Java Just in Time:
Collected concepts after chapter 11

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.

3.4 Operating environment: standard input (page 187)

In addition to **standard output**, when programs **execute** they also have a **standard input** which allows text **data** to be entered into the program as it runs. If they are **run** from some kind of **command line interface**, such as a Unix **shell** or a Microsoft Windows **Command Prompt**, then this input is typically typed on the keyboard by the end user.

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.
}
```

4.4 Class: objects: contain a group of variables (page 158)

We can group a collection of **variable**s into one entity by creating an **object**. For example, we might wish to represent a point in two dimensional space using an *x* and a *y* value to make up a coordinate. We would probably wish to combine our `x` and `y` variables into a single object, a `Point`.

4.5 Class: objects: are instances of a class (page 158)

Before we can make **object**s, we need to tell Java how the objects are to be **construct**ed. For example, to make a `Point` object, we would need to tell Java that there are to be a pair of **variable**s inside it, called `x` and `y`, and tell it what **type**s these variables have, and how they get their values. We achieve this by writing a **class** which will act as a template for the creation of objects. We need to write such a template class for each kind of object we wish to have. For example, we would write a `Point` class describing how to make `Point` objects. If, on the other hand, we wanted to group together a load of variables describing attributes of wardrobes, so we could make objects each of which represents a single wardrobe, then we would probably call that class `Wardrobe`. Java lets us choose any name that we feel is appropriate, except **reserved word**s (although by convention we always start the name with a capital letter).

Once we have described the template, we can get Java to make objects of that class at **run time**. We say that these objects are **instance**s of the class. So, for example, particular `Point` objects would all be instances of the `Point` class. We can create as many different `Point` objects as we wish, each containing its own `x` and `y` variables, all from the one template, the `Point` class.

4.6 Class: objects: this reference (page 180)

Sometimes, in **constructor method**s or in **instance method**s of a **class** we wish to refer to the **object** that the constructor is creating, or to which the instance method belongs. For this purpose, whenever the **reserved word `this`** is used in or as an **expression** it means a **reference**

to the object that is being created by the constructor or that owns the instance method, etc.. We can only use the **this reference** in places where it makes sense, such as constructor methods, instance methods and **instance variable** initializations. So, `this` (when used in this way) behaves somewhat like an extra instance variable in each object, automatically set up to contain a reference to that object.

For example, in a `Point` class we may wish to have an instance method that yields a point which is half way between the origin and `this` point.

```
public Point halfThisPoint()
{
  return halfWayPoint(new Point(0, 0));
} // halfThisPoint
```

An alternative implementation would be as follows.

```
public Point halfThisPoint()
{
  return new Point(0, 0).halfWayPoint(this);
} // halfThisPoint
```

4.7 Class: objects: may be mutable or immutable (page 193)

Sometimes when we **design** a **class** we desire that the **instance**s of it are **immutable object**s. This means that once such an **object** has been **construct**ed, its **object state** cannot be changed. That is, there is no way for the values of the **instance variable**s to be altered after the object is constructed.

By contrast, objects which can be altered are known as **mutable object**s.

4.8 Class: is a type (page 161)

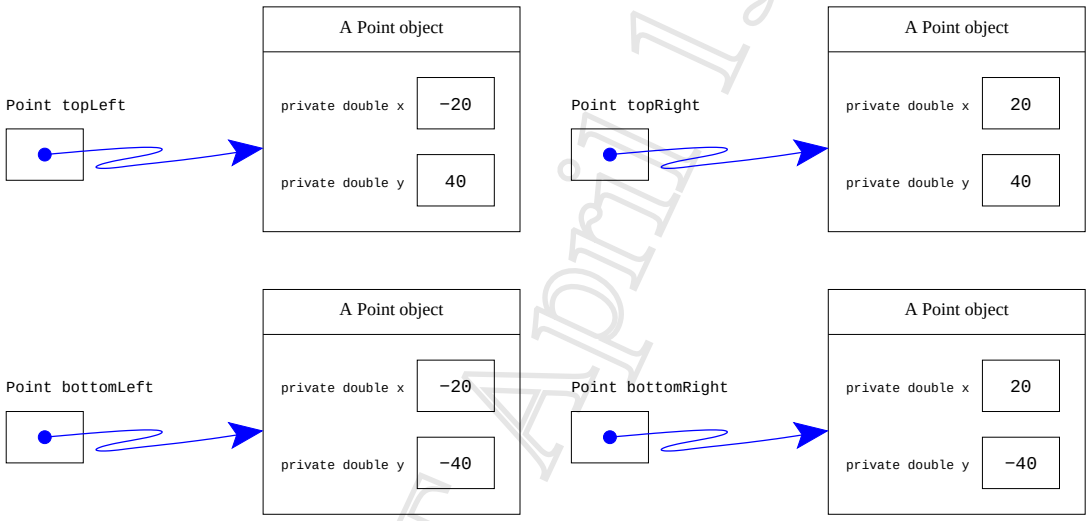
A **type** is essentially a **set** of values. The `int` type is all the whole numbers that can be represented using 32 **binary digit**s, the `double` type is all the **real** numbers that can be represented using the **double precision** technique and the `boolean` type contains the values `true` and `false`. A **class** can be used as a template for creating **object**s, and so is regarded in Java as a type: the set of all objects that can be created which are **instance**s of that class. For example, a `Point` class is a type which is the set of all `Point` objects that can be created.

4.9 Class: making instances with new (page 162)

An **instance** of a **class** is created by calling the **constructor method** of the class, using the **reserved word** `new`, and supplying **method argument**s for the **method parameter**s. At **run time** when this code is **execute**d, the Java **virtual machine**, with the help of the constructor method code, creates an **object** which is an instance of the class. Although it is not stated in its heading, a constructor method always **return**s a value, which is a **reference** to the **new**ly created object. This reference can then be stored in a **variable**, if we wish. For example, if we have a `Point` class, then we might have the following code.

```
Point topLeft     = new Point(-20, 40);
Point bottomLeft  = new Point(-20, -40);
Point topRight    = new Point(20, 40);
Point bottomRight = new Point(20, -40);
```

This declares four variables, of **type** `Point` and creates four instances of the class `Point` representing the four corners of a rectangle. The four variables each contain a reference to one of the points. This is illustrated in the following diagram.



All four `Point` objects each have two **instance variable**s, called `x` and `y`.

4.10 Class: accessing instance variables (page 164)

The **instance variable**s of an **object** can be accessed by taking a **reference** to the object and appending a dot (`.`) and then the name of the **variable**. For example, if the variable `p1` contains a reference to a `Point` object, and `Point` objects have an instance variable called `x`, then the code `p1.x` is the instance variable `x`, belonging to the `Point` referred to by `p1`.

4.11 Class: importing classes (page 188)

At the start of the source **file** for a Java **class** we can write one or more **import statement**s. These start with the **reserved word** `import` and then give the **fully qualified name** of a class that lives in some **package** somewhere, followed by a semi-colon(`;`). An **import** for a class permits us to talk about it from then on, by using only its class name, rather than having to always write its fully qualified name. For example, importing `java.util.Scanner` would mean that every time we refer to `Scanner` the Java **compiler** knows we really mean `java.util.Scanner`.

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

If we wish, we can import all the classes in a package using a * instead of a class name.

```
import java.util.*;
```

Many programmers consider this to be lazy, and it is better to import exactly what is needed, if only to help show precisely what is used by the class. There is also the issue of ambiguity: if two different packages have classes with the same name, but this class only needs one of them, then the lazy approach would cause an unnecessary problem.

However, every Java program has an automatic import for every class in the standard **package** `java.lang`, because these classes are used so regularly. That is why we can refer to `java.lang.System` and `java.lang.Integer`, etc. as just `System` and `Integer`, etc.. In other words, every class always implicitly includes the following import statement for convenience.

```
import java.lang.*;
```

4.12 Class: stub (page 191)

During development of a program with several **class**es, we often produce a **stub** for the classes we have not yet implemented. This just contains some or all of the **public** items of the class, with empty, or almost empty, bodies for the **method**s. In other words, it is the bare minimum needed to allow the classes we have so far developed to be **compile**d.

Any **non-void method**s are written with a single **return statement** to yield some temporary value of the right **type**.

These stubs are then developed into the full class code at some later stage.

5 Method

5.1 Method (page 118)

A **method** in Java is a section of code, dedicated to performing a particular task. All programs have a **main method** which is the starting point of the program. We can have other methods too, and we can give them any name we like – although we should always choose a name which suits the purpose. By convention, method names start with a lower case letter. For example, `System.out.println()` is a method which prints a line of text. Apart from its slightly strange spelling, the name `println` does reflect the meaning of the method.

5.2 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)
```

5.3 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
```

5.4 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
```

5.5 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
```

5.6 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
```

5.7 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.

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

5.8 Method: private (page 118)

A **method** should be declared with a **private** visibility **modifier** if it is not intended to be usable from outside the **class** it is defined in. This is done by writing the **reserved word `private`** instead of **`public`** in the heading.

