stuff here.
}
```

5   Method

5.1   Method: main method: programs contain a main method (page 17)

All Java programs contain a section of code called `main`, and this is where the computer will start to **execute** the program. Such sections of code are called **method**s because they contain instructions on how to do something. The **main method** always starts with the following heading.

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

5.2   Method: main method: is public (page 17)

The **main method** starts with the **reserved word `public`**, which means it can be accessed from anywhere in the running Java environment. This is necessary – the program could not be **run** by the **virtual machine** if the starting point was not accessible to it.

```
public
```

5.3   Method: main method: is static (page 17)

The **main method** of the program has the **reserved word static** which means it is allowed to be used in the **static context**. A context relates to the use of **computer memory** during the **run**ning of the program. When the **virtual machine** loads a program, it creates the static context for it, allocating computer memory to store the program and its **data**, etc.. A **dynamic context** is a certain kind of allocation of memory which is made later, during the running of the program. The program would not be able to start if the main method was not allowed to run in the static context.

```
public static
```

5.4   Method: main method: is void (page 17)

In general, a **method** (section of code) might calculate some kind of **function** or formula, and **return** the answer as a result. For example, the result might be a number. If a method returns a result then this must be stated in its heading. If it does not, then we write the **reserved word void**, which literally means (among other definitions) 'without contents'. The **main method** does not return a value.

```
public static void
```

5.5   Method: main method: is the program starting point (page 17)

The starting part, or **main method**, of the program is always called main, because it is the main part of the program.

```
public static void main
```

5.6   Method: main method: always has the same heading (page 18)

The **main method** of a Java program must always have a heading like this.

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

This is true even if we do not intend to use any **command line argument**s. So a typical single **class** program might look like the following.

5009

```
public class MyFabulousProgram
{
  public static void main(String[] args)
  {
    ... Stuff here to perform the task.
  }
}
```

6 Command line arguments

6.1 Command line arguments: program arguments are passed to main (page 17)

Programs can be given **command line argument**s which typically affect their behaviour. Arguments given to a Java program are strings of text **data**, and there can be any number of them in a **list**. In Java, String[] means 'list of strings'. We have to give a name for this list, and usually we call it args. The chosen name allows us to refer to the given data from within the program, should we wish to.

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

6.2 Command line arguments: program arguments are accessed by index (page 26)

The **command line argument**s given to the **main method** are a **list** of strings. These are the **text data string** arguments supplied on the **command line**. The strings are **index**ed by **integer**s (whole numbers) starting from zero. We can access the individual strings by placing the index value in square brackets after the name of the list. So, assuming that we call the list args, then args[0] is the first command line argument given to the program, if there is one.

6.3 Command line arguments: length of the list (page 79)

The **command line argument**s passed to the **main method** are a **list** of strings. We can find the length of a list by writing a dot followed by the word length, after the name of the list. For example, args.length yields an **int** value which is the number of items in the list args.

6.4   Command line arguments: list index can be a variable (page 79)

The **index** used to access the individual items from a **list** of strings does not have to be an **integer literal**, but can be an **int variable** or indeed an **arithmetic expression**. For example, the following code adds together a list of **integer**s given as **command line argument**s.

```
int sumOfArgs = 0;
for (int argIndex = 0; argIndex < args.length; argIndex = argIndex + 1)
  sumOfArgs = sumOfArgs + Integer.parseInt(args[argIndex]);
System.out.println("The sum is " + sumOfArgs);
```

The benefit of being able to use a **variable**, rather than an integer literal is that the access can be done in a **loop** which controls the value of the variable: thus the actual value used as the index is not the same each time.

7   Type

7.1   Type (page 36)

Programs can process various different kinds of **data**, such as numbers, text data, images etc.. The kind of a data item is known as its **type**.

7.2   Type: String: literal (page 18)

In Java, we can have a **string literal**, that is a fixed piece of text to be used as **data**, by enclosing it in double quotes. It is called a string literal, because it is a **type** of data which is a string of **character**s, exactly as listed. Such a piece of data might be used as a message to the user.

```
"This is a fixed piece of text data -- a string literal"
```

7.3   Type: String: literal: must be ended on the same line (page 21)

In Java, **string literal**s must be ended on the same line they are started on.

7.4   Type: String: literal: escape sequences (page 49)

We can have a **new line character** embedded in a **string literal** by using the **escape sequence**
\n. For example, the following code will print out three lines on **standard output**.

```
System.out.println("This text\nspans three\nlines.");
```

It will generate the following.

