ID1018 Programming I
Lecture Notes

TCOMK

Fredrik Kilander

October 28, 2016

Contents

Introduction

Intended audience

These lecture notes are intended for the English-speaking students on the course ID1018 Programming I, at the KTH School of Information and Communication (ICT). Students who are adept at the Swedish language can, and should, primarily consult the Swedish course literature, but are of course welcome to this text as well.

Why you should read this book

These lecture notes give you the opportunity to prepare for the lectures. It highlights and broadens the topics presented there, and helps you form questions.

What this book is not

This book is not a substitute for the course literature. It is an added benefit, something which is intended to help you understand the lecture series, and why things are presented in a certain way.

This book is not a manual for the Java programming language. There are already too many such books. It does, however, contain summary information on the portions of the Java language as it is used in the ID1018 course, but for full information always consult the proper reference materials (the course literature or online documentation).

This book is not a magic text that will turn you into a computer programmer. You must do that yourself, with patience and practice. Programming is a creative art, and before one can begin to create one must know the basic tools and what is possible to do with them.

Further notes

Each chapter is intended to map against a lecture. That said, some chapters contain a lot more text than can be covered in a single lecture. This is mostly because some things cannot be glossed over simply.

Some chapters also contain reflections and digressions that may appear out of scope for a first programming course. View these as pointers towards the future.

Examples are not always consistent between chapters, usually because of the need to have less complicated examples.

The examples frequently use `...` to indicate that there *is* some code there, but the exact nature of said code is not critical.

Also, there may remain some undiscovered errors and bugs.

Chapter 1

The basic tools

This chapter presents the abstract computer, the Java programming language, and the basic tools needed to program in Java.

1.1 The abstract computer

Modern computers, like our laptops, desktops, pads, and mobile phones, are extremely complicated machines. They are fast, efficient, miniatyrized, and many draw little enough power to run on batteries for several hours. Despite the differences in appearence and form factor, they still contain several of the basic building blocks that have been around since when the general computing machines first arose, in the late 1940's.

These components are:

- primary memory
- secondary memory
- a central processing unit
- peripherals
- buses

1.1.1 Primary memory

The primary memory of the computer holds the data that the computer needs in order to actually do something. If all that needs to be done is to add

Binary	Decimal
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7

Table 1.1: Binary digits

two numbers together, those two numbers must be somewhere in the primary memory, together with an instruction to add them.

The primary memory is organized in *words*. Each word can hold a certain number of binary digits, or *bits*. It is technically simpler to build the electronic circuitry of the computer using elements that can only be in two different states (e.g. low or high, off or on, zero or one). Therefore, the smallest piece of data that the computer operates on is a single bit, and that bit is either 0 or 1.

When several bits are arranged together in a group and given an order of increasing significance from say, right to left, integer (whole) numbers can be represented by turning bits on or off. Table 1.1 illustrates this.

The number of bits in a memory word are decided by the architecture selected by the manufacturer of the computer. In the 1980's, when micro-computers and the first game consoles appeared, 8 bits in a word was the norm for home users. Soon there were 16 bit machines, while professional computing enjoyed word sizes like 32, 36, 48, 64, and even 128 bit machines for supercomputing performance.

Modern computers are usually 64 or 32 bit machines, which means that a single memory word has 64 or 32 bits in it, and can represent 2^{64} or 2^{32} different patterns. Wider words make for a faster computer, because if an 8-bit machine needs to fetch a 64 bit number, it must read from memory eight times. A 64-bit machine can fetch all those bits in one memory read.

Memory address

Each memory word has an *address*, so that a particular word can be accessed for reading or writing. Usually the address size is equal to the memory word size of the computer, but this is no law, it is a design decision. In addition, primary memory consists of electronic circuits arranged in modules, and each such module has a certain capacity, like 2^{30} bytes (1 Gbyte). This means that even if the computer hardware can address 64 Gbyte of memory, it may only have 8 Gbyte of physical memory installed.

Normally a programmer need not consider memory, unless the program is about to consume a lot of it, a lot being something like 20% or more of the physical memory installed. The operating system (like Linux, iOS, Windows) makes sure that the available memory is shared between all running programs, that it is used efficiently, and looks larger than it actually is with the help of secondary memory.

Other situations where memory *does* matter are programming for mobile devices that have no secondary memory, or for embedded systems where everything is small.

Memory persistence

In most cases the contents of the primary memory disappear when the computer is turned off. Power is required to hold the circuits active so that they can represent the state of their many many bits.

Persistent memory technologies, like an SD-card, generally do not work well as primary memory. They are too slow because it takes time – i.e. energy – to write something that sticks around when the power is gone.

1.1.2 Secondary memory

Secondary memory consists of harddiscs, SD memory cards, USB memory sticks, BlueRay, DVD, and CD discs, and in older times magnetic tape, punched paper cards or tape. The purpose of secondary memory is to hold vast amounts of data when it is not being used in primary memory, or when the computer is turned off. Secondary memory is generally slower to read or write, but the tradeoff is that it can hold a lot of data.

A computer harddisc, or disc drive, is a mechanical device that consist of one or more discs, or platters, joined on the same axis and covered with a magnetic material. A moving arm extends an array of reading and recording heads in between the platters, and makes it possible to write information on the platters in circular *tracks*. The number of tracks vary, and upwards of 80 tracks exists, depending on the size and precision of the mechanism.

Since a recording track is circular, it is divided into separate *sectors*, a sector being the smallest addressable unit of information on the harddisc. A typical sector size is 4 Kbyte (4096 bytes, 32796 bits). This means that if only just a single bit has to be updated, the whole sector must be read into primary memory, modified, and then written back out again, all 4 Kbyte of it.