5.9 Method: accepting parameters (page 118)

A **method** may be given **method parameter**s which enable it to vary its effect based on their values. This is similar to a program being given **command line argument**s, indeed the arguments given to a program are passed as parameters to the **main method**.

Parameters are declared in the heading of the method. For example, main methods have the following heading.

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

The text inside the brackets is the declaration of the parameters. A method can have any number of parameters, including zero. If there is more than one, they are separated by commas (,). Each parameter consists of a **type** and a name. For example, the following method is given two parameters, a **`double`** and an **`int`**.

```
private static void printHeightPerYear(double height, int age)
{
  System.out.println("At age " + age + ", height per year ratio is "
                     + height / age);
} // printHeightPerYear
```

You should think of parameters as being like **variable**s defined inside the method, except that they are given initial values before the method body is **execute**d. For example, the single parameter to the main method is a variable which is given a **list** of strings before the method begins execution, these strings being the command line arguments supplied to the program.

The names of the parameters are not important to Java – as long as they all have different names! The names only mean something to the human reader, which is of course important. The above method could easily have been written as follows.

```
private static void printHeightPerYear(double howTall, int howOld)
{
  System.out.println("At age " + howOld + ", height per year ratio is "
                     + howTall / howOld);
} // printHeightPerYear
```

You might think the first version is subjectively nicer than the second, but clearly both are better than this next one!

```
private static void printHeightPerYear(double d, int i)
```

```
{
  System.out.println("At age " + i + ", height per year ratio is "
                     + d / i);
} // printHeightPerYear
```

And that is only marginally better than calling the parameters, say `x` and `y`. However, Java does not care – it is not clever enough to be able to, as it can have no understanding of the problem being solved by the code.

5.10 Method: accepting parameters: of a class type (page 164)

The **method parameter**s of a **method** can be of any **type**, including **class**es. A parameter which is of a class type must be given a **method argument** value of that type when the method is invoked, for example a **reference** to an **object** which is an **instance** of the class named as the parameter type.

5.11 Method: calling a method (page 119)

The body of a **method** is **execute**d when some other code refers to it using a **method call**. For example, the program calls a method named `println` when it executes `System.out.println("Hello world!")`. For another example, if we have a method, named `printHeightPerYear`, which prints out a height to age ratio when it is given a height (in metres) and an age, then we could make it print the ratio between the height `1.6` and the age `14` using the following method call.

```
printHeightPerYear(1.6, 14);
```

When we call a method we supply a **method argument** for each **method parameter**, separating them by commas (`,`). These argument values are copied into the corresponding parameters of the method – the first argument goes into the first parameter, the second into the second, and so on.

The arguments passed to a method may be the current values of **variable**s. For example, the above code could have been written as follows.

```
double personHeight = 1.6;
int personAge = 14;

printHeightPerYear(personHeight, personAge);
```

As you may expect, the arguments to a method are actually **expression**s rather than just **literal value**s or variables. These expressions are **evaluate**d at the time the method is called. So we might have the following.

```
double growthLastYear = 0.02;

printHeightPerYear(personHeight - growthLastYear, personAge - 1);
```

5.12 Method: void methods (page 120)

Often, a **method** might calculate some kind of **function** or formula, perhaps based on its **method parameter**s, and **return** the answer as a result. The result might be an **`int`** or a **`double`** or some other **type**. If a method returns a result then the **return type** of the result must be stated in its heading. If it does not, then we write the word **`void`** instead, which literally means (among other definitions) 'without contents'. For example, the **main method** of a program does not return a result – it is always a **void method**.

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

5.13 Method: returning a value (page 122)

A **method** may **return** a result back to the code that called it. If this is so, we declare the **return type** of the result in the method heading, in place of the **reserved word `void`**. Such methods are often called **non-void method**s. For example, the following method takes a Celsius temperature, and returns the corresponding Fahrenheit value.

```
private static double celsiusToFahrenheit(double celsiusValue)
{
  double fahrenheitValue = celsiusValue * 9 / 5 + 32;
  return fahrenheitValue;
} // celsiusToFahrenheit
```

The method is declared with a return type of **`double`**, by writing that **type** name before the method name.

The **return statement** is how we specify what value is to be returned as the result of the method. The **statement** causes the execution of the method to end, and control to transfer back to the code that called the method.

The result of a non-void method can be used in an **expression**. For example, the method above might be used as follows.

```
double celsiusValue = Double.parseDouble(args[0]);
System.out.println("The Fahrenheit value of "
                   + celsiusValue + " Celsius is "
                   + celsiusToFahrenheit(celsiusValue) + ".");
```

The return statement takes any expression after the reserved word **`return`**. So our method above could be implemented using just one statement.

```
private static double celsiusToFahrenheit(double celsiusValue)
{
  return celsiusValue * 9 / 5 + 32;
} // celsiusToFahrenheit
```

5.14 Method: returning a value: of a class type (page 176)

A **method** may **return** a result back to the code that called it, and this may be of any **type**, including a **class**. In such cases, the value returned will typically be a **reference** to an **object** which is an **instance** of the class named as the **return type**.

For example, in a `Point` class with **instance variable**s `x` and `y`, we might have an **instance method** to return a `Point` which is half way along a straight line between this `Point` and a given other `Point`.

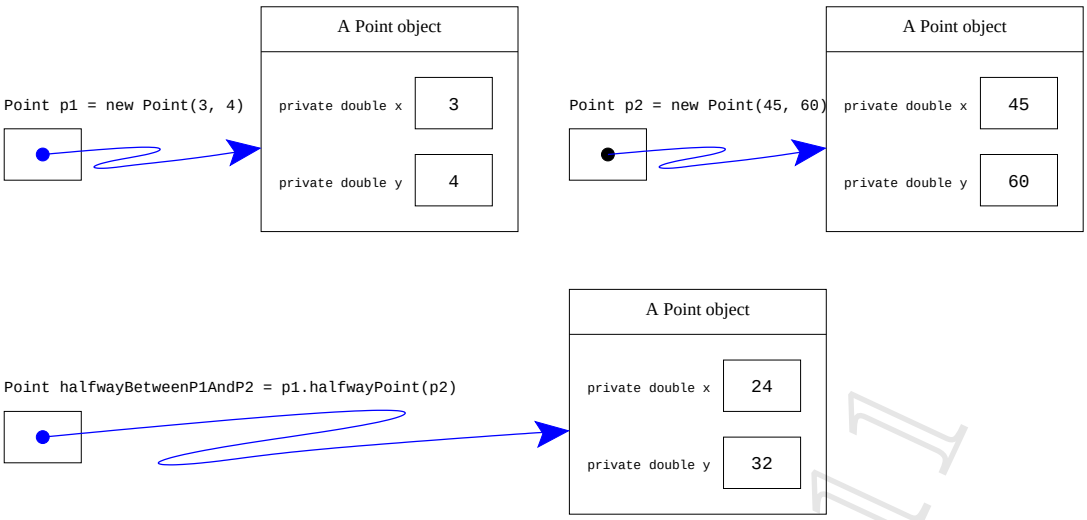
```
public Point halfWayPoint(Point other)
{
  double newX = (x + other.x) / 2;
  double newY = (y + other.y) / 2;
  return new Point(newX, newY);
} // halfWayPoint
```

The method creates a **new object** and then returns a reference to it. This might be used as follows.

```
Point p1 = new Point(3, 4);
Point p2 = new Point(45, 60);

Point halfWayBetweenP1AndP2 = p1.halfWayPoint(p2);
```

The reference to the new `Point` returned by the instance method, is stored in the **variable** `halfWayBetweenP1AndP2`. It would, of course, be the point $(24, 32)$. This is illustrated in the following diagram.



5.15 Method: returning a value: multiple returns (page 196)

The **return statement** is how we specify what value is to be **return**ed as the result of a **non-void method**. The **statement** causes the execution to end, and control to transfer back to the code that called the **method**. Typically, this is written as the last statement in the method, but we can actually write one or more anywhere in the method.

The Java **compiler** checks to make sure that we have been sensible, and that:

- There is no path through the method that does not end with a return statement.
- There is no code in the method that can never be reached due to an earlier occurring return statement.

5.16 Method: changing parameters does not affect arguments (page 124)

We can think of **method parameter**s as being like **variable**s defined inside the **method**, but which are given their initial value by the code that calls the method. This means the method can change the values of the parameters, like it can for any other variable defined in it. Such changes have no effect on the environment of the code that called the method, regardless of where the **method argument** values came from. An argument value, be it a literal constant, taken straight from a variable, or the result of some more complex **expression**, is simply copied into the corresponding parameter at the time the method is called. This is known as **call by value**.

5.17 Method: changing parameters does not affect arguments: but referenced objects can be changed (page 208)

All **method parameter**s obtain their values from the corresponding **method argument** using the **call by value** principle. This means a **method** cannot have any effect on the calling environment via its method parameters if they are of a **primitive type**.

However, if a method parameter is of a **reference type** then there is nothing to stop the code in the method following the **reference** supplied as the argument, and altering the state of the **object** it refers to (if it is a **mutable object**). Indeed, such behaviour is often exactly what we want.