```
This text
spans three
lines.
```

There are other escape sequences we can use, including the following.

Sequence	Name	Effect
\b	Backspace	Moves the cursor back one place, so the next **character** will over-print the previous.
\t	Tab (horizontal tab)	Moves the cursor to the next 'tab stop'.
\n	New line (line feed)	Moves the cursor to the next line.
\f	Form feed	Moves to a new page on many (text) printers.
\r	Carriage return	Moves the cursor to the start of the current line, so characters will over-print those already printed.
\"	Double quote	Without the backslash escape, this would mark the end of the string literal.
\'	Single quote	This is just for consistency – we don't need to escape a single quote in a string literal.
\\	Backslash	Well, sometimes you want the backslash character itself.

Note: System.out.println() always ends the line with the platform dependent **line separator**, which on Linux is a new line character but on Microsoft Windows is a **carriage return character** followed by a new line character. In practice you may not notice the difference, but the above code is not strictly the same as using three separate System.out.println() calls and is not 100% portable.

7.5   Type: String: concatenation (page 26)

The + **operator**, when used with two string **operand**s, produces a string which is the **concatenation** of the two strings. For example "Hello " + "world" produces a string which is
Hello (including the space) concatenated with the string world, and so has the same value as
"Hello world".

There would not be much point concatenating together two **string literal**s like this, compared with having one string literal which is already the text we want. We would be more likely to use concatenation when at least one of the operands is not a fixed value, i.e. is a **variable** value. For example, `"Hello " + args[0]` produces a string which is `Hello ` (including the space) concatenated with the first **command line argument** given when the program is **run**.

The resulting string can be used anywhere that a single string literal could be used. For example `System.out.println("Hello " + args[0])` would print the resulting string on the **standard output**.

7.6   Type: String: conversion: from int (page 38)

The Java **operator** + is used for both **addition** and **concatenation** – it is an **overloaded operator**. If at least one of the **operand**s is a **text data string**, then Java uses concatenation, otherwise it uses addition. When only one of the two operands is a string, and the other is some other **type** of **data**, for example an **int**, the Java **compiler** is clever enough to understand the programmer wishes that data to be converted into a string before the concatenation takes place. It is important to note the difference between an **integer** and the decimal digit string we usually use to represent it. For example, the **integer literal** 123 is an **int**, a number; whereas the **string literal** `"123"` is a text data string – a string of 3 separate **character**s.

Suppose the **variable** `noOfPeopleToInviteToTheStreetParty` had the value 51, then the code

```
System.out.println("Please invite " + noOfPeopleToInviteToTheStreetParty);
```

would print out the following text.

```
Please invite 51
```

The number 51 would be converted to the string `"51"` and then concatenated to the string `"Please invite "` before being processed by `System.out.println()`.

Furthermore, for our convenience, there is a separate version of `System.out.println()` that takes a single **int** rather than a string, and prints its decimal representation. Thus, the code

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

has the same effect as the following.

```
System.out.println("" + noOfPeopleToInviteToTheStreetParty);
```

7.7 Type: String: conversion: from double (page 55)

The Java **concatenation operator**, +, for joining **text data string**s can also be used to convert a **double** to a string. For example, the **expression** `"" + 123.4` has the value `"123.4"`.

7.8 Type: int (page 36)

One of the **type**s of **data** we can use in Java is called **int**. A data item which is an **int** is an **integer** (whole number), such as `0`, `-129934` or `982375`, etc..

7.9 Type: double (page 54)

Another of the **type**s of **data** we can use in Java is known as **double**. A data item which is a **double** is a **real** (fractional decimal number), such as `0.0`, `-129.934` or `98.2375`, etc.. The type is called **double** because it uses a means of storing the numbers called **double precision**. On computers, real numbers are only approximated, because they have to be stored in a finite amount of memory space, whereas in mathematics we have the notion of infinite decimals. The double precision storage approach uses twice as much memory per number than the older **single precision** technique, but gives numbers which are much more precise.

7.10 Type: casting an int to a double (page 79)

Sometimes we have an **int** value which we wish to be regarded as a **double**. The process of conversion is known as **cast**ing, and we can achieve it by writing `(double)` in front of the **int**. For example, `(double)5` is the **double** value `5.0`. Of course, we are most likely to use this feature to cast the value of an **int variable**, rather than an **integer literal**.

8 Standard API

8.1 Standard API: System: out.println() (page 18)

The simplest way to print a message on **standard output** is to use:

```
System.out.println("This text will appear on standard output");
```

5014

System is a **class** (that is, a piece of code) that comes with Java as part of its **application program interface** (**API**) – a large number of classes designed to support our Java programs. Inside System there is a thing called out, and this has a **method** (section of code) called println. So overall, this method is called System.out.println. The method takes a string of text given to it in its brackets, and displays that text on the standard output of the program.

8.2   Standard API: Integer: parseInt() (page 41)

One simple way to turn a **text data string**, say "123" into the **integer** (whole number) it represents is to use the following.

```
Integer.parseInt("123");
```

Integer is a **class** (that is, a piece of code) that comes with Java. Inside Integer there is a **method** (section of code) called parseInt. This method takes a text data string given to it in its brackets, converts it into an **int** and **return**s that number. A **run time error** will occur if the given string does not represent an **int** value.

For example

```
int firstArgument;
firstArgument = Integer.parseInt(args[0]);
```

would take the first **command line argument** and, assuming it represents a number (i.e. it is a string of digits with a possible sign in front), would turn it into the number it represents, then store that number in firstArgument. If instead the first argument was some other text data string, it would produce a run time error.

8.3   Standard API: Double: parseDouble() (page 54)

One simple way to turn a **text data string**, say "123.456" into the **real** (fractional decimal number) it represents is to use the following.

```
Double.parseDouble("123.456");
```

Double is a **class** (that is, a piece of code) that comes with Java. Inside Double there is a **method** (section of code) called parseDouble. This method takes a text data string given to it in its brackets, converts it into an **double** and **return**s that number. A **run time error** will occur if the given string does not represent a number. For example

```
double firstArgument = Double.parseDouble(args[0]);
```

would take the first **command line argument** and, assuming it represents a number, would turn it into the number it represents, then store that number in firstArgument. To represent a number, the string must be a sequence of digits, possibly with a decimal point and maybe a negative sign in front. If instead the first argument was some other text data string, it would produce a run time error.

8.4   Standard API: Math: pow() (page 73)

Java does not have an **operator** to compute powers. Instead, there is a standard **class** called Math which contains a collection of useful **method**s, including pow(). This takes two numbers, separated by a comma, and gives the value of the first number raised to the power of the second.

For example, the **expression** Math.pow(2, 10) produces the value of 2^{10} which is 1024.

8.5   Standard API: Math: abs() (page 87)

Java does not have an **operator** to yield the **absolute value** of a number, that is, its value ignoring its sign. Instead, the standard **class** called Math contains a **method**, called abs. This method takes a number and gives its absolute value.

For example, the **expression** Math.abs(-2.7) produces the value 2.7, as does the expression Math.abs(3.4 - 0.7).

8.6   Standard API: Math: PI (page 87)

The standard **class** called Math contains a constant value called PI that is set to the most accurate value of π that can be represented using the **double** number **type**. We can refer to this value using Math.PI, as in the following example.