In recent years the platters, arm, and read-write heads are often replaced with solid state circuitry, like flash memory. This makes for faster and quieter operations, and is not prone to mechanical breakdowns. These solid-state drives, or SSDs, are great, but they still cost more per byte of storage than the mechanical drives, and they have particular issues of their own, like speed and memory degradation after long use.

In a computer, a fast secondary memory like a harddisc provides storage for files, programs, and portions of the primary memory which the operating system temporarily has thrown out to create the illusion of a much larger primary memory (virtual memory).

The contents of a secondary memory is organized by a *file system*, which describes how files and directories are stored, how they may be named, and how they can be created and read. Many different file systems exist, and can even co-exist on the same computer, as long as the operating system is equipped to understand them.

A particular section of the computer's main disc drive is commonly set aside for the last of the power on and startup sequence, called *booting*. When turned on, the computer must go through a set of predefined steps in order to find and start an operating system. The *boot sector* on the main disc drive contains programs and information that directs the computer to the files that are needed in order to start properly.

1.1.3 Peripherals

In order to accept inputs and present outputs the computer needs to connect to peripheral devices. These devices are commonly a graphics card (which in turn connects to a screen), a keyboard for typing text and numbers, a pointing device like a mouse, fingerpad, fingerstick, or touch-screen, a sound card for making noises, fingerprint readers, cameras, memory cards and sticks, optical disc drives, and so on.

Several of the peripherals are often built into the computer case, while other devices attach to *ports*, like VGA, HDMI, DisplayPort (graphics), phono plugs (audio in and out), or USB (Universal Serial Bus) which can connect almost anything these days.

A group of very important peripherals are the network interfaces, wired or wireless. These connect the computer to other computers.

1.1.4 A central processing unit

The central processing unit, or CPU, is the engine of the computer, the fat spider at the centre of things. The CPU basically does two things: it reads instructions from words in the primary memory, and then executes the instructions, something which results in that data are manipulated or moved around between the CPU, primary memory, secondary memory, or the peripherals.

It is the CPU's ability to execute several billions of instructions per second that makes it appear powerful. Each instruction is actually not that remarkable. For example, fetch that, multiply with this, store the result over there. The sequence of instructions is designed to achieve the greater goal, for example to multiply all values in a table with a specified term.

A computer program then consists of many thousands of such CPU instructions, carefully arranged in appropriate sequences for the task. Many sequences can be reused, by using the current values in other parts of the primary memory. For example, one location could contain the address of the table to multiply, a second location the length of the table, and a third the value to multiply. Another part of the program can then load the correct values into those locations, and call the table multiplication sequence once for each table.

The CPU instructions, or *machine instructions* as they are also known, are patterns of bits that the CPU can decode and understand. Each model of a CPU has a specific set of instructions that it understands, and if it encounters an instruction that is not recognized this results in a very serious error.

The CPU itself consists of a *core* which performs logical and arithmetic operations, units that fetch and decode instructions, and a small amount of high-speed memory often referred to *registers*. In addition there are usually also several levels of *cache memory* between the registers and the primary memory. The cache memory speeds up the CPU by fetching several words at a time from the primary memory, and when the CPU wants the content of the next address (a very common situation) it is already inside the CPU, in the cache.

Modern CPUs often contain multiple cores and are thus capable of performing work in parallell. One core can execute instructions that belong to the operating system, while another computes a user program. The orchestration of this is done by the operating system, or by a combination of the operating system and the supporting circuitry.

It is important to realise that even with a single-core CPU, a multitasking operating system is capable of switching between tasks very fast, thereby creating the illusion of several programs running concurrently. In such environments it is common to have 50 or more programs (or *processes* as they called by the operating system) active in the computer at the same time. However, many of these processes are not actually running all the time. Most of them spend their time waiting for something to happen, like a timer expiring, or some network data to arrive.

1.1.5 Buses

A humble but critical part of any computer are the electrical pathways through which data is transferred between the components. A single wire (with a an

Decimal		Binary	
Kilo	$10^3 = 1,000$	Kbyte	$2^{10} = 1,024$
Mega	$10^6 = 1,000,000$	Mbyte	$2^{20} = 1,048,576$
Giga	$10^9 = 1,000,000,000$	Gbyte	$2^{30} = 1,073,741,824$
Tera	10^{12}	Tbyte	2^{40}
Peta	10^{15}	Pbyte	2^{50}
Exa	10^{18}	Exabyte	2^{60}

Table 1.2: Metric and binary prefixes

appropriate ground return) can carry a single signal, or one bit at a time. Using more wires increases the number of bits that can be simultaneously transported. A collection of such wires is commonly referred to as a *bus*, and the main bus in the computer is known as the *system bus*. The system bus is often 16 wires wide, but by using the wires twice (multiplexing) for each word to transfer, the devices connected to the bus see a 32-bit bus with half the actual speed. The reason for this is completely in the analogue domain; the physical wires consist of metal strips on the main circuitboard (the *motherboard*) and they must be placed carefully in the already crowded environment.

The CPU, primary and secondary memory, and the peripherals are all connected to the system bus. By setting values in the bus controller's registers, the CPU can dictate how data is to be transferred.

1.2 Metric prefixes in the binary system

There is often a need to talk about large quantities, for example 8 Gbyte of RAM, or a harddisc of 1 Tbyte. The metric prefixes, however, are based in the decimal system with base 10, while the digital quantities almost always are based on powers of 2. This means, that when someone says 1 Kbyte, they do not mean precisely 1000 byte, but rather 1024 byte, because $2^{10} = 1024$.

Table 1.2 shows the various metric prefixes and their binary counterparts.

1.3 The 8-bit byte

A group of 8 bits is commonly referred to as a *byte*. The byte is used a lot in computing: as a unit for defining the size of things (so and so many thousand or million bytes), as the smallest piece of memory that can be directly manipulated as a unit, and as the base for transfer rates (bytes per second). Since bits almost always come packaged in bytes, it is often more convenient to talk about bytes rather than bits.