In the abstract example below, assume that `changeState()` is an **instance method** in the **class** `SomeClass` which alters the values of some of the **instance variable**s.

```
public static void changeSomething(SomeClass object, SomeType value)
{
  object.changeState(value); // This really changes the object referred to.
  object = null;             // This has no effect outside of this method.
  ...
} // changeSomething
  ...
  SomeClass variable = new SomeClass();
  changeSomething(variable, someValueOfSomeType);
```

At the end of the above code, the change caused by the first line of the method has had an impact outside of the method, whereas the second line has had no such effect.

5.18 Method: constructor methods (page 159)

A **class** which is to be used as a template for making **object**s should be given a **constructor method**. This is a special kind of **method** which contains instructions for the **construct**ion of objects that are **instance**s of the class. A constructor method always has the same name as the class it is defined in. It is usually declared as being **public**, but we do not specify a **return type** or write the **reserved word** `void`. Constructor methods can have **method parameter**s, and typically these are the initial values for some or all of the **instance variable**s.

For example, the following might be a constructor method for a `Point` class, which has two instance variables, `x` and `y`.

```
public Point(double requiredX, double requiredY)
{
  x = requiredX;
```

```
    y = requiredY;
  } // Point
```

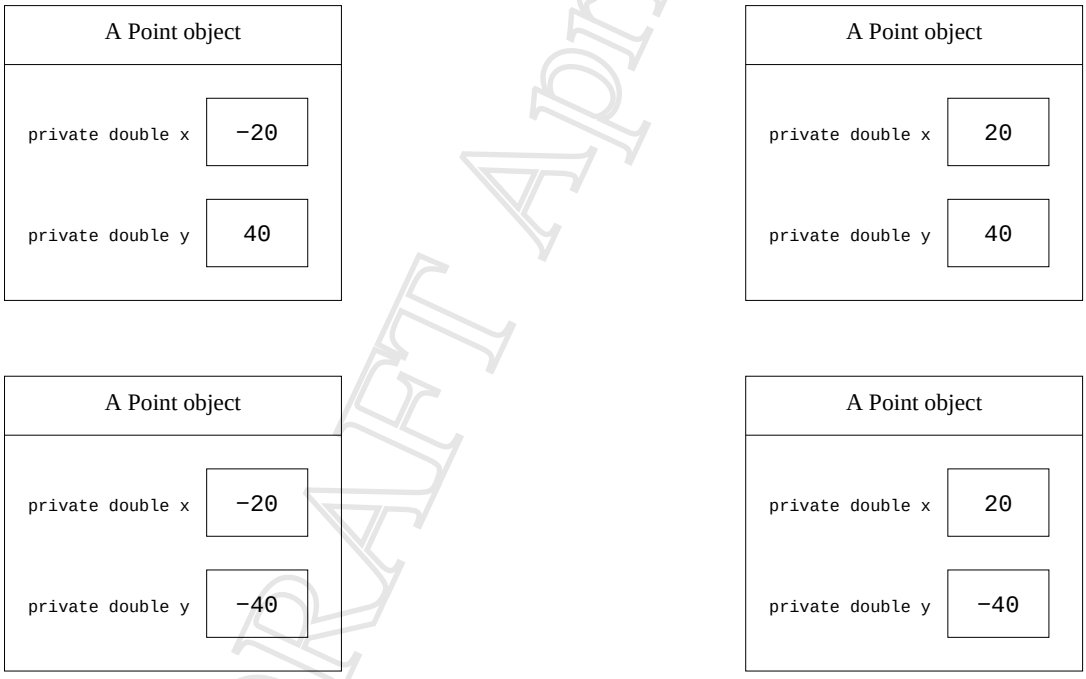
This says that in order to construct an object which is an instance of the class `Point`, we need to supply two **`double`** values, the first will be placed in the `x` instance variable, and the second in the `y` instance variable. Constructor methods are called in a similar way to any other **method**, except that we precede the **method call** with the **reserved word `new`**. For example, the following code would create a **new** object, which is an instance of the class `Point`, and in which the instance variables `x` and `y` have the values `7.4` and `-19.9` respectively.

```
new Point(7.4, -19.9);
```

We can create as many `Point` objects as we wish, each of them having their own pair of instance variables, and so having possibly different values for `x` and `y`. These next four `Point` objects are the coordinates of a rectangle which is centred around the origin of a graph, point (0, 0).

```
new Point(-20, 40);
new Point(-20, -40);
new Point(20, 40);
new Point(20, -40);
```

This is illustrated in the following diagram.



All four `Point` objects each have two instance variables, called `x` and `y`.

5.19 Method: constructor methods: more than one (page 203)

A **class** can have more than one **constructor method** as long as the number, order and/or **type**s of the **method parameter**s are different. This distinction is necessary so that the **compiler** can tell which constructor should be used when an **object** is being created.

5.20 Method: class versus instance methods (page 166)

When we define a **method**, we can write the **reserved word** `static` in its heading, meaning that it can be **execute**d in the **static context**, that is, it can be used as soon as the **class** is loaded into the **virtual machine**. These are known as **class method**s, because they belong to the class. By contrast, if we omit the `static` **modifier** then the method is an **instance method**. This means it can only be run in a **dynamic context**, attached to a particular **instance** of the class.

This parallels the distinction between **class variable**s and **instance variable**s. There is one copy of a class variable, created when the class is loaded. There is one copy of an instance variable for every instance, created when the instance is created.

We can think of methods in the same way: class methods belong to the class they are defined in, and there is one copy of their code at **run time**, ready for use immediately. Instance methods belong to an instance, and there are as many copies of the code at run time as there are instances. Of course, the virtual machine does not really make copies of the code of instance methods, but it *behaves* as though it does, in the sense that when an instance method is executed, it runs in the context of the instance that it belongs to.

For example, suppose we have a `Point` class with instance variables `x` and `y`. We might wish to have an instance method which takes no **method parameter**s, but **return**s the distance of a point from the origin. Pythagoras[18] tells us that this is $\sqrt{x^2+y^2}$. (We can use the `sqrt()` method from the `Math` class.)

```
public double distanceFromOrigin()
{
  return Math.sqrt(x * x + y * y);
} // distanceFromOrigin
```

A class method can be accessed by taking the name of the class, and appending a dot (`.`) and then the name of the method. `Math.sqrt` is a handy example right now.

An instance method belonging to an **object** can be accessed by taking a **reference** to the *object* and appending a dot (`.`) and then the name of the method. For example, if the **variable** `p1` contains a reference to a `Point` object, then the code `p1.distanceFromOrigin()` invokes the instance method `distanceFromOrigin()`, belonging to the `Point` referred to by `p1`.

The following code would print the numbers 5 and 75.

```
Point p1 = new Point(3, 4);
Point p2 = new Point(45, 60);

System.out.println(p1.distanceFromOrigin());
System.out.println(p2.distanceFromOrigin());
```

When the method is called via p1 it uses the instance variables of the object referred to by p1, that is the values 3 and 4 respectively. When the method is called via p2 it uses the values 45 and 60 instead.

For another example, we may wish to have a method which determines the distance between a point and a given other point.

```
public double distanceFromPoint(Point other)
{
  double xDistance = x - other.x;
  double yDistance = y - other.y;

  return Math.sqrt(xDistance * xDistance + yDistance * yDistance);
} // distanceFromPoint
```

The following code would print the number 70.0, twice.

```
System.out.println(p1.distanceFromPoint(p2));
System.out.println(p2.distanceFromPoint(p1));
```

5.21 Method: a method may have no parameters (page 173)

The list of **method parameter**s given to a **method** may be empty. This is typical for methods which always have the same effect or **return** the same result, or their result depends on the value of **instance variable**s rather than some values in the context where the method is called.

5.22 Method: return with no value (page 206)

A **void method** may contain **return statement**s which do not have an associated **return** value – just the **reserved word** `return`. These cause the execution of the **method** to end, and control to transfer back to the code that called the method. Every void method behaves as though it has an implicit return statement at the end, unless it has one explicitly written.

The use of return statements throughout the body of a method permits us to design them using a **single entry, multiple exit** principle: every call of the method starts at the beginning, but depending on **condition**s the execution may exit at various points.

5.23 Method: accessor methods (page 207)

A **public instance method** whose job it is to reveal all or some part of the **object state**, without changing it, is known as an **accessor method**. Perhaps the most obvious example of this is an instance method called `getSomeVariable`, where `someVariable` is the name of an **instance variable**. However, a well **design**ed **class** with good **encapsulation** does not systematically reveal to its user what its instance variables are. Hence the more general idea of an accessor method: it exposes the value of some *feature*, which might or might not be directly implemented as an instance variable.

5.24 Method: mutator methods (page 207)

A **public instance method** whose job it is to set or update all or some part of the **object state** is known as a **mutator method**. Perhaps the most obvious example of this is an instance method called `setSomeVariable`, where `someVariable` is the name of an **instance variable**. However, the more general idea of a mutator method is that it changes the value of some feature, which might or might not be directly implemented as an instance variable.

Obviously, only **mutable object**s have mutator methods.

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 (page 135)

The **type** of **text data string**s, such as **string literal** values and **concatenation**s of such, is called `String` in Java.