```
double circleArea = Math.PI * circleRadius * circleRadius;
```

9 Statement

9.1 Statement (page 18)

A command in a programming language, such as Java, which makes the computer perform a task is known as a **statement**. `System.out.println("I will output whatever I am told to")` is an example of a statement.

9.2 Statement: simple statements are ended with a semi-colon (page 18)

All simple **statement**s in Java must be ended by a semi-colon (`;`). This is a rule of the Java language **syntax**.

9.3 Statement: assignment statement (page 37)

An **assignment statement** is a Java **statement** which is used to give a value to a **variable**, or change its existing value. This is only allowed if the value we are assigning has a **type** which matches the type of the variable.

9.4 Statement: assignment statement: assigning a literal value (page 37)

We can assign a **literal value**, that is a constant, to a **variable** using an **assignment statement** such as the following.

```
noOfPeopleLivingInMyStreet = 47;
```

We use a single **equal sign** (`=`), with the name of the variable to the left of it, and the value we wish it to be given on the right. In the above example, the **integer literal** 47 will be placed into the variable `noOfPeopleLivingInMyStreet`. Assuming the variable was declared as an **int variable** then this assignment would be allowed because 47 is an **int**.

9.5 Statement: assignment statement: assigning an expression value (page 38)

More generally than just assigning a **literal value**, we can use an **assignment statement** to assign the value of an **expression** to a **variable**. For example, assuming we have the variable

```
int noOfPeopleToInviteToTheStreetParty;
```

then the code

```
noOfPeopleToInviteToTheStreetParty = noOfPeopleLivingInMyStreet + 4;
```

when **execute**d, would **evaluate** the expression on the right of the **equal sign** (=) and then place
the resulting value in the variable noOfPeopleToInviteToTheStreetParty.

9.6 Statement: assignment statement: updating a variable (page 70)

Java **variable**s have a name and a value, and this value can change. For example, the following
code is one way of working out the maximum of two numbers.

```
int x;
int y;
int z;
... Code here that gives values to x, y and z.

int maximumOfXYandZ = x;
if (maximumOfXYandZ < y)
  maximumOfXYandZ = y;
if (maximumOfXYandZ < z)
  maximumOfXYandZ = z;
```

See that the variable maximumOfXYandZ is given a value which then might get changed, so that
after the end of the second **if statement** it holds the correct value.

A very common thing we want the computer to do, typically inside a **loop**, is to perform a
variable update. This is when a variable has its value changed to a new value which is based
on its current one. For example, the code

```
count = count + 1;
```

will add one to the value of the variable count. Such examples remind us that an **assignment
statement** is *not* a definition of **equality**, despite Java's use of the single **equal sign**!

9.7   Statement: assignment statement: updating a variable: shorthand operators (page 87)

The need to undertake a **variable update** is so common, that Java provides various **shorthand operator**s for certain types of update.

Here are some of the most commonly used ones.

Operator	Name	Example	Longhand meaning
++	postfix increment	x++	x = x + 1
--	postfix decrement	x--	x = x - 1
+=	compound assignment: add to	x += y	x = x + y
-=	compound assignment: subtract from	x -= y	x = x - y
*=	compound assignment: multiply by	x *= y	x = x * y
/=	compound assignment: divide by	x /= y	x = x / y

The point of these **postfix increment**, **postfix decrement** and **compound assignment** operators is not so much to save typing when a program is being written, but to make the program easier to read. Once you are familiar with them, you will benefit from the shorter and more obvious code.

There is also a historical motivation. In the early days of the programming language C, from which Java inherits much of its **syntax**, these shorthand **operator**s caused the **compiler** to produce more efficient code than their longhand counterparts. The modern Java compiler with the latest optimization technology should remove this concern.

9.8   Statement: if else statement (page 60)

The **if else statement** is one way in Java of having **conditional execution**. It essentially consists of three parts: a **condition** or **boolean expression**, a **statement** which will be **execute**d when the condition is **true** (the **true part**), and another statement which will be executed when the condition is **false** (the **false part**). The whole statement starts with the **reserved word if**. This is followed by the condition, written in brackets. Next comes the statement for the true part, then the reserved word **else** and finally the statement for the false part.

For example, assuming we have the **variable** noOfPeopleToInviteToTheStreetParty containing the number suggested by its name, then the code

```
if (noOfPeopleToInviteToTheStreetParty > 100)
  System.out.println("We will need a big sound system!");
else
  System.out.println("We should be okay with a normal HiFi.");
```

5019

will cause the computer to compare the current value of `noOfPeopleToInviteToTheStreetParty` with the number `100`, and if it is greater then print out the message `We will need a big sound system!` or otherwise print out the message `We should be okay with a normal HiFi.` – it will never print out both messages. Notice the brackets around the condition and the semi-colons at the end of the two statements inside the if else statement. Notice also the way we lay out the code to make it easy to read, splitting the lines at sensible places and adding more **indentation** at the start of the two inner statements.

9.9   Statement: if else statement: nested (page 62)

The **true part** or **false part** statements inside an **if else statement** may be any valid Java **statement**, including other if else statements. When we place an if else statement inside another, we say they are **nested**.

For example, study the following code.