The byte also has a special relationship with character encodings. Text is represented in the computer by giving each character a small number. `'A'` for example, has the number 64. Every uppercase letter of the alphabet has a number, and every lowercase letter a different number. In addition, there are numbers assigned to digits (`'0'`,`'1'`,`'2'`...) and punctuation marks, and spaces and newline and more.

In 1963 the American Standard Code for Information Interchange (ASCII) defined characters for each of the numbers 0–127. If two computer systems both used ASCII, then two different programs on different computers could exchange texts. Even better, those numbers could be transformed into images and shown on a screen, or printed on paper, and the text would come out as it had been written.

Now, 0–127 is 128 different values, so 128 characters are defined by ASCII. However, by no coincidence at all, that is also exactly the number of different values that 7 bits can encode: $2^7 = 128$. This means that each character will fit into an 8-bit byte, with one bit left over.

When ASCII was defined, before the age of digital telephone networks, computer communications commonly relied on audio-based modems and noisy telephone lines. That last bit in each character made it possible to detect one bit-error, using a scheme called *parity*. Hence, the 8th bit was also known as the parity bit.

The ASCII standard was great for interoperability, but was of course only for English texts. Languages with additional characters, could not be supported except by substituting some ASCII characters for the desired national ones, reintroducing some of the confusion that the ASCII standard tried to avoid.

As communication methods became more reliable and computer usage spread, it became increasingly jurassic to cling to the 8th parity bit in the character byte, and in the 1990s new standards were drafted for the Latin languages. These were called ISO-8859-1 up to ISO-8859-16 and the use of the 8th bit allowed for another 96 characters and a new set of control characters. A character code was now a number between 0 and 255, because with 8 bits you get $2^8 = 256$ different numbers.

Already when the ISO-8859 standard was being drafted it was realised that it would not suffice; the problem with choosing the correct character code table remained and there were still many glyphs from other languages of the world not covered by the standard. As a result, work proceeded towards *Unicode*, a standard in which each character is encoded by a 24-bit number. This makes it possible to give a number to almost any character from any language.

Unicode defines the mapping from numbers to printable glyphs. Other standards define how the character numbers are represented in computer memory or in a file. A currently popular encoding is UTF-8, which takes advantage of the fact that most Western language texts only require characters that can be represented with a single byte. When Unicode numbers higher than 255 are required, UTF-8 uses an escape sequence to indicate that more bytes must be read to acquire the complete number for the character.

The first 128 characters of Unicode are identical to ASCII, and the first 256 characters are identical to ISO-8859-1, thus making Unicode backwards compatible with both those standards.

1.4 Compilers and interpreters

The CPU only understands machine code, the set of instructions it was designed to execute. This means that a computer program must either be written in machine code, which is a slow, painful, and error-prone manual process, or it is written in some other language which is then automatically translated into machine code.

There are two ways of achieving such a translation; compilation or interpretation. Compilation means that the programmer writes the *source code* for the program, in whatever language is convenient and saves the text in a file. Another program, the *compiler*, then reads the source code and translates it into the corresponding machine instructions. When the translation completes without error, the resulting machine code output can be given to the operating system and scheduled for running.

Since the machine code is highly targeted towards the CPU, it follows that the compiler must know exactly how to output the correct machine instructions. Otherwise the program will not run. This also means, that if the programmer is sharing the source code with other people, they may have to recompile the program before they can run it on their computers.

The compiler reads the source code, line by line, and statement by statement. When it fully understands the meaning of a statement, it then puts together (compiles) the machine instructions that will achieve that meaning, and writes them to the output file.

An *interpreter*, on the other hand does not write machine instructions. Instead, when it understands the meaning of a statement, the interpreter immediately attempts to achieve that meaning. It does this by selecting and calling appropriate sequences of instructions already built into the interpreter.

If the programmer shares the source code with other people, and they too have an interpreter installed for that programming language, they can run the program immediately, and do not need to recompile it. This is because their version of the interpreter is adapted for their version of operating system and computer.

Compiling a program is often more efficient, because the CPU can execute it in its native set of instructions. The drawback is the required compilation step before the program can run.

Using an interpreter means that the program can be executed directly by the interpreter, but also that even the smallest program needs the bulk of the interpreter to run. It cannot be a standalone program, because the interpreter is required in order to run the program.

1.4.1 The Java virtual machine

The Java programming language is both compiled and interpreted. The Java compiler outputs code for a virtual machine, a CPU that only exists in the form of an interpreter. The interpreter does not understand Java source code. It reads and executes the machine code of the Java virtual machine. That code is often called *byte code*, because of the way it is structured.

Creating and running a Java program thus involves the following steps:

- Write a text file with the Java source code
- Let the Java compiler translate the source code into byte code
- Start the Java virtual machine and let it interpret the byte code

While the above seems to involve the worst of two worlds, there is actually a huge benefit in terms of portability. Anyone who compiles a Java program, can then share the byte code with everyone who has a Java interpreter installed. Since the virtual machine inside the Java interpreter is identical everywhere, all Java programs can run on it.

Another benefit of the compilation into byte code is that the resulting binary becomes quite small, and can be quickly sent over across computer networks. This makes it possible for Java virtual machines to send byte codes between each other, and thus create advanced things like mobile agents, or remote code loaded on demand.

1.4.2 The java and class files

Java source codes are written in files that end with `.java`. The Java compiler reads the source code and generates files named `.class`. A single file of Java source code can generate one or more class files, depending on what it contains.

Source code files are edited with a text editor. It is often convenient to use an editor that understands the Java syntax and perform indentation, highlighting, and other services.