7.3 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"
```

7.4 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.

7.5 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.

7.6 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**.

7.7 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);
```

7.8 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"`.

7.9 Type: String: conversion: from object (page 177)

It is quite common for **class**es to have an **instance method** which is **design**ed to produce a `String` representation of an **object**. It is conventional in Java for such **method**s to be called `toString`. For example, a `Point` class with `x` and `y` **instance variable**s might have the following `toString()` method.

```
public String toString()
{
  return "(" + x + "," + y + ")";
} // toString
```

For convenience, whenever the Java **compiler** finds an **object reference** as an **operand** of the **concatenation operator** it assumes that the object's `toString()` method is to be invoked to produce the required `String`. For example, consider the following code.

```
Point p1 = new Point(10, 40);
System.out.println("The point is " + p1.toString());
```

Thanks to the compiler's convenient implicit assumption about `toString()`, the above code could, and probably would, have been written as follows.

```
Point p1 = new Point(10, 40);
System.out.println("The point is " + p1);
```

For our further convenience, there is a separate version of `System.out.println()` that takes any single object rather than a string, and prints its `toString()`. Thus, the code

```
System.out.println(p1);
```

has the same effect as the following.

```
System.out.println("" + p1);
```

7.10 Type: String: conversion: from object: null reference (page 211)

For convenience, whenever the Java **compiler** finds an **object reference** as an **operand** of the **concatenation operator** it assumes that the object's `toString()` **instance method** is to be invoked to produce the required `String`. However, the reference might be the **null reference** in which case there is no object on which to invoke `toString()`, so instead, the string `"null"` is used.

In fact, assuming `someString` is some `String` and `myVar` is a **variable** of a **reference type**, then the code:

```
someString + myVar
```

is actually treated as follows.

```
someString + (myVar == null
              ? "null"
              : (myVar.toString() == null ? "null" : myVar.toString()))
```

The same applies to the first operand of string concatenation if that is an object reference.

For this reason, most Java programmers prefer to use `"" + myVar` rather than `myVar.toString()` when they wish to convert the object referenced by `myVar` to a string, because it avoids the possibility of an **exception** if `myVar` contains the null reference.

7.11 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..

7.12 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.

7.13 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**.

7.14 Type: boolean (page 133)

There is a **type** in Java called **`boolean`**, and this is the type of all **condition**s used in **if else statement**s and **loop**s. It is named after the English mathematician, George Boole whose work

in 1847 established the basis of modern logic[12]. The type contains just two **boolean literal** values called **`true`** and **`false`**. For example, `5 <= 5` is a **boolean expression**, which, because it has no **variable**s in it, always has the same value when **evaluate**d. Whereas the **expression** `age1 < age2 || age1 == age2 && height1 <= height2` has a value which depends on the values of the variables in it.

7.15 Type: long (page 145)

The **type `int`** allows for the storage of **integer**s in the range -2^{31} through to $2^{31}-1$. This is because it uses four **byte**s, i.e. 32 **binary digit**s. $2^{31}-1$ is 2147483647. Although this is plenty for most purposes, we sometimes need whole numbers in a bigger range. The type **`long`** represents **long integer**s and uses eight bytes, i.e. 64 **bit**s. A **`long` variable** can store numbers from -2^{63} through to $2^{63}-1$. The value of $2^{63}-1$ is 9223372036854775807.

A **long literal** is written with an `L` on the end, to distinguish it from an **int literal**, as in `-15L` and `2147483648L`.

7.16 Type: short (page 145)

The **type `short`** represents **short integer**s using two **byte**s, i.e. 16 **binary digit**s. A **short variable** can store numbers from -2^{15} through to $2^{15}-1$. The value of $2^{15}-1$ is 32767. We would typically use this type when we have a huge number of **integer**s, which happen to lie in the restricted range, and we are concerned about the amount of memory (or **file** space) needed to store them.

7.17 Type: byte (page 145)

The **type `byte`** represents **integer**s using just one **byte**, i.e. 8 **binary digit**s. A **`byte` variable** can store numbers from -2^{7} through to $2^{7}-1$. The value of $2^{7}-1$ is 127.

7.18 Type: char (page 145)

Characters in Java are represented by the **type `char`**. A **char variable** can store a single **character** at any time.

7.19 Type: char: literal (page 145)

A **character literal** can be written in our program by enclosing it in single quotes. For example `'J'` is a character literal.

7.20 Type: char: literal: escape sequences (page 146)

When writing a **character literal** we can use the same **escape sequence**s that are available within **string literal**s. These include the following.

```
char backspace = '\b';          char tab = '\t';
char newline = '\n';            char formFeed = '\f';
char carriageReturn = '\r';     char doubleQuote = '\"';
char singleQuote = '\'';        char backslash = '\\';
```

7.21 Type: float (page 146)

The **type** `float` is for **real** (fractional decimal) numbers, using the **floating point representation** with a **single precision** storage. It uses only four **byte**s per number, compared with `double` which employs **double precision** storage and so is far more accurate, but needs eight bytes per number.

A **float literal** is written with an `f` or `F` on the end, as in `0.0F`, `-129.934F` or `98.2375f`.

7.22 Type: primitive versus reference (page 162)

Each **type** in Java is either a **primitive type** or a **reference type**. Values of primitive types have a size which is known at **compile time**. For example, every `int` value comprises four **byte**s. Types for which the size of an individual value is only known at **run time**, such as **class**es, are known as reference types because the values are always accessed via a **reference**.

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");
```

`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: System: out.println(): with no argument (page 98)

The **class** `System` also contains a version of the `out.println()` **method** which takes no arguments. This outputs nothing except a **new line**. It has the same effect as calling `System.out.println()` with an empty string as its argument, that is

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

has the same effect as the following.

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

So, for example

```
System.out.print("Hello world!");
System.out.println();
```

would have the same effect as the following.

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

`System.out.println()` with no argument is most useful when we need to end a line which has been generated a piece at a time, or when we want to have a blank line.

8.3 Standard API: System: out.print() (page 98)

The **class** `System` contains a **method** `out.print()` which is almost the same as `out.println()`. The only difference is that `out.print()` does not produce a **new line** after printing its output. This means that any output printed after this will appear on the same line. For example

```
System.out.print("Hello");
System.out.print(" ");
System.out.println("world!");
```

would have the same effect as the following.

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

`System.out.print()` is most useful when the output is being generated a piece at a time, often within a **loop**.

8.4 Standard API: System: out.printf() (page 126)

The **class** `System` contains a **method** `out.printf()`, introduced in Java 5.0, which is similar to `out.print()` except that we can use it to produce formatted output of values.

A simple use of this is to take an **integer** value and have it printed with **space padding** to a given positive integer field width. This means the output contains leading spaces followed by the usual representation of the integer, such that the number of **character**s printed is at least the given field width.

The following code fragment includes an example which prints a string representation of `123`, with leading spaces so that the result has a width of ten characters.

```
System.out.println("1234567890");
System.out.printf("%10d%n", 123);
```

Here is the effect of these two **statement**s.

```
1234567890
       123
```

The first `%` tells `out.printf()` that we wish it to format something, the `10` tells it the minimum total width to produce, and the following letter says what kind of conversion to perform. A `d` tells it to produce the representation of a decimal whole number, which is given after the **format specifier** string, as the second **method argument**. The `%n` tells `out.printf()` to output the platform dependent **line separator**.

The method can be asked to format a floating point value, such as a **`double`**. In such cases we give the minimum total width, a dot (`.`), the number of decimal places, and an `f` conversion. For example,

```
System.out.printf("%1.2f%n", 123.456);
```

needs more than the given minimum width of 1, and so produces the following.

```
123.46
```

Whereas, the format specifier in

```
System.out.println("1234567890");
System.out.printf("%10.2f%n", 123.456);
```

prints a total of ten characters for the number, two of which are decimal places.

```
1234567890
    123.46
```

8.5 Standard API: System: out.printf(): zero padding (page 140)

We can ask
`System.out.printf()` for **zero padding** rather than **space padding** of a number by placing a leading zero on the desired minimum width in the **format specifier**.

The following code fragment contains an example which prints a string representation of `123`, with leading zeroes so that the result is ten **character**s long.

```
System.out.println("1234567890");
System.out.printf("%010d%n", 123);
```

Here is the effect.

```
1234567890
0000000123
```

Similarly,

```
System.out.println("1234567890");
System.out.printf("%010.2f%n", 123.456);
```

produces the following.

```
1234567890
0000123.46
```

8.6 Standard API: System: in (page 187)

Inside the `System` **class**, in addition to the **class variable** called `out`, there is another called `in`. This contains a **reference** to an **object** which represents the **standard input** of the program.

Perhaps surprisingly, unlike the **standard output**, the standard input in Java is not easy to use as it is, and we typically access it via some other means, such as a `Scanner`.

8.7 Standard API: System: getProperty() (page 195)

When a program is **run**ning, various **system property** values hold information about such things as the Java version and platform being used, the home directory of the user, etc.. The **class method** `System.getProperty()` takes the name of such a property as its `String` **method parameter** and **return**s the corresponding `String` value.

8.8 Standard API: System: getProperty(): line.separator (page 195)

`System.getProperty()` maps the name `line.separator` onto the **system property** which is the **line separator** for the platform in use.

8.9 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.

8.10 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.

8.11 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`.

8.12 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)`.

8.13 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;
```

8.14 Standard API: Math: random() (page 205)

The standard **class** `java.lang.Math` contains a **class method** called `random`. This takes no **method argument**s and **return**s some **`double`** value, r, such that $0.0 \leq r < 1.0$ is true. The value is chosen in a pseudo random fashion, using an **algorithm** which exhibits the characteristics of an approximately uniform distribution of random numbers.