```
if (noOfPeopleToInviteToTheStreetParty > 300)
  System.out.println("We will need a Mega master 500 Watt amplifier!");
else
  if (noOfPeopleToInviteToTheStreetParty > 100)
    System.out.println("We will need a Maxi Master 150 Watt amplifier!");
  else
    System.out.println("We should be okay with a normal HiFi.");
```

Depending on the value of `noOfPeopleToInviteToTheStreetParty`, this will report one of *three* messages. Notice the way we have laid out the code above – this is following the usual rules that inner statements have more **indentation** than those they are contained in, so the second if else statement has more spaces because it lives inside the first one. However, typically we make an exception to this rule for if else statements nested in the false part of another, and we would actually lay out the code as follows.

```
if (noOfPeopleToInviteToTheStreetParty > 300)
  System.out.println("We will need a Mega master 500 Watt amplifier!");
else if (noOfPeopleToInviteToTheStreetParty > 100)
  System.out.println("We will need a Maxi Master 150 Watt amplifier!");
else
  System.out.println("We should be okay with a normal HiFi.");
```

This layout reflects our *abstract* thinking that the collection of statements is *one* construct offering three choices, even though it is implemented using two if else statements. This idea extends to cases where we want many choices, using many nested if else statements, without the indentation having to increase for each choice.

9.10   Statement: if statement (page 64)

Sometimes we want the computer to **execute** some code depending on a **condition**, but do nothing if the condition is `false`. We could implement this using an **if else statement** with an empty **false part**. For example, consider the following code.

```
if (noOfPeopleToInviteToTheStreetParty > 500)
   System.out.println("You may need an entertainment license!");
else ;
```

This will print the message if the **variable** has a value **greater than** 500, or otherwise execute the **empty statement** between the **reserved word** `else` and the semi-colon. Such empty statements do nothing, as you would probably expect!

It is quite common to wish nothing to be done when the condition is `false`, and so Java offers us the **if statement**. This is similar to the if else statement, except it simply does not have the word `else`, nor a false part.

```
if (noOfPeopleToInviteToTheStreetParty > 500)
   System.out.println("You may need an entertainment license!");
```

9.11   Statement: compound statement (page 66)

The Java **compound statement** is simply a list of any number of **statement**s between an opening left brace ({) and a closing right brace (}). You could think of the body of a **method**, e.g. main(), as being a compound statement if that is helpful. The meaning is straightforward: when the computer **execute**s a compound statement, it merely executes each statement inside it, in turn. More precisely of course, the Java **compiler** turns the **source code** into **byte code** that has this effect when the **virtual machine** executes the **compile**d program.

We can have a compound statement wherever we can have any kind of statement, but it is most useful when combined with statements which have another statement within them, such as **if else statement**s and **if statement**s.

For example, the following code reports three messages when the **variable** has a value **greater than** 500.

```
if (noOfPeopleToInviteToTheStreetParty > 500)
{
   System.out.println("You may need an entertainment license!");
   System.out.println("Also hire some street cleaners for the next day?");
   System.out.println("You should consider a bulk discount on lemonade!");
}
```

When the **condition** of the if statement is `true`, the body of the if statement is executed. This single statement is itself a compound statement, and so the three statements within it are executed. It is for this sort of purpose that the compound statement exists.

Note how we lay out the compound statement, with the opening brace at the same **indentation** as the if statement, the statements within it having extra indentation, and the closing brace lining up with the opening one.

Less usefully, a compound statement can be empty, as in the following example.

```
if (noOfPeopleToInviteToTheStreetParty > 500)
{
  System.out.println("You may need an entertainment license!");
  System.out.println("Also hire some street cleaners for the next day?");
  System.out.println("You should consider a bulk discount on lemonade!");
}
else {}
```

As you might expect, the meaning of an empty compound statement is the same as the meaning of an **empty statement**!

9.12 Statement: while loop (page 71)

The **while loop** is one way in Java of having **repeated execution**. It essentially consists of two parts: a **condition**, and a **statement** which will be **execute**d repeatedly while the condition is `true`. The whole statement starts with the **reserved word `while`**. This is followed by the condition, written in brackets. Next comes the statement to be repeated, known as the **loop body**.

For example, the following code is a long winded and inefficient way of giving the **variable** x the value 21.

```
int x = 1;
while (x < 20)
  x = x + 2;
```

The variable starts off with the value 1, and then repeatedly has 2 added to it, until it is no longer **less than** 20. This is when the **loop** ends, and x will have the value 21.

Notice the brackets around the condition and the semi-colon at the end of the statement inside the loop. Notice also the way we lay out the code to make it easy to read, splitting the lines at sensible places and adding more **indentation** at the start of the inner statement.

Observe the similarity between the while loop and the **if statement** – the *only* difference in **syntax** is the first word. There is a similarity in meaning too: the while loop executes its body zero or *more* times, whereas the if statement executes its body zero or *one* time. However, **if statement**s are *not* loops and you should avoid the common novice phrase "if loop" when referring to them!

9.13   Statement: for loop (page 77)

Another kind of **loop** in Java is the **for loop**, which is best suited for situations when the number of **iteration**s of the **loop body** is known before the loop starts. We shall describe it using the following simple example.