Sometimes the editor also helps with compilation, by running the compiler from a user command, and showing the output from the compiler when it needs to report on errors found in source code.

Requirements for Java programming

Before one can begin to write and run Java programs, certain assets need to be secured. These are:

- A source code editor
- The Java compiler *(javac)*
- The Java interpreter *(java)*

The Java interpreter is often installed as *a runtime environment* (rte). The rte contains other tools as well, and the libraries required to run Java programs.

The Java compiler has to be installed separately, as a Software Development Kit (SDK). The SDK includes the compiler and other tools needed to develop Java programs.

All three components, editor, Java interpreter, and Java compiler can often be found together by instead finding and downloading an Integrated Development Environment (IDE). Using an IDE can appear daunting at first, but it soon becomes very helpful as its very purpose is to streamline the programming process.

There are two development setups supported in the ID1018 Programming I course:

- The TextPad tool - This is a text editor that also supports compilation and running of Java programs. Development with TextPad goes through the traditional write-compile-run cycle.
- The Eclipse IDE - This is a complete IDE with support for projects, incremental compilation, context-sensitive documentation, autocomplete, and indentation.

Using the command line

This is how to compile and run a Java program from the commandline. A command window is a program that accepts text commands typed in by the user. On Windows it is called `CMD.EXE`, and on Linux the general notion is a *command shell*, and there are many such available, `bash` being a common one. If you have a Linux system, a virtual terminal with the default command shell can be opened with the keybord chord `ctrl-alt-T`.

Assuming that on a Linux system, the source code for a Java program resides in the file `HelloWorld.java`. The program can then be compiled with the command `javac` (the hashmark is the command prompt):

```
# javac HellowWorld.java
```

The java compiler (the `javac` program) expects to be told which *source files* to compile. It does not make assumptions about it, so the whole filename must be given, and it must end with `.java`.

If there is no output from the compiler, this is good. It means that the compiler accepted the source code and generated a `.class` file. Just to check, we can list the files:

```
# ls HelloWorld.*
HelloWorld.class  HelloWorld.java
```

When we want to run the program, we start a Java virtual machine (JVM) and tell it what to run. For reasons that will become apparent later, but at this point must be taken on faith, we do *not* tell the JVM which file to run. Instead we give it *the name of the class that contains the code where the program starts executing.* In this example, the name of that class is `HelloWorld`:

```
# java HelloWorld
Hello world!
```

The JVM will search for a matching `.class` file, and when it finds it will load it and begin to execute the byte code instructions in it. In this example, the text `Hello world!` is printed on the command window.

The above example would look like this in a Windows environment:

```
> javac HelloWorld.java
> dir/b helloworld.*
HelloWorld.class
HelloWorld.java
> java HelloWorld
Hello world!
```

1.5 Java language concepts

1.5.1 Classes, objects, and instances

Java is an object-oriented programming language. On the most general level this means that there exists only two kinds of entities in a Java program; primitive variables and objects. Primitive variables can only exist inside an object, so objects are the basic building blocks.

A class is the definition of an object. The class has a unique name, and its definition consists of a list of *members*. These members can be inner or nested classes, primitive variables, and *methods* (callable subprograms).

Some members belong only in the definition of the class. To indicate this the programmer declares them to be `static`.

By using the `new` operator on the name of a class, an *instance* of the class is created. A new copy of the object is created, and memory is allocated for all its non-static variables. It is common that a Java program defines a small number of new classes, and uses several of the classes that already exist in the Standard Library.

Here is the complete source code for the HelloWorld program:

```
public class HelloWorld { // (1) Name of class

  public static void main (String [] args) { // (2) main

    System.out.println("Hello world!"); // (3) output

  }
}
```

The definition begins on line (1) with the declaration that this is a class, it is public (available to all), and has the name `HelloWorld`. Then follows a *block* given by the matching pair of `'{' '}'` curly braces. This block surrounds the body of the class definition.

Class `HelloWorld` has only a single member, the static method named `main`. Again, a block of curly braces surrounds the body of the method.

Inside method `main`, there is a single line of code (3) which generates the output seen on the screen when the program is run.

Note how the lines of code are indented to the right to emphasize the block structure, and what part they belong to. The compiler does not require this indentation, but it makes the code easier to read for humans. The following source code has exactly the same meaning but is mangled horribly:

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

You should always strive to write well-formatted source code, that is easy to read and understand.

The main method

The `main` method is the starting point of a Java program. When the java interpreter (JVM) is asked to run a Java program, a class name is given to it. The JVM then locates a `.class` file that matches the given name. If the class also contains a `main` method, then that class is loaded and the main method is called. When the main method exits, so does the program.

Every `main` method must be declared exactly as in the HelloWorld example. It must be `public`, it must be `static`, it must be `void` (return no value), and it must accept a single parameter of type `String []` (array of strings). If these conditions are not met, the JVM will not recognise the method as the starting point of the program, and the launch will fail.

The `main` method is declared `static`. This means that it is part of the class definition, and does not require an object instance in order to be called. This makes sense, because we need to be able to run code before we have any instances, so that we can create such instances if we need them.

The argument to the `main` method is an array of strings. These strings are the commandline arguments to the program, if there are any.

A Java class can contain zero or one `main` method. A Java source file, can contain more than one class, but only one of them can have a `main` method, and the name of that class and the name of the source file must match exactly, letter by letter.

> For reasons of sanity it is strongly recommended to put each class in its own source code (`.java`) file.

1.5.2 Variables

Variables are, as the name suggests, containers of something that can vary in the course of the program's execution. All variables in Java are mapped to a location in the memory of the JVM.

The variables in Java consist of 9 different types, and each type have a different meaning, and require a different number of bytes of memory for its storage.