8.15 Standard API: Scanner (page 188)

Since the advent of Java 5.0 there is a standard **class** called `java.util.Scanner` which provides some simple features to read input **data**. In particular, it can be used to read `System.in` by passing that to its **constructor method** as follows.

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

Each time we want a line of text we invoke the `nextLine()` **instance method**.

```
  String line = inputScanner.nextLine();
  ...
```

Or maybe we want to read an **integer** using `nextInt()`.

```
int aNumber = inputScanner.nextInt();
// Skip past anything on the same line following the number.
inputScanner.nextLine();
...
```

Essentially, `System.in` accesses the **standard input** as a stream of **byte**s of data. A `Scanner` turns these bytes into a stream of **character**s (i.e. **`char`** values) and offers a variety of instance methods to scan these into whole lines, or various tokens separated by **white space**, such as spaces, tabs and end of lines. Some of these instance methods are listed below.

Public method interfaces for class `Scanner` (some of them).

Method	Return	Arguments	Description
`nextLine`	`String`		Returns all the text from the current point in the character stream up to the next end of line, as a `String`.
`nextInt`	**`int`**		Skips any spaces, tabs and end of lines and then reads characters which represent an integer, and **return**s that value as an **`int`**. It does not skip spaces, tabs or end of lines following those characters. The characters must represent an integer, or a **run time error** will occur.
`nextBoolean`	**`boolean`**		Similar to `nextInt()` except for a **`boolean`** value.
`nextByte`	**`byte`**		Similar to `nextInt()` except for a **`byte`** value.
`nextDouble`	**`double`**		Similar to `nextInt()` except for a **`double`** value.
`nextFloat`	**`float`**		Similar to `nextInt()` except for a **`float`** value.
`nextLong`	**`long`**		Similar to `nextInt()` except for a **`long`** value.
`nextShort`	**`short`**		Similar to `nextInt()` except for a **`short`** value.

There are very many more features in this class, including the ability to change what is considered to be characters that separate the various tokens.

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.");
```

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.

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.

9.14 Statement: for loop: multiple statements in for update (page 136)

Java **for loop**s are permitted to have more than one **statement** in their **for update**, that is, the part which is **execute**d after the **loop body**. Rather than always being one statement, this part may be a list of statements with commas (`,`) between them.

One appropriate use for this feature is to have a for loop that executes twice, once each for the two possible values of a **boolean variable**.

For example, the following code prints out scenarios to help train people to live in the city of Manchester!

```
boolean isRaining = true;
boolean haveUmbrella = true;
for (int countU = 1; countU <= 2; countU++, haveUmbrella = !haveUmbrella)
  for (int countR = 1; countR <= 2; countR++, isRaining = !isRaining)
  {
    System.out.println("It is" + (isRaining ? "" : " not") + " raining.");
    System.out.println
       ("You have " + (haveUmbrella ? "an" : "no") + " umbrella.");
    if (isRaining && !haveUmbrella)
      System.out.println("You get wet!");
    else
      System.out.println("You stay dry.");
    System.out.println();
  } // for
```

9.15 Statement: statements can be nested within each other (page 92)

Statements that control execution flow, such as **loop**s and **if else statement**s have other **statement**s inside them. These inner statements can be any kind of statement, including those that control the flow of execution. This allows quite complex **algorithm**s to be constructed with unlimited nesting of different and same kinds of control statements.

For example, one simple (but inefficient) way to print out the non-negative multiples of x which lie between y (≥ 0) and z inclusive, is as follows.

```
for (int number = 0; number <= z; number += x)
  if (number >= y)
    System.out.println("A multiple of " + x + " between " + y
                       + "and " + z + " is " + number);
```

9.16 Statement: switch statement with breaks (page 107)

Java provides a **conditional execution statement** which is ideal for situations where there are many choices based on some value, such as a number, being **equal** to specific fixed values for each choice. It is called the **switch statement**. The following example code will applaud the user when they have correctly guessed the winning number of `100`, encourage them when they are one out, or insult them otherwise.

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

switch (userGuess)
{
  case 99: case 101:
    System.out.println("You are close!");
    break;
  case 100:
    System.out.println("Bingo! You win!");
    System.out.println("You have guessed correctly.");
    break;
  default:
    System.out.println("You are pathetic!");
    System.out.println("Have another guess.");
    break;
} // switch
```

The switch statement starts with the **reserved word `switch`** followed by a bracketed **expression** of a **type** that has discrete values, such as **`int`** (notably not **`double`**). The body of the statement is enclosed in braces, ({ and }), and consists of a list of entries. Each of these starts with a list of labels, comprising the reserved word **`case`** followed by a value and then a colon (`:`). After the labels we have one or more statements, typically ending with a **break statement**. One (at most) label is allowed to be the reserved word **`default`** followed by a colon – usually written at the end of the list.

When a switch statement is **execute**d, the expression is **evaluate**d and then each label in the body is examined in turn to find one whose value is equal to that of the expression. If such a match is found, the statements associated with that label are executed, down to the special **break statement** which causes the execution of the switch statement to end. If a match is not found, then instead the statements associated with the **`default`** label are executed, or if there is no **`default`** then nothing is done.

9.17 Statement: switch statement without breaks (page 110)

A less common form of the **switch statement** is when we omit the **break statement**s at the end of the list of **statement**s associated with each set of **`case`** labels. This, perhaps surprisingly, causes execution to "fall through" to the statements associated with the next set of **`case`** labels. Most of the time we do *not* want this to happen – so we have to be careful to remember the break statements.

We can also mix the styles – having break statements for some entries, and not for some others. The following code is a bizarre, but interesting way of doing something reasonably simple. It serves as an illustration of the switch statement, and as a puzzle for you. It takes two **integer**s, the second of which is meant to be in the range one to ten, and outputs a result which is some **function** of the two numbers. What is that result?

```
int value = Integer.parseInt(args[0]);
int power = Integer.parseInt(args[1]);

int valueToThePower1 = value;
int valueToThePower2 = valueToThePower1 * valueToThePower1;
int valueToThePower4 = valueToThePower2 * valueToThePower2;
int valueToThePower8 = valueToThePower4 * valueToThePower4;

int result = 1;

switch (power)
{
  case 10: result *= valueToThePower1;
  case 9:  result *= valueToThePower1;
  case 8:  result *= valueToThePower8;
           break;
  case 7:  result *= valueToThePower1;
  case 6:  result *= valueToThePower1;
  case 5:  result *= valueToThePower1;
  case 4:  result *= valueToThePower4;
           break;
  case 3:  result *= valueToThePower1;
  case 2:  result *= valueToThePower2;
           break;
  case 1:  result *= valueToThePower1;
           break;
} // switch

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

If you find the semantics of the switch statement somewhat inelegant, then do not worry – you are not alone! Java inherited it from C, where it was designed more to ease the work of the

compiler than to be a good construct for the programmer. You will find the switch statement is less commonly used than the **if else statement**, and the majority of times you use it, you will want to have break statements on every set of **`case`** labels. Unfortunately, due to them being optional, accidentally missing them off does not cause a **compile time error**.

9.18 Statement: do while loop (page 112)

The **do while loop** is the third way in Java of having **repeated execution**. It is similar to the **while loop** but instead of having the **condition** at the start of the **loop**, it appears at the end. This means the condition is **evaluate**d *after* the **loop body** is **execute**d rather than before. The whole **statement** starts with the **reserved word `do`**. This is followed by the statement to be repeated, then the reserved word **`while`** and finally the condition, written in brackets.

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

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

Observe the semi-colon that is needed after the condition.

Of course, the body of the do while loop might be a **compound statement**, in which case we might lay out the code as follows.

```
int x = 0;
int y = 100;
do
{
  x++;
  y--;
} while (x != y);
```

The above is a long winded and inefficient way of giving both the variables `x` and `y` the value `50`.

Note that, because the condition is evaluated *after* the body is executed, the body is executed at least once. This is in contrast to the while loop, which might have have its body executed zero times.

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
```

12.6 Code clarity: comments: multi-line comments (page 189)

Another form of **comment** in Java allows us to have text which spans several lines. These start with the symbol `/*` and end with the symbol `*/`, which typically will be several lines later in the code. These symbols, and all text between them, is ignored by the **compiler**.

Less usefully, we can have the start and end symbols on the same line, with program code on either side of the comment, if we wish.

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.