```
for (int count = 1; count <= 10; count = count + 1)
   System.out.println("Counting " + count);
```

The **statement** starts with the **reserved word for**, which is followed by three items in brackets, separated by semi-colons. Then comes the loop body, which is a single statement (often a **compound statement** of course). The first of the three items in brackets is a **for initialization**, which is performed once just before the loop starts. Typically this involves declaring a **variable** and giving an initial value to it, as in the above example int count = 1. The second item is the **condition** for continuing the loop – the loop will only **execute** and will continue to execute while that condition is **true**. In the example above the condition is count <= 10. Finally, the third item, a **for update**, is a statement which is executed at the *end* of each iteration of the loop, that is *after* the loop body has been executed. This is typically used to change the value of the variable declared in the first item, as in our example count = count + 1.

So the overall effect of our simple example is: declare count and set its value to 1, check that it is **less than** 10, print out Counting 1, add one to count, check again, print out Counting 2, add one to count, check again, and so on until the condition is **false** when the value of count has reached 11.

We do not really need the for loop, as the **while loop** is sufficient. For example, the code above could have been written as follows.

```
int count = 1;
while (count <= 10)
{
   System.out.println("Counting " + count);
   count = count + 1;
}
```

However you will see that the for loop version has placed together all the code associated with the control of the loop, making it easier to read, as well as a little shorter.

5023

There is one very subtle difference between the for loop and while loop versions of the example above, concerning the **scope** of the variable count, that is the area of code in which the variable can be used. Variables declared in the initialization part of a for loop can only be used in the for loop – they do not exist elsewhere. This is an added benefit of using for loops when appropriate: the variable, which is used solely to control the loop, cannot be accidentally used in the rest of the code.

10 Error

10.1 Error (page 20)

When we write the **source code** for a Java program, it is very easy for us to get something wrong. In particular, there are lots of rules of the language that our program must obey in order for it to be a valid program.

10.2 Error: syntactic error (page 20)

One kind of error we might make in our programs is **syntactic error**s. This is when we break the **syntax** rules of the language. For example, we might miss out a closing bracket, or insert an extra one, etc.. This is rather like missing out a word in a sentence of natural language, making it grammatically incorrect. The sign below, seen strapped to the back of a poodle, contains bad grammar – it has an is missing.

My other dog an Alsatian.

Syntactic errors in Java result in the **compiler** giving us an error message. They can possibly confuse the compiler, resulting in it thinking many more things are wrong too!

10.3 Error: semantic error (page 22)

Another kind of error we might make is a **semantic error**, when we obey the rules of the **syntax** but what we have written does not make any sense – it has no semantics (meaning). Another sign on a different poodle might say

My other dog is a Porsche.

which is senseless because a Porsche is a kind of car, not a dog.

10.4 Error: compile time error (page 22)

Java **syntactic error**s and many **semantic error**s can be detected for us by the **compiler** when it processes our program. Errors that the compiler can detect are called **compile time error**s.

10.5 Error: run time error (page 24)

Another kind of error we can get with programs is **run time error**s. These are errors which are detected when the program is **run** rather than when it is **compile**d. In Java this means the errors are detected and reported by the **virtual machine**, java.

Java calls run time errors **exception**s. Unfortunately, the error messages produced by java can look very cryptic to novice programmers. A typical one might be as follows.

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

You can get the best clue to what has caused the error by just looking at the words either side of the colon (:). In the above example, the message is saying that java cannot find the **method** called main.

10.6 Error: logical error (page 29)

The most tricky kind of error we can make in our programs is a **logical error**. For these mistakes we do not get an error message from the **compiler**, nor do we get one at **run time** from the **virtual machine**. These are the kind of errors for which the Java program we have written is meaningful as far as Java is concerned, it is just that our program does the wrong thing compared with what we wanted. There is no way the compiler or virtual machine can help us with these kinds of error: they are far, far too stupid to understand the *problem* we were trying to solve with our program.

For this reason, many logical errors, especially very subtle ones, manage to slip through undetected by human program testing, and end up as **bug**s in the final product – we have all heard stories of computer generated demands for unpaid bills with *negative* amounts, etc..

11 Execution

11.1 Execution: sequential execution (page 23)

Programs generally consist of more than one **statement**, in a list. We usually place these on separate lines to enhance human readability, although Java does not care about that. Statements in such a list are **execute**d sequentially, one after the other. More correctly, the Java **compiler** turns each one into corresponding **byte code**s, and the **virtual machine** executes each collection of byte codes in turn. This is known as **sequential execution**.

11.2 Execution: conditional execution (page 60)

Having a computer always obey a list of instructions in a certain order is not sufficient to solve many problems. We often need the computer to do some things only under certain circumstances, rather than every time the program is **run**. This is known as **conditional execution**, because we get the computer to **execute** certain instructions **condition**ally, based on the values of the **variable**s in the program.

11.3 Execution: repeated execution (page 70)

Having a computer always obey instructions just once within the **run** of a program is not sufficient to solve many problems. We often need the computer to do some things more than once. In general, we might want some instructions to be **execute**d, zero, one or many times. This is known as **repeated execution**, **iteration**, or **loop**ing. The number of times a loop of instructions is executed will depend on some **condition** involving the **variable**s in the program.

12 Code clarity

12.1 Code clarity: layout (page 31)

Java does not care how we lay our code out, as long as we use some **white space** to separate adjacent symbols that would otherwise be treated as one symbol if they were joined. For example `public void` with no space between the words would be treated as the single symbol `publicvoid` and no doubt cause a **compile time error**. So, if we were crazy, we could write all our program **source code** on one line with the minimum amount of space between symbols!

```
public class HelloSolarSystem{public static void main(String[]args){System.out.println("Hello Mercury!");System.out.println("He
```

Oh dear – it ran off the side of the page (and that was with a smaller font too). Let us split it up into separate lines so that it fits on the page.