The *primitive* data types make up 8 of the possible variable types. Table 1.3 shows the primitive data types in Java. The size of the `boolean` type is omitted because it depends on the implementation of the JVM.

The 9th variable type is the *reference* variable. It does not contain a value; instead it refers to some object. In other programming languages this is called a pointer. Reference variables are not allowed to refer to everything. Instead, the

Type	Bytes	Range	Use
boolean		false or true	Value of a test
byte	1	-128 – +127	Raw binary data
char	2	0 – +65,535	Unicode characters
short	2	-32,768 – +32,767	Short integers
int	4	-2^{31} – $+2^{31}-1$	Integers
long	8	-2^{63} – $+2^{63}-1$	Long integers
float	4		Single precision floating point
double	8		Double precision floating point

Table 1.3: Java primitive data types

programmer must specify exactly what kinds of objects a particular reference variable can refer to. This is called the *reference type.*

1.5.3 Program statements

Program statements define what the computer is supposed to do when the program is run. There are basically three kinds of statements:

- assignments to variables
- method calls
- control constructs

Assignment statements are used to calculate the value of an *expression* and then store that value in a variable.

Method calls invoke named subprograms, with or without arguments. In the `HelloWorld` example, the statement:

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

is a call to the method `println` with the string `"Hello World!"` as the argument.

Control constructs are syntactic elements in the Java language that allows for the selection and iteration of statements. These are the selection control constructs:

```
if, if else, switch case
```

and these are the iteration control constructs:

```
while, do while, for
```

In addition, these modifiers affect control constructs when used:

```
  break, continue
```

1.5.4 Comments

Comments in Java source code are for human eyes (mostly). They serve to explain what the code is meant to be doing, how a method is supposed to be used, and what any return values mean. There are two ways of writing a comment in Java:

```
  // This comment ends at the end of the line
  /* This comment ends at the next star-slash symbol.
     It can span multiple lines. */
```

Comments are also useful for including and excluding diagnostic code, or experimenting with minor alternatives when developing.

> Good comments add insights that are not obvious from the program statements.
>
> Make a habit of writing good comments, because other people do not know how you were thinking, and you yourself *will* forget someday.
>
> When you update the source code, update the comments too.

1.5.5 Naming conventions

The Java language is a case-sensitive language. This means that upper- and lowercase letters are different characters. The names `foo`, `Foo`, `FOO`, and `fOo` are all different.

The language specification does not dictate how to use upper- and lowercase in the selection of names. However, there are certain conventions that when followed makes it easier to read and understand the source code. One such widespread convention is presented below.

Names for classes

The first letter of a class name is capitalized, the following letters are in lowercase:

```
public class Node
public class Amplifier
```

When several words are needed to make up the complete class name, use *camel case*, capitalizing the first letter of each word and then concatenating the words:

```
public class HelloWorld
public class StandardVehicle
```

Names for variables

The first letter of a variable name is in lowercase. If several words are needed, use camel case for the subsequent words:

```
int amount
int numberOfInstances
```

Names for constants

A variable that is declared to be `final` can only be assigned once. From then on it will keep its value and it cannot be changed. It is constant. Such variables are therefore called *constants*, and they are useful when defining limits and default values. Constants are written in all uppercase letters, and use the underscore characters to separate words:

```
final int LINES = 8;
final int NOF_BUFFERS = 32;
```

Names for methods

The first letter of a method name is in lowercase. If several words are needed, use camel case for the subsequent words:

```
... main (String [] args)
... isClosedAndLocked (Door d)
```

> Follow the naming convention, because you can then see from the name alone if it is a class name, a variable, a constant, or a method call.

1.6 Basic input and output

When the computer's operating system runs a program, by default it connects three streams to it. They are:

- the standard input stream (keyboard)

- the standard output stream (screen)
- the standard error stream (screen)

The standard input stream consists of characters typed on the keyboard. In a windowed environment, as is the case with most computers, the input stream is shared by several windows, and therefore is directed by *focus*, i.e. the active window.

A Java program that runs in a command window, receives keyboard input when the command window has focus (is active). A Java program that runs in Eclipse, receives keyboard input when Eclipse has focus and its internal console window has focus.

The standard output stream consists of characters generated by the program and sent to the standard output stream, such as by the `System.out.println()` method. Since the standard output stream by default is connected to the command or console window in which it runs, output from the program and input from the keyboard appears together.

The standard error stream is intended for error messages. By default it also goes to the command or console window.

Further features of interest:

- The command window can be used to redirect the three standard streams, so that input can be read from files or other programs, and output sent to files or other programs.
- A program (Java programs included) can open additional streams for reading and writing, for example to files or network connections.

When writing a Java program, these predefined reference variables refer to objects that represent the standard streams:

- `System.in` - the standard input stream
- `System.out` - the standard output stream
- `System.err` - the standard error stream

The objects that represent the two output streams (`out` and `err`) each have three useful methods for printing:

- `print(...)` - print something
- `println(...)` - print something and make a new line

- `printf(...)` - C-style formatted printing

Reading input from the keyboard is a little more complicated, for reasons that will be explained later. A taste of the issue is given by the following question: *if you see the character* `'1'`*, does that represent the number 1 or the digit 1?*

> To print something on the console, use `System.out.print()` or `System.out.println()`.

1.6.1 The Java Standard Library

In the Java runtime environment (rte) there exists a Standard Library of predefined classes. These are organized in *packages*, the highest level of organization in the Java language. Some of these packages are required in order to run even the simplest program, others are there for more special cases and demanding applications. It turns out, that three packages are more often used than others. They are:

- `java.lang` - Required to run a Java program, contains helper classes for the language itself
- `java.util` - A set of general utilities (e.g. collections, sorting) that many programs benefit from
- `java.io` - Classes that support opening additional I/O streams, for reading and writing files.