13.3 Design: object oriented design (page 184)

When we are developing programs in an **object oriented programming** language, such as Java, we should use the principle of **object oriented design**. We start by identifying the **class**es we shall have in the program, by examining the **requirements statement** of the problem which the program is to solve. This is recognizing the idea that problems inherently involve interactions between 'real world' objects. These will be modelled in our program, by it creating **object**s which are **instance**s of the classes we identify.

In this view then, an object is an entity which has some kind of **object state** which might change over time, and some kind of **object behaviour** which might be based on its state.

From the requirements, we think carefully about the state and the behaviour of the objects in the problem. Then we decide how to model their behaviour using **instance method**s, and their state using **instance variable**s. There may, in general, be a need for **class variable**s and **class method**s too.

13.4 Design: object oriented design: noun identification (page 185)

One way to analyse the **requirements statement** in order to decide what **class**es to have in the program, is to simply go through the requirements and list all the nouns and noun phrases we can find. This is called **noun identification** and is useful because the objects inherent in the solution to most problems actually appear as nouns in the description of the problem. Some of the nouns will relate to **object**s that will exist at **run time**, and some will relate to classes in the program.

It is not the case that every noun found will be a class or an object, of course, and sometimes

we need classes that do not appear as nouns in the requirements. However, the technique is usually a good way of starting the process.

13.5 Design: object oriented design: encapsulation (page 187)

An important principle in **object oriented design** is the idea of **encapsulation**. A well designed **class** encapsulates the behaviour of the **object**s that can be created from it, in such a way that in order to use the class, one only needs to know about its **public method**s (including **constructor method**s) and what they mean, rather than how they work and what **instance variable**s the class may have. To help achieve good encapsulation, we follow the principle of **putting the logic where the data is** – all the code pertaining to the behaviour of particular objects are included in their class, rather than sprinkled around the various different classes of the program.

Encapsulation is an instance of **abstraction**. Abstraction is the process of ignoring detail which is not necessary for us to know (at the moment). We can use a class without having to know how it works, for example, if it is written by somebody else. Or, we can **design** the details of one class at a time for our programs, without at that moment being concerned with the details of how the other classes work.

For an example which has little to do with Java, assume you have just bought a cheap DVD TV recorder from your local supermarket. Do you need to know how it works in order to use it? Do you need to remove the case lid in order to use it? No, you only need to know about the buttons on the *outside* of the case. That is, until it breaks (after all it was a cheap one). Only at that point do you, or perhaps better still a TV gadget engineer, need to remove the case and poke around inside.

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`.

14.5 Variable: can be defined within a compound statement (page 92)

We can declare a **variable** within the body of a **method**, such as `main()`, (practically) anywhere where we can have a **statement**. The variable can then be used from that point onwards within the method body. The area of code in which a variable may be used is called its **scope**.

However, if we declare a variable within a **compound statement**, its scope is restricted to the compound statement: it does not exist after the end of the compound statement. This is a good thing, as it allows us to localize our variables to the exact point of their use, and so avoid cluttering up other parts of the code with variables available to be used but which have no relevance.

Consider the following symbolic example.

```
public static void main(String[] args)
{
  ...
  int x = ...
  ... x is available here.
  while (...)
  {
    ... x is available here.
    int y = ...
    ... x and y are available here.
  } // while
  ... x is available here, but not y,
  ... so we cannot accidentally refer to y instead of x.
} // main
```

The variable `x` can be used from the point of its definition onwards up to the end of the method, whereas the variable `y` can only be used from the point of its definition up to the end of the compound statement which is the body of the **loop**.

14.6 Variable: local variables (page 124)

When we declare **variable**s inside a **method**, they are local to that method and only exist while that method is running – they cannot be accessed by other methods. They are known as **local variable**s or **method variable**s. Also, different methods can have variables with the same name – they are different variables.

14.7 Variable: class variables (page 124)

We can declare **variable**s directly inside a **class**, outside of any **method**s. Such **class variable**s exist from the moment the class is loaded into the **virtual machine** until the end of the program, and they can be accessed by any method in the class. For example, the following are three class variables which might be used to store the components of today's date.

```
private static int presentDay;
private static int presentMonth;
private static int presentYear;
```

Notice that we use the **reserved word `static`** in their declaration. Also, class variables have a visibility **modifier** – the above have all been declared as being **private**, which means they can only be accessed by code inside the class which has declared them.

14.8 Variable: a group of variables can be declared together (page 129)

Java permits us to declare a group of **variable**s which have the same **type** in one declaration, by writing the type followed by a comma-separated list of the variable names. For example

```
int x, y;
```

declares two variables, both of type **`int`**. We can even assign values to the variables, as in the following.

```
int minimumVotingAge = 18, minimumArmyAge = 16;
```

This shorthand is not as useful as one might think, because of course, we typically have a **comment** before each variable explaining what its meaning is. However, we can sometimes have one comment which describes a group of variables.

14.9 Variable: boolean variable (page 133)

The **`boolean` type** can be used in much the same way as **`int`** and **`double`**, in the sense that we can have **boolean variable**s and **method**s can have **`boolean`** as their **return type**.

For example, consider the following code.

```
if (age1 < age2 || age1 == age2 && height1 <= height2)
  System.out.println("You are in the correct order.");
else
  System.out.println("Please swap over.");
```

We could, if we wished, write it using a **`boolean`** variable.

```
boolean correctOrder = age1 < age2 || age1 == age2 && height1 <= height2;
if (correctOrder)
  System.out.println("You are in the correct order.");
else
  System.out.println("Please swap over.");
```

Some people would argue that this makes for more readable code, as in effect, we have named the **condition** in a helpful way. How appropriate that is would depend on how obvious the code is otherwise, which is context dependent and ultimately subjective. Of course, the motive for storing the condition value in a **variable** is less subjective if we wish to use it more than once.

```
boolean correctOrder = age1 < age2 || age1 == age2 && height1 <= height2;
if (correctOrder)
  System.out.println("You are in the correct order.");
else
  System.out.println("Please swap over.");

... Lots of stuff here.

if (!correctOrder)
  System.out.println("Don't forget to swap over!");
```

Many novice programmers, and even some so-called experts, when writing the code above may have actually written the following.

```
boolean correctOrder;
if (age1 < age2 || age1 == age2 && height1 <= height2)
  correctOrder = true;
```

```
else
  correctOrder = false;

if (correctOrder == true)
  System.out.println("You are in the correct order.");
else
  System.out.println("Please swap over.");

... Lots of stuff here.

if (correctOrder == false)
  System.out.println("Don't forget to swap over!");
```

There are three *terrible* things wrong with this code (two of them are the same really) – identify them, *and do not write code like that*!

14.10 Variable: char variable (page 145)

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

```
char firstLetter = 'J';
```

declares a variable of type **`char`**, with the name `firstLetter`. At **run time**, this variable is allowed to hold a **`char` data** item, that is a single **character**.

14.11 Variable: instance variables (page 159)

The **variable**s that we wish to have inside **object**s are called **instance variable**s because they belong to the **instance**s of a **class**. We declare them in much the same way as we declare **class variable**s, except without the **reserved word `static`**. For example, the following code is part of the definition of a `Point` class with two instance variables to be used to store the components of a `Point` object.

```
public class Point
{
  private double x;
  private double y;
  ...
} // class Point
```

Like class variables, instance variables have a visibility **modifier** – the above variables have both been declared as being **private**, which means they can only be accessed by code inside the class which has declared them.

Class variables belong to the class in which they are declared, and they are created at **run time** in the **static context** when the class is loaded into the **virtual machine**. There is only one copy of each class variable. By contrast, instance variables are created dynamically, in a **dynamic context**, when the object they are part of is created during the **run** of the program. There are as many copies of each instance variable as there are instances of the class: each object has its own set of instance variables.

14.12 Variable: instance variables: should be private by default (page 175)

Java allows us to give **public** visibility to our **instance variable**s if we wish, but generally it is a good idea to define them as **private**. This permits us to alter the way we implement the **class**, without it affecting the code in other classes. For example, the programmer who has the job of maintaining a `Point` class with instance variables `x` and `y`, might decide it was better to re-implement the class to use instance variables that store the polar coordinate radius and angle instead. This might be because some new **method**s being added to the class would work much more easily in the polar coordinate system. Because the `x` and `y` instance variables had originally been made private, the programmer would know that there could not be any mention of them in other classes. So it would be safe to replace them with ones of a different name and which work differently. To make the points behave the same as before, the values given to the **constructor method** would be converted from *x* and *y* values to polar values, before being stored, and the `toString()` method could convert them back again.

14.13 Variable: of a class type (page 161)

As a **class** is a **type**, we can use one in much the same way as we use the built-in types, such as **`int`**, **`double`** and **`boolean`**. This means we can declare a **variable** whose type is a class. For example, if we have a class `Point` then we can have variables of type `Point`.

```
Point p1;
Point p2;
```

The above defines two **local variable**s or **method variable**s of type `Point`. We also can have **class variable**s and even **instance variable**s whose type is a class.