```
public class HelloSolarSystem{public static void main(String[]args){
System.out.println("Hello Mercury!");System.out.println(
"Hello Venus!");System.out.println("Hello Earth!");System.out.println
("Hello Mars!");System.out.println("Hello Jupiter!");System.out.
println("Hello Saturn!");System.out.println("Hello Uranus!");System.
out.println("Hello Neptune!");System.out.println("Goodbye Pluto!");}}
```

Believe it or not, this program would still **compile** and **run** okay, but hopefully you will agree that it is not very easy for *us* to read. Layout is very important to the human reader, and programmers must take care and pride in laying out their programs as they are written. So we split our program *sensibly*, rather than arbitrarily, into separate lines, and use **indentation** (i.e. spaces at the start of some lines), to maximize the readability of our code.

12.2   Code clarity: layout: indentation (page 32)

A **class** contains structures **nested** within each other. The outer-most structure is the class itself, consisting of its heading and then containing it's body within the braces. The body contains items such as the **main method**. This in turn consists of a heading and a body contained within braces.

The idea of **indentation** is that the more nested a part of the code is, the more space it has at the start of its lines. So the class itself has no spaces, but its body, within the braces, has two or three. Then the body of the main method has two or three more. You should be consistent: always use the same number of spaces per nesting level. It is also a good idea to avoid using **tab character**s as they can often look okay on your screen, but not line up properly when the code is printed.

In addition, another rule of thumb is that opening braces ({) should have the same amount of indentation as the matching closing brace (}). You will find that principle being used throughout this book. However, some people prefer a style where opening braces are placed at the end of lines, which this author believes is less clear.

```
public class HelloWorld {

  public static void main(String[] args) {
    System.out.println("Hello world!");
  }
}
```

12.3   Code clarity: layout: splitting long lines (page 43)

One of the features of good layout is to keep our **source code** lines from getting too long. Very long lines cause the reader to have to work harder in horizontal eye movement to scan the code. When code with long lines is viewed on the screen, the reader either has to use a horizontal scroll bar to see them, or make the window so wide that other windows cannot be placed next to it. Worst of all, when code with long lines is printed on paper there is a good chance that the long lines will disappear off the edge of the page! At very least, they will be wrapped onto the next line making the code messy and hard to read.

So a good rule of thumb is to keep your source code lines shorter than 80 **character**s long. You can do this simply in most **text editor**s by never making the text window too wide and never using the horizontal scroll bar while writing the code.

When we do have a **statement** that is quite long, we simply split it into separate lines at carefully chosen places. When we choose such places, we bear in mind that most human readers scan down the left hand side of the code lines, rather than read every word. So, if a line is a continuation of a previous line, it is important to make this obvious at the start of it. This means using an appropriate amount of **indentation**, and choosing the split so that the first symbol on the continued line is not one which could normally start a statement.

A little thought at the writing stage quickly leads to a habit of good practise which seriously reduces the effort required to read programs once they are written. Due to **bug** fixing and general maintenance over the lifetime of a real program, the code is read many more times than it is written!

12.4   Code clarity: comments (page 82)

In addition to having careful layout and **indentation** in our programs, we can also enhance human readability by using **comment**s. These are pieces of text which are ignored by the **compiler**, but help describe to the human reader what the program does and how it works.

For example, every program should have comments at the start saying what it does and briefly how it is used. Also, **variable**s can often benefit from a comment before their declaration explaining what they are used for. As appropriate, there should be comments in the code too, *before* certain parts of it, explaining what these next **statement**s are going to do.

One form of comment in Java starts with the symbol //. The rest of that source line is then the text of the comment. For example

```
  // This is a comment, ignored by the compiler.
```

12.5 Code clarity: comments: marking ends of code constructs (page 83)

Another good use of **comment**s is to mark every closing brace (}) with a comment saying what code construct it is ending. The following skeleton example code illustrates this.

```
public class SomeClass
{
  public static void main(String[] args)
  {
    ...
    while (...)
    {
      ...
      ...
      ...
    } // while
    ...
  } // main

} // class SomeClass
```

13 Design

13.1 Design: hard coding (page 36)

Programs typically process input **data**, and produce output data. The input data might be given as **command line argument**s, or it might be supplied by the user through some **user interface** such as a **graphical user interface** or **GUI**. It might be obtained from **file**s stored on the computer.

Sometimes input data might be built into the program. Such data is said to be **hard coded**. This can be quite common while we are developing a program and we haven't yet written the code that obtains the data from the appropriate place. In other cases it might be appropriate to have it hard coded in the final version of the program, if such data only rarely changes.

13.2 Design: pseudo code (page 73)

As our programs get a little more complex, it becomes hard to write them straight into the **text editor**. Instead we need to **design** them *before* we implement them.

We do not design programs by starting at the first word and ending at the last, like we do when we implement them. Instead we can start wherever it suits us – typically at the trickiest bit.

Neither do we express our designs in Java – that would be a bad thing to do, as Java forces our mind to be cluttered with trivia which, although essential in the final code, is distracting during the design.

Instead, we express our **algorithm** designs in **pseudo code**, which is a kind of informal programming language that has all unnecessary trivia ignored. So, for example, we do not bother writing the semi-colons at the end of **statement**s, or the brackets round **condition**s etc.. We might not bother writing the **class** heading, nor the **method** heading, if it is obvious to us what we are designing. And so on.

Also, during design in pseudo code, we can vary the level of **abstraction** to suit us – we do not have to be constrained to use only the features that are available in Java.

14 Variable

14.1 Variable (page 36)

A **variable** in Java is an entity that can hold a **data** item. It has a name and a value. It is rather like the notion of a variable in algebra (although it is not quite the same thing). The name of a variable does not change – it is carefully chosen by the programmer to reflect the meaning of the entity it represents in relation to the problem being solved by the program. However, the *value* of a variable can (in general) be changed – we can vary it. Hence the name of the concept: a **variable** is an entity that has a (possibly) varying value.

The Java **compiler** implements variables by mapping their names onto **computer memory** locations, in which the values associated with the variables will be stored at **run time**.

So one view of a variable is that it is a box, like a pigeon hole, in which a value can be placed. If we wish, we can get the program to place a different value in that box, replacing the previous; and we can do this as many times as we want to.

Variables only have values at run time, when the program is **run**ning. Their names, created by the programmer, are already fixed by the time the program is **compile**d. Variables also have one more attribute – the **type** of the data they are allowed to contain. This too is chosen by the programmer.

14.2   Variable: int variable (page 37)

In Java, **variable**s must be declared in a **variable declaration** before they can be used. This is done by the programmer stating the **type** and then the name of the variable. For example the code