Importing packages

A Java program that wants to use a class from the Standard Library, can do so in two ways. The first alternative is to use the fully qualified name of the class, like in:

```
java.util.LinkedList
```

which refers to the class `LinkedList` in package `java.util`. This quickly becomes tedious though, because of all the required typing, so it is rarely used.

The second alternative is to *import* the desired package into the source code file. This is done with the `import` clause:

```
import java.util.LinkedList;
...
  LinkedList
```

When the compiler encounters the name `LinkedList` is sees from the import clause where that class is defined.

If several classes are imported from the same package, a fully qualified import clause is required for each. However, it is allowed to replace the class name with a wildcard, and thus make available *all* classes from the package:

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

> Import clauses must be first in the source code file, before the class definition.
>
> Do not use import wildcards unless you have a lot of imports from the same package.
>
> The package `java.lang` does not need to be imported. It is always available, because it is always required.

1.7 The JVM, memory, and garbage collection

When the Java Virtual Machine is started to run a Java program, it allocates a certain amount of memory from the operating system, the exact size of which depends on the JVM and the amount of physical memory installed on the host computer. This memory is called *the heap* because of the data structure in which the JVM keeps it. The heap is used for dynamic memory requirements, such as the bytecode for loaded classes and memory for object instance variables.

When an object is created, the JVM needs to have three things in memory: the bytecode for the constructors and methods in the class, memory for the static class variables, and memory for the instance variables. When additional instances of the same class are created, more memory is needed for the new instance variables. A particular and unique object instance is therefore basically represented by the chunk of memory that holds its instance variables.

A program creates and discards object instances as it executes. The discarded objects are those *to which the program has no reference.* The code in the program can no longer access those instances.

Repeated object creation means that the heap of free memory becomes progressively smaller as the program runs. When the amount of free memory falls below certain limits, the JVM begins a process called *garbage collection.* Garbage collection (or *gc* for short) scans all the object instances that have been created in order to determine which ones are still active (referred to by the program), and which ones are garbage (discarded by the program). The garbage objects are then recycled, and their memory placed back into the heap of free memory.

Garbage collection is usually completely transparent to both the running program and its users. When all goes well it is not noticed, except perhaps for a small, temporary slowdown. In the early years of Java, garbage collection could be rather crude, and in effect make the program appear unresponsive for several seconds. The reason for this was that the JVM basically stopped executing the user's program and instead devoted itself to the task of garbage collection for a while.

These days better garbage collection algorithms have been developed, computers are faster, and memories bigger. However, it is still perfectly possible (and actually quite simple) to write a Java program that consumes all the memory in the heap. When that happens, the program terminates with an `OutOfMemoryError`.

When the heap is exhausted the JVM (and the program) terminates. The JVM does not ask the host operating system for more memory. A program that needs more memory than what the JVM offers by default, must launch the JVM with the appropriate commandline parameters to allocate more memory from the start.

A poorly written program may suffer from a phenomenon called a *memory leak*. A memory leak occurs when the program allocates more dynamic memory than it returns. If the program runs long enough, it will eventually consume all the available dynamic memory.

The failure to return dynamic memory may be a bug; the programmer believes that objects have been discarded when in fact they are not. It may also be a design flaw, like constantly adding elements to a list without considering if there should be a maximum length for the list.

Most programs do not suffer from memory issues. They do not process that much data, or they do not run long enough. Server programs, that are to run continuously, must be free of memory leaks or they *will* crash eventually.

The following sections outlines some best practices to protect against memory leaks.

1.7.1 Discard unused objects

As long as the running program maintains a reference to an object, that object will not be garbage collected, nor will any of the objects referred to by that object. Discard objects that are no longer needed by setting the variables that refer to them to `null`.

1.7.2 Discard and recreate

When running in a memory-constrained environment, like a cell-phone, wearable, or embedded system, it may be prudent to trade memory for time. A huge lookup table may speed things up, but if it is only needed in certain phases of the program, perhaps it is more economical to discard it when it is not needed, and then spend the extra seconds to recreate it when it is required again.

1.7.3 The size of dynamic data structures

Dynamic data structures like lists, hashtables, sets, trees, and so on, will continue to grow unless they are discarded or trimmed for size. A list of log messages, could perhaps be a queue of log messages, so that old entries will be discarded. If old data must be preserved, can it be replaced by summaries, or offloaded to a file or database?

1.7.4 Hidden references

A program that uses support libraries like middleware, graphic tools, and so on, very often does not know the internals of those libraries. The library code may do all sorts of things that consume memory, and in ways that are not at all obvious. Read the documentation carefully, and follow the protocols and procedures to close files and connections, terminate threads, and deactivate resources.

Chapter 2

Data storage

This chapter is about variables and objects. Both occupy memory when a program is running, and both hold and represent data.

2.1 Variables

Variables are storage locations in primary memory. Variables only exist while the program that defines them is running. They are collectively called variables, because the contents of memory is expected to *vary* in ways defined by the program. Without variables it would be difficult to store input data, to compute expressions, and to assign results to memory.

Each variables is given a name (see 1.5.5 for naming conventions) which should be carefully chosen by the programmer. The name should be neither too short or too long, and it should aptly reflect on the purpose of the variable. Choosing good variable names is an important part of the programming skill.

2.1.1 Data types

The concept of a *data type* is necessary because at the bottom of the machine, both physical or virtual, all that exists are patterns of bits. To define the *meaning* of each such pattern, we need to establish some rules of interpretation. Without such rules, it is far too easy to compute results that makes no sense, or worse, appear to make sense but are in fact wrong.