14.14 Variable: of a class type: stores a reference to an object (page 162)

There is one important difference between a **variable** whose **type** is a built-in **primitive type**, such as `int` and one whose type is a **class**. With the former, Java knows from the type how much memory will be needed for the variable. For example, a **double variable** needs more memory than an **int variable**, but all variables of type `int` need the same amount of memory, as do those of type `double`. Java needs this information so that it knows how to allocate memory addresses for variables.

By contrast, it is not possible to calculate how much memory will be needed to store an **object**, because **instance**s of different classes will have different sizes, and in some cases it is possible for different instances of the same class to have different sizes! The only time the size of an object is reliably known is when it is created, at **run time**.

To deal with this situation in a systematic way, variables which are of a class type do not store an object, but instead store a **reference** to an object. A reference to an object is essentially the memory address at which the object resides in memory, and is only known at run time when the object is created. Because they are really just memory addresses, the size of all references is the same, and is fixed. So by using references in variables of a class type, rather than actually storing objects, Java knows how much memory to allocate for any such variable.

Strictly speaking then, a type which is a class, is actually the **set** of possible *references* to instances of the class, rather than the set of actual instances themselves.

14.15 Variable: of a class type: stores a reference to an object: avoid misunderstanding (page 170)

Students new to the idea of **reference**s often fail to appreciate their significance, and make one or sometimes both of the following two mistakes.

1. Misconception: A **variable** is an **object**.
2. Misconception: A variable contains an object.

Neither of these are true, as we have already said: variables (of a **class type**) can contain a *reference* to an object. A common question is “why do we have to write `Date` twice in the following?”.

```
Date someBirthday
= new Date(birthDate.day, birthDate.month, birthDate.year + 1);
```

It is because we are doing three things.

1. We are declaring a variable.
2. We are **construct**ing an object.
3. We are storing a reference to that object in the variable.

So we can have a variable without an object.

```
Date someBirthday;
```

And we can have an object without a variable – could that be useful?

```
new Date(birthDate.day, birthDate.month, birthDate.year + 1);
```

Yes, it can be useful: for example, when we want to use objects just once, straight after constructing them.

```
System.out.println(new Point(3, 4).distanceFromPoint(new Point(45, 60)));
```

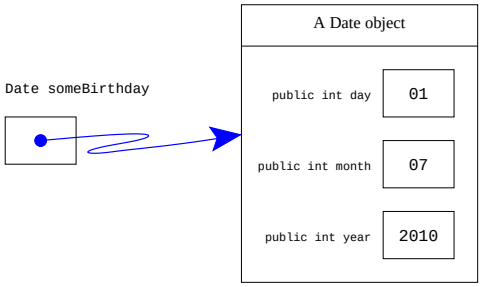
If we wish, we can have two variables referring to the same object.

```
Date theSameBirthday = someBirthday;
```

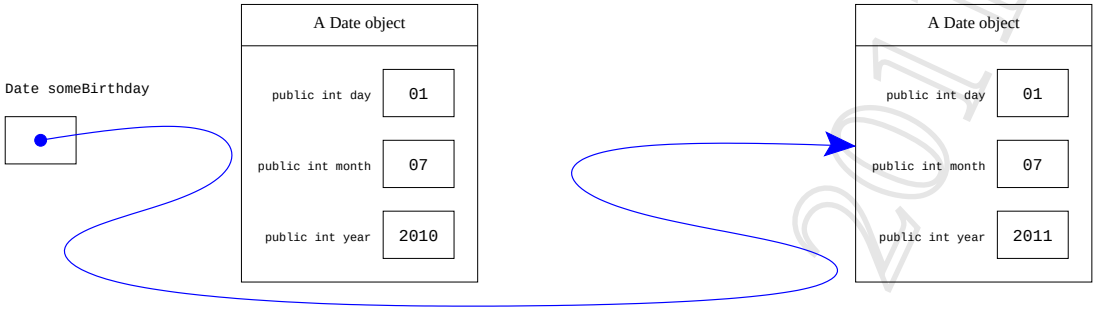
Also, we can change the value of a variable making it refer to a different object.

```
someBirthday = new Date(someBirthday.day, someBirthday.month,
                        someBirthday.year + 1);
```

This creates a **new** `Date` **object**, and stores the **reference** to it in `someBirthday` – overwriting the reference to the previous `Date` object. This is illustrated in the following diagram.




```
someBirthday = new Date(someBirthday.day, someBirthday.month, someBirthday.year + 1);
```



14.16 Variable: of a class type: null reference (page 192)

When an **object** is created, the **constructor method return**s a **reference** to it, which is then used for all accesses to the object. Typically, this reference is stored in a **variable**.

```
Point p1 = new Point(75, 150);
```

There is a special reference value, known as the **null reference**, which does not refer to an object. We can talk about it using the **reserved word** `null`. It is used, for example, as a value for a variable when we do not want it to refer to any object at this moment in time.

```
Point p2 = null;
```

So, in the example code here we have two `Point` variables, `p1` and `p2`, but (at **run time**) only one `Point` object.

Suppose the `Point` **class** has **instance method**s `getX()` and `getY()` with their obvious implementations. Then obtaining the *x* value of the object referenced by `p1` is fine; the following code would print `75`.

```
System.out.println(p1.getX());
```

However, the similar code involving `p2` would cause a **run time error** (an **exception** called `NullPointerException`).

```
System.out.println(p2.getX());
```

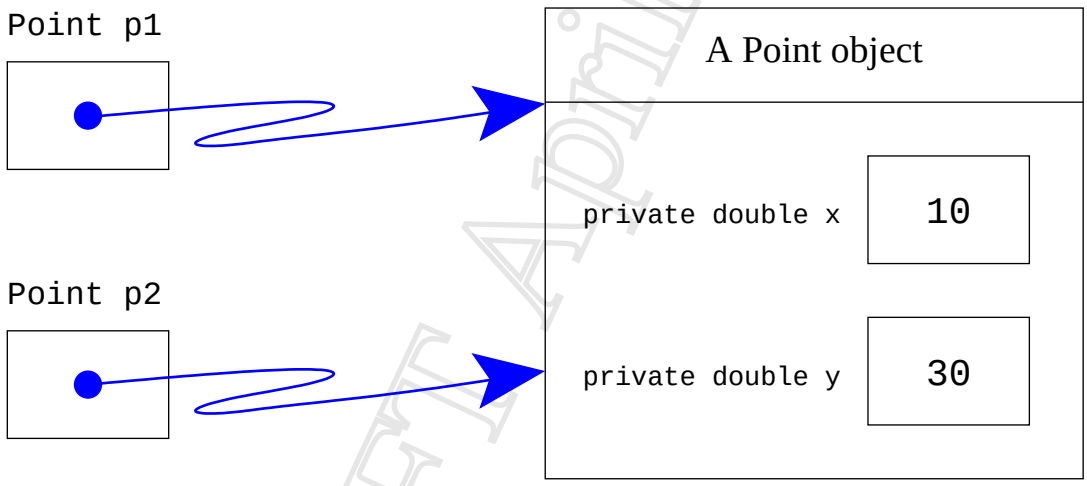
This is because there is no object referenced by `p2`, and so any attempt to access the referenced object must fail.

14.17 Variable: of a class type: holding the same reference as some other variable (page 216)

A **variable** which is of a **class type** can hold a **reference** to any **instance** of that class (plus the **null reference**). There is nothing to stop two (or more) variables having the same reference value. For example, the following code creates one `Point` **object** and has it referred to by two variables.

```
Point p1 = new Point(10, 30);

Point p2 = p1;
```



This reminds us that a variable is *not* itself an object, but merely a holder for a reference to an object.

Having two or more **variable**s refer to the same **object** can cause us no problems if it is an **immutable object** because we cannot change the object's state no matter which variable we use to access it. So, in effect, the object(s) referred to by the two variables behave the same as they would if they were two different objects. The following code has the same *effect* as the above fragment, almost no matter what we do with `p1` and `p2` subsequently.

```
Point p1 = new Point(10, 30);

Point p2 = new Point(10, 30);
```

The only behavioural difference between the two fragments is the **condition**s `p1 == p2` and `p1 != p2` which are **`true`** and **`false`** respectively for the first code fragment, and the other way round for the second one.

If, on the other hand, an **object referenced by more than one variable** is a **mutable object** we have to be careful because any change made via any one of the variables causes the change to occur in the (same) object referred to by the other variables. This may be, and often is, exactly what we want, or it may be a problem if our **design** is poor or if we have made a mistake in our code and the variables were not meant to share the object.