```
int noOfPeopleLivingInMyStreet;
```

declares an **int variable**, that is a variable the value of which will be an **int**, and which has the name noOfPeopleLivingInMyStreet. Observe the semi-colon (;) which, according to the Java **syntax** rules, is needed to terminate the variable declaration. At **run time**, this variable is allowed to hold an **integer** (whole number). Its value can change, but it will always be an **int**. The name of a variable should reflect its intended meaning. In this case, it would seem from its name that the programmer intends the variable to always hold the number of people living in his or her street. The programmer would write code to ensure that this meaning is always reflected by its value at run time.

By convention, variable names start with a lower case letter, and consist of a number of words, with the first letter of each subsequent word capitalized.

14.3   Variable: a value can be assigned when a variable is declared (page 42)

Java permits us to assign a value to a **variable** at the same time as declaring it. You could regard this as a kind of **assignment statement** in which the variable is also declared at the same time. For example

```
int noOfHousesInMyStreet = 26;
```

14.4   Variable: double variable (page 54)

We can declare **double variable**s in Java, that is **variable**s which have the **type double**. For example the code

```
double meanAgeOfPeopleLivingInMyHouse;
```

declares a **variable** of type **double**, with the name meanAgeOfPeopleLivingInMyHouse. At **run time**, this variable is allowed to hold a **double data** item, that is a **real** (fractional decimal number). The value of this variable can change, but it will always be a **double**, including of course, approximations of *whole* numbers such as 40.0.

15 Expression

15.1 Expression: arithmetic (page 38)

We can have **arithmetic expression**s in Java rather like we can in mathematics. These can contain **literal value**s, that is constants, such as the **integer literal**s 1 and 18. They can also contain **variable**s which have already been declared, and **operator**s to combine sub-expressions together. Four common **arithmetic operator**s are **addition** (+), **subtraction** (-), **multiplication** (*) and **division** (/). Note the use of an asterisk for multiplication, and a forward slash for division – computer keyboards do not have multiply or divide symbols.

These four operators are **binary infix operator**s, because they take two **operand**s, one on either side of the operator. + and - can also be used as the **unary prefix operator**s, **plus** and **minus** respectively, as in -5.

When an **expression** is **evaluate**d (**expression evaluation**) Java replaces each variable with its current value and works out the result of the expression depending on the meaning of the operators. For example, if the variable noOfPeopleLivingInMyStreet had the value 47 then the expression noOfPeopleLivingInMyStreet + 4 would evaluate to 51.

15.2 Expression: arithmetic: int division truncates result (page 52)

The four **arithmetic operator**s, +, -, * and / of Java behave very similarly to the corresponding operators in mathematics. There is however one serious difference to look out for. When the **division operator** is given two **integer**s (whole numbers) it uses **integer division** which always yields an integer as its result, by throwing away any fractional part of the answer. So, 8 / 2 gives the answer 4 as you might expect, but 9 / 2 also gives 4 – not 4.5 as it would in mathematics. It does not round to the nearest whole number, it always rounds towards zero. In mathematics 15 / 4 gives 3.75. In Java it yields 3 not 4.

15.3 Expression: arithmetic: associativity and int division (page 52)

Like the **operator**s + and -, the operators * and / have equal **operator precedence** (but higher than + and -) and also have **left associativity**.

However, there is an extra complication to consider because the Java / operator truncates its answer when given two **integer**s. Consider the following two **arithmetic expression**s.

Expression	Implicit brackets	Value
9 * 4 / 2	(9 * 4) / 2	18
9 / 2 * 4	(9 / 2) * 4	16

In mathematics one would expect to get the same answer from both these **expression**s, but not in Java!

15.4 Expression: arithmetic: double division (page 55)

The Java **division operator**, /, uses **double division** and produces a `double` result if at least one of its **operand**s is a `double`. The result will be the best approximation to the actual answer of the division.

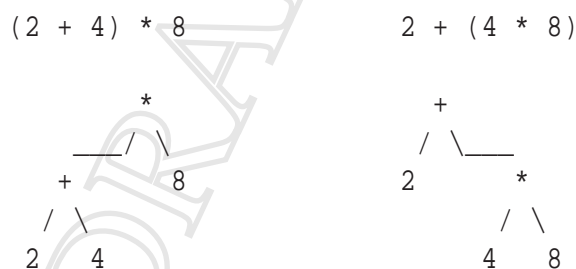| Expression	Result	Type of Result
8 / 2	4	int
8 / 2.0	4.0	double
9 / 2	4	int
9 / 2.0	4.5	double
9.0 / 2	4.5	double
9.0 / 2.0	4.5	double

15.5 Expression: brackets and precedence (page 45)

In addition to **operator**s and **variable**s, **expression**s in Java can have round brackets in them. As in mathematics, brackets are used to define the structure of the expression by grouping parts of it into sub-expressions. For example, the following two expressions have different structures, and thus very different values.