In Java, the data type concept has two flavours, but they are actually just different sides of the same problem; to make sense of the bit patterns. For the primitive data types (`char, int` etc), the data type defines how many bytes that

Access modifier	Member	Assignment	Type	Name
`private`	(default)	(default)	`boolean`	*identifier*
`protected`	`static`	`final`	`byte`	
(default)			`char`	
`public`			`short`	
			`int`	
			`long`	
			`float`	
			`double`	

Table 2.1: Variable declaration

are used to hold one item, and how the bits in those bytes are to be interpreted as a whole group.

For the reference types, i.e. references to objects, the value itself is always an address in the JVM memory, so instead it is the object that is referred to that determines the type. Since all objects are defined by a class, the class is the type for reference variables.

In general, it is important to remember that a data type is defined by the programming language. It is not dictated by the underlying machine. Instead, the interpreter of a language, or the runtime libraries of a compiler, must map the data type of the language onto the physical memory of the current computer. Of course, there are exceptions from this, in particular in programming languages like C that prefer to be close to the machine for reasons of efficiency.

2.1.2 Variable declaration

Before a variable can be used it must be *declared*. The declaration tells the compiler the name of the variable, the type it has, and the variable's *scope*. The scope determines from where in the source code the variable may be accessed.

Table 2.1 describes how to declare a primitive variable. For a class variable (a variable that is a member of a class, alongside methods and inner and nested classes) exactly one entry is chosen from each of columns. The entry (default) indicates the empty choice from that column. For example:

```
private int foo;
long sum;
public static final double MAX_LEVEL = 1.0;
```

The access modifier only applies to class variables. It determines how visible or hidden the variable is from other classes. A class variable is always visible inside the class that declares it.

The `static` modifier declares that the variable is part of the definition of the class, and not a member of instances of the class. A static class variable only exists in one copy, the one in the class definition. A non-static instance variable is given memory for each new object instance that is created of its class, and the variable does not exist at all except in such instances.

The `final` modifier indicates that the variable can only be assigned once.

The data type is the chosen type of the variable, and the *identifier* is the name given to it. There are certain rules that apply to identifiers, because the rules make it easier to build compilers:

- The identifier must begin with an underscore or a letter.
- The rest of the identifier may only contain letters, digits, and the underscore character.

The scope of a class variable can be the declaring class (private), classes that inherit from the declaring class (protected), classes in the same package (default), or every other class (public).

Static variables can be reached from static code and from code in object instances. Instance (non-static) variables, can only be reached via a reference to an object, or from non-static code of the declaring class.

Local variables

Local variables exist in methods, as the method parameter variables, and any variables declared inside of a block (a block is a piece of code surrounded by '{' '}' braces).

Local variables follow the simple rule that the block currently executing can access local variables in its own block and the surrounding outer blocks. For example:

```
{
  int x = 10;
  {
    int y = x - 5; // OK, because x is declared
                   // in an outer block
  }
  x = y - 2;       // Error; variable y no longer
                   // exists, the block that de-
                   // clared it has been exited.
}
```

Local variables are allocated differently from class variables. While static variables exist throughout the program's whole execution, and instance variables appear and disappear with the object instances, local variables only come into existence when the thread of execution enters the block in which they are declared. When execution exits the block, the local variables in the block are no longer available.

Care should be taken when naming local variables so that they do not *shadow* class variables, or local variables in surrounding blocks.

Local variable details

The technical reason for the behaviour of the local variables is that class variables are allocated from the heap, while local variables are given space on the method call stack. A consequence of this is that when method `m1` calls method `m2`, all of `m1`'s context (point of execution and local variables) is suspended. The code in method `m2` cannot access any of the variables in `m1`, because it has now pushed its own local variables on the call stack, on top of those of `m1`. Only when `m2` returns and method `m1` resumes are its local variables available again, and then only to method `m1`.

An important point to remember from this discussion, is that when a method is called, all its local variables are unique to that call. This provides isolation between method calls, and also makes *recursion* feasible, i.e. that a method calls itself.

2.1.3 Assignment

A lot of the work in a computer program is to compute expressions, and to assign the results of the expressions to variables. For example:

```
int x = 0; // Declare local variable x, initialized to 0
x = 1 + 2; // Compute the expression 1 + 2 and assign
           // the result to x
System.out.println(x); // Print the value of x
```

As can be seen on the first line of the example, the local variable `x` is both declared and assigned its first value. This *initialization* is very common for class variables, and a requirement for local variables.

The general pattern of an assignment statement, is this:

variable = *expression* ;

As will be seen later on, the left-hand side of the assignment is actually an expression too, one that evaluates to a variable.

The assignment operator, the `'='` symbol, is not mathematical or logical equality. Instead it should be read as an arrow pointing from right to left, in the direction the data flows. The result of the expression overwrites the previous contents of the memory location represented by the variable name. A better symbol for assignment would be to write:

$$\textit{variable} \leftarrow \textit{expression} \;;$$

but tradition and the ASCII code legacy prevents that.

2.1.4 Constants

Constants are variables that have been declared `final`. This means that they can only be assigned once, and often this is done in the initialization of the variable declaration, like this:

```
final int TOP_LAYER = 3;
```

If a final instance variable is not initialized by its declaration, it must be assigned by some other part of the object instance creation code. The constant must have been given its value before the newly created object instance is handed over to the program.

Names of constants are by convention spelled in all uppercase, with words separated by the underscore character.

More on final

The `final` modifier have no computational semantics. It is just a way to ask the compiler to check that no assignments are made to the variable, except the first one. A source code with `final` declarations in it that compiles without error, behaves the same without the `final` modifiers. This is not to say they should be removed or omitted, properly used they help with writing correct code.

It is possible to declare local constants, they are just local variables declared as `final`. In addition, it is allowed to declare method parameters as `final`. For example:

```
public void foo (final int x) {
  ...
}
```

The parameter variable `x` contains the argument provided by the caller. Declaring `x` to be `final`, disallows any assignments to `x` in method `foo`. This is however redundant, because parameter passing in Java is by value. This means that arguments are copied, so a method cannot modify the caller's context.