Consider the following simple example.

```
public class Employee
{
  private final String name;
  private int salary;

  public Employee(String requiredName, int initialSalary)
  {
    name = requiredName;
    salary = initialSalary;
  } // Employee

  public String getName()
  {
    return name;
  } // getName

  public void setSalary(int newSalary)
  {
    salary = newSalary;
  } // setSalary

  public int getSalary()
  {
    return salary;
  } // getSalary

} // class Employee

  ...

  Employee debora = new Employee("Debs", 50000);
```

```
Employee sharmane = new Employee("Shaz", 40000);

...

Employee worstEmployee = debora;
Employee bestEmployee = sharmane;

...
```

Now let us have an accidental piece of code.

```
worstEmployee = bestEmployee;
```

Then we carry on with intentional code.

```
...

bestEmployee.setSalary(55000);
worstEmployee.setSalary(0);

System.out.println("Our best employee, " + bestEmployee.getName()
                   + ", is paid " + bestEmployee.getSalary());
System.out.println("Our worst employee, " + worstEmployee.getName()
                   + ", is paid " + worstEmployee.getSalary());
```

The effect of the accidental sharing is to give Sharmane, who is our best employee, a pay increase to 55,000 immediately followed by a pay cut to zero because `worstEmployee` and `bestEmployee` are both referring to the same object, the one which is also referred to by `sharmane`. Meanwhile our worst employee, Debora, gets to keep her 50,000! Further more, the report only actually talks about Sharmane in both contexts!

```
Our best employee, Shaz, is paid 0
Our worst employee, Shaz, is paid 0
```

14.18 Variable: final variables (page 194)

When we declare a **variable** we can write the **reserved word `final`** as one of its **modifier**s before the **type** name. This means that once the variable has been given a value, that value cannot be altered.

If an **instance variable** is declared to be a **final variable** then it must be explicitly assigned a value by the time the **object** it belongs to has finished being **construct**ed. This would be done either by assigning a value in the **variable declaration**, or via an **assignment statement** inside the **constructor method**.

14.19 Variable: final variables: class constant (page 205)

A **class variable** which is declared to be a **final variable** (i.e. its **modifier**s include the **reserved word**s `static` and `final`) is also known in Java as a **class constant**. An example of this is the **variable** in the **class** `java.lang.Math` called `PI`.

```
public static final double PI = 3.14159265358979323846;
```

By convention, class constants are usually named using only capital letters with the words separated by underscores (_).

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: arithmetic: remainder operator (page 149)

Another **arithmetic operator** in Java is the **remainder operator**, also known as the **modulo** operator, %. When used with two **integer operand**s, it yields the remainder obtained from dividing the first operand by the second. As an example, the following **method** determines whether a given **`int` method parameter** is an even number.

```
public static boolean isEven(int number)
{
  return number % 2 == 0;
} // isEven
```

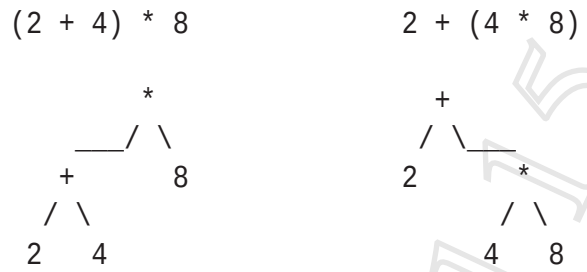
15.6 Expression: brackets and precedence (page 45)

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.



What if there were no brackets?

```
2 + 4 * 8
```

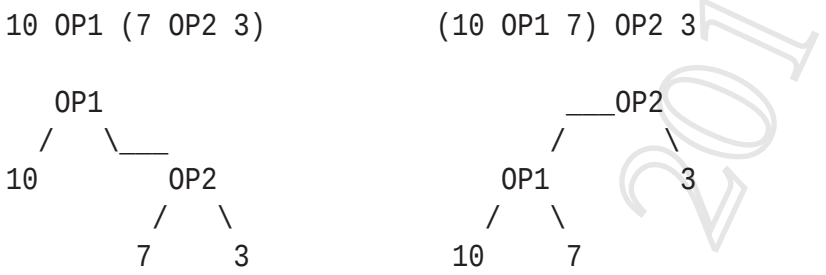
Java allows us to have expressions without any brackets, or more generally, without brackets around *every* sub-expression. It provides rules to define what the structure of such an expression is, i.e., where the missing brackets should go. If you look at the 4 in the above expression, you will see that 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.

15.7 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.



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.

15.8 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.

15.9 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.

15.10 Expression: boolean: logical operators (page 128)

For some **algorithm**s, we need **condition**s on **loop**s etc. that are more complex than can be made simply by using the **relational operator**s. Java provides us with **logical operator**s to enable us to glue together simple conditions into bigger ones. The three most commonly used logical operators are **conditional and**, **conditional or** and **logical not**.

Operator	Title	Posh title	Description
&&	and	**conjunction**	`c1 && c2` is **`true`** if and only if both conditions `c1` and `c2` **evaluate** to **`true`**. Both of the two conditions, known as **conjunct**s, must be **`true`** to satisfy the combined condition.
||	or	**disjunction**	`c1 || c2` is **`true`** if and only if at least one of the conditions `c1` and `c2` evaluate to **`true`**. The combined condition is satisfied, unless both of the two conditions, known as **disjunct**s, are **`false`**.
!	not	**negation**	`!c` is **`true`** if and only if the condition `c` evaluates to **`false`**. This operator negates the given condition.

We can define these **operator**s using **truth table**s, where ? means the **operand** is not evaluated.

c1	c2	c1 && c2
true	**true**	**true**
true	**false**	**false**
false	?	**false**

c1	c2	c1 || c2
true	?	**true**
false	**true**	**true**
false	**false**	**false**

c	!c
true	**false**
false	**true**

Using these operators, we can make up complex conditions, such as the following.

```
age1 < age2 || age1 == age2 && height1 <= height2
```

As with the **arithmetic operator**s, Java defines **operator precedence** and **operator associativity** to disambiguate complex conditions that are not fully bracketed, such as the one above. && and || have a lower precedence than the relational operators which have a lower precedence than the arithmetic ones. ! has a very high precedence (even more so than the arithmetic operators) and && has a higher precedence than ||. So the above example **expression** has implicit brackets as follows.

```
(age1 < age2) || ((age1 == age2) && (height1 <= height2))
```

This might be part of a program that **sort**s people standing in a line by age, but when they are the same age, it sorts them by height. Assuming that the **int variable**s `age1` and `height1` contain the age and height of one person, and the other two variables similarly contain that **data** for another, then the following code might be used to tell the pair to swap their order if necessary.

```
if (age1 < age2 || age1 == age2 && height1 <= height2)
  System.out.println("You are in the correct order.");
else
  System.out.println("Please swap over.");
```

We might have, perhaps less clearly, chosen to write that code as follows.

```
if (!(age1 < age2 || age1 == age2 && height1 <= height2))
  System.out.println("Please swap over.");
else
  System.out.println("You are in the correct order.");
```

You might find it tricky, but it's worth convincing yourself: yet another way of writing code with the same effect would be as follows.

```
if (age1 > age2 || age1 == age2 && height1 > height2)
  System.out.println("Please swap over.");
else
  System.out.println("You are in the correct order.");
```

In mathematics, we are used to writing expressions such as $x \leq y \leq z$ to mean true, if and only if y lies in the range x to z, inclusive. In Java, such expressions need to be written as `x <= y && y <= z`.

Also, in everyday language we are used to using the words 'and' and 'or' where they have very similar meanings to the associated Java operators. However, we say things like "my mother's age is 46 or 47". In Java, we would need to write `myMumAge == 46 || myMumAge == 47` to capture the same meaning. Another example, "my brothers are aged 10 and 12", might be coded as `myBrother1Age == 10 && myBrother2Age == 12`.

However, there are times in everyday language when we say "and" when we really mean "or" in logic, and hence would use `||` in Java. For example, "the two possible ages for my dad are 49 *and* 53" is really the same as saying "my dad's age is 49 *or* my dad's age is 53".

15.11 Expression: conditional expression (page 94)

The **conditional operator** in Java permits us to write **conditional expression**s which have different sub-expressions **evaluate**d depending on some **condition**. The general form is

```
c ? e1 : e2
```

where `c` is some condition, and `e1` and `e2` are two **expression**s of some **type**. The condition is evaluated, and if the value is **`true`** then `e1` is evaluated and its value becomes the result of the expression. If the condition is **`false`** then `e2` is evaluated and its value becomes the result instead.

For example

```
int maxXY = x > y ? x : y;
```

is another way of achieving the same effect as the following.

```
int maxXY;
if (x > y)
  maxXY = x;
else
  maxXY = y;
```

16 Package

16.1 Package (page 187)

There are hundreds of **class**es that come with Java in its **application program interface** (**API**), and even more that are available around the world for reusing in our programs if we wish. To help manage this huge number of classes, they are grouped into collections of related classes, called **package**s. But even this is not enough to make things manageable, so packages are grouped into a hierarchy in a rather similar way to how a well organized **file system** is arranged into directories and sub-directories. For example, there is one group of standard packages called `java` and another called `javax`.

16.2 Package: java.util (page 188)

One of the standard Java **package**s in the package group `java` is called `util`. This means its full name is `java.util` – the package addressing mechanism uses a dot (`.`) in much the same way as Unix uses a slash, or Microsoft Windows uses a backslash, to separate directories in a filename path. `java.util` contains many generally useful utility **class**es. For example, there is a class called `Scanner` which lives there, so its **fully qualified name** is `java.util.Scanner`. This fully qualified name is unique: if someone else was to create a class called `Scanner` then it would not be in the same package, so the two would not be confused.

We can refer to a class using its fully qualified name, for example the following declares a **variable** of **type** `java.util.Scanner` and creates an **instance** of the class too.

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