```
(2 + 4) * 8
2 + (4 * 8)
```

The value of the first expression is made from the **addition** of 2 and 4 and then **multiplication** of the resulting 6 by 8 to get 48. The second expression is **evaluate**d by multiplying 4 with 8 to get 32 and then adding 2 to that result, ending up with 34.

To help us see the structure of these two expressions, let us draw them as **expression tree**s.

```
(2 + 4) * 8                2 + (4 * 8)


        *                      +
    __ / \                   / \__
    +     8                 2      *
   / \                            / \
  2   4                          4   8
```

5033

What if there were no brackets?
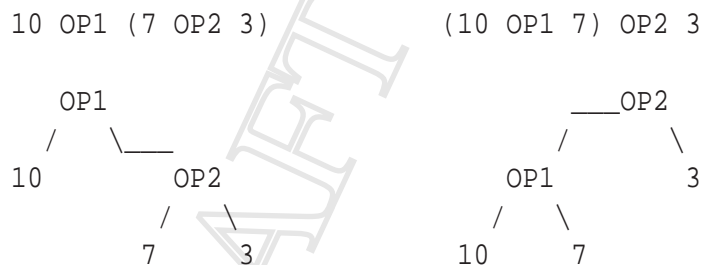
```
2 + 4 * 8
```

Java allows us to have expressions without any brackets, or more generally, without brackets around *every* sub-expression. It provides rules to define what the structure of such an expression is, i.e., where the missing brackets should go. If you look at the 4 in the above expression, you will see that it has an operator on either side of it. In a sense, the + operator and the * operator are both fighting to have the 4 as an **operand**. Rather like a tug of war, + is pulling the 4 to the left, and * is tugging it to the right. The question is, which one wins? Java, as in mathematics, provides the answer by having varying levels of **operator precedence**. The * and / operators have a higher precedence than + and -, which means * fights harder than +, so it wins! 2 + 4 * 8 evaluates to 34.

15.6   Expression: associativity (page 48)

The principle of **operator precedence** is insufficient to disambiguate all **expression**s which are not fully bracketed. For example, consider the following expressions.

```
10 + 7 + 3
10 + 7 - 3
10 - 7 + 3
10 - 7 - 3
```

In all four expressions, the 7 is being fought over by two **operator**s which have the same precedence: either two +, two -, or one of each. So where should the missing brackets go? The **expression tree**s could have one of the two following structures, where OP1 is the first operator, and OP2 is the second.

```
   10 OP1 (7 OP2 3)          (10 OP1 7) OP2 3

       OP1                          ___OP2
      /  \___                      /       \
    10       OP2               OP1          3
            /  \              /  \
           7    3            10    7
```

Let us see whether it makes a difference to the results of the expressions.

Expression	Value
`(10 + 7) + 3`	20
`10 + (7 + 3)`	20
`(10 + 7) - 3`	14
`10 + (7 - 3)`	14
`(10 - 7) + 3`	6
`10 - (7 + 3)`	0
`(10 - 7) - 3`	0
`10 - (7 - 3)`	6

As you can see, it does make a difference sometimes – in these cases when the first operator is **subtraction** (`-`). So how does Java resolve this problem? As in mathematics, Java operators have an **operator associativity** as well as a precedence. The operators `+`, `-`, `*` and `/` all have **left associativity** which means that when two of these operators of equal precedence are both fighting over one **operand**, it is the left operator that wins. If you like, the tug of war takes place on sloping ground with the left operator having the advantage of being lower down than the right one!

Expression	Implicit brackets	Value
`10 + 7 + 3`	`(10 + 7) + 3`	20
`10 + 7 - 3`	`(10 + 7) - 3`	14
`10 - 7 + 3`	`(10 - 7) + 3`	6
`10 - 7 - 3`	`(10 - 7) - 3`	0

The operators `*` and `/` also have equal precedence (but higher than `+` and `-`) so similar situations arise with those too.

15.7 Expression: boolean (page 60)

An **expression** which when **evaluate**d yields either `true` or `false` is known as a **condition**, and is typically used for controlling **conditional execution**. Conditions are also called **boolean expression**s.

15.8 Expression: boolean: relational operators (page 60)

Java gives us six **relational operator**s for comparing values such as numbers, which we can use to make up **condition**s. These are all **binary infix operator**s, that is they take two **operand**s, one either side of the **operator**. They yield `true` or `false` depending on the given values.

Operator	Title	Description
==	Equal	This is the **equal** operator, which provides the notion of **equality**. a == b yields **true** if and only if the value of a is the same as the value of b.
!=	Not equal	This is the **not equal** operator, providing the the notion of not equality. a != b yields **true** if and only if the value of a is *not* the same as the value of b.
<	Less than	This is the **less than** operator. a < b yields **true** if and only if the value of a is less than the value of b.
>	Greater than	This is the **greater than** operator. a > b yields **true** if and only if the value of a is greater than the value of b.
<=	Less than or equal	This is the **less than or equal** operator. a <= b yields **true** if and only if the value of a is less than value of b, or is equal to it.
>=	Greater than or equal	This is the **greater than or equal** operator. a >= b yields **true** if and only if the value of a is greater than value of b, or is equal to it.