The `final` declaration on parameter variables is therefore useful in two ways: it helps to avoid an assignment by mistake in the code of the method (and risk losing the actual argument). It also instructs the programmer that parameter `x` is only meant to be read, and this is particularly important when the parameter variable has a reference type.

2.2 Primitive data types in Java

The primitive data types in Java are:

- boolean - logical value, `true` or `false`
- byte - one byte, signed integer, -128 – $+127$
- char - two bytes, unsigned integer, used for Unicode numbers
- short - two bytes, signed integer, $-32,728$ – $+32,767$
- int - four bytes, signed integer, -2^{31} – $+2^{31}-1$
- long - eight bytes, signed integer, -2^{63} – $+2^{63}-1$
- float - four bytes, single precision floating point, $\pm 1.4 * 10^{-45}$ – $3.4 * 10^{38}$, approximately 6 significant digits
- double - eight bytes, double precision floating point, $\pm 4.5 * 10^{-324}$ – $1.78 * 10^{308}$, approximately 15 significant digits

2.2.1 boolean

The `boolean` data type can only hold two values, `true` and `false`. It is used to store truth values, for example the outcome of a test, or a flag that indicates that certain things should be done in various parts of the program.

2.2.2 byte

The `byte` data type is a single byte, 8 bits of data. It can be used to hold small values, but its real usefulness comes when exchanging raw binary data with files or other processes. A block of memory can then be represented by an array of bytes, and imported or exported efficiently.

2.2.3 short

The `short` data type occupies two bytes, or 16 bits of data. It is preferable over the `int` only when you need a lot of 16-bit data in memory at the same time.

2.2.4 char

The `char` data type is used to hold character values from the Unicode character standard. The char data type takes two bytes (16 bits) and is unsigned, which means that in terms of numeric values, its lowest value is zero and its highest value is $65,535$.

The use of 16 bit Unicode numbers is a compromise, because there are Unicode characters with numbers that need more than 16 bits. For such exotic characters one must use an `int` instead, and the corresponding Unicode number is then called a *codepoint* in the Java documentation.

For normal use the `char` data type is perfectly sufficient to represent the characters in a text.

2.2.5 int

The `int` data type is the four byte (32 bits) workhorse of integer arithmetic. It is a good first choice for most integer variables. It takes up four bytes, which is a good match to most computer architectures, and therefore corresponds well to a word of computer memory.

2.2.6 long

The `long` data type has eight bytes (64 bits) which is sufficient in most cases where numbers that do not fit into an `int` are needed. In practice, the most common use for `long` consists of counters that are not expected to wrap around in the lifetime of the program, even if it runs continuously for many years.

System time is returned as a `long`, holding a count of the number of milliseconds that have passed since 1 January 1970. It is expected that a `long` will suffice to represent time this way even after the Sun itself has exploded in 5 billion years.

Sometimes a `long` can also be used to hold a group of bits that need to be manipulated together, like a fast set of up to 63 members (it is common to sacrifice the 64th member to avoid the trouble of negative numbers when manipulating the set).

2.2.7 float

The `float` data type allows for an approximation of real numbers according to the standard IEEE 754. This is done by dividing the 32 bits (four bytes) of the `float` into three fields:

```
[sign (1)][mantissa (23)][exponent (8)]
```

The number that corresponds to a certain bit pattern is then found by calculating:

$$(-1)^s * m * 2^{(e-127)} \tag{2.1}$$

where s, m, and e are the sign, mantissa, and exponent, respectively.

The advantage of the floating point format is that by virtue of the exponent one can represent numbers from a large range. It is also possible to use the bits in the mantissa efficiently. If, for example, a value is small, like $0.000,000,123$, the exponent can be used shift the decimal point to the right, avoiding the need to store the leading zeros. The decimal point is floated to the optimal position.

It should however be remembered that there are only 2^{32} different bit patterns in 32 bits, and there are infinitely many possible numbers between any two real numbers. This means that most floating point numbers can only be stored as an approximation of the true value.

The approximate nature of floating point values, means that repeated calculations will accumulate roundoff errors. In general the roundoff errors cancel each other out, but this only delays the degradation of the least significant digits, it does not prevent it.

2.2.8 double

The `double` data type is a floating point data type according to the IEEE 754 standard. It uses 8 bytes (64 bits) divided into three fields:

```
[sign (1)][mantissa (53)][exponent (11)]
```

As can be seen by the 53 bit mantissa, the `double` data type can represent real numbers with higher precision than the `float`. The bigger exponent also gives it a greater range.

It is important to remember that the term "double" refers to the storage size in memory, and not to the precision. Since each added bit doubles the number of possible values in a binary number, the `double` data type is $2^{53-23} = 2^{30} = 1,073,741,824$ times more precise than a `float`.

Always use `double` for floating point, unless you have good reasons to do otherwise.

	To						
From	byte	char	short	int	long	float	double
byte	-	no	yes	yes	yes	yes	yes
char	no	-	no	yes	yes	yes	yes
short	no	no	-	yes	yes	yes	yes
int	no	no	no	-	yes	yes	yes
long	no	no	no	no	-	yes	yes
float	no	no	no	no	no	-	yes
double	no	no	no	no	no	no	-

Table 2.2: Automatic type conversions

2.3 Type conversions

With so many different primitive data types, it is common that values must be transferred between variables that are of different data types. It is also common that the data type of an expression does not match the type of the variable into which the calculated value is to be stored.

To enable variable assignment between values with different data types, *type conversion* must be done. The type conversion can often be *automatic* and inserted by the compiler. However, sometimes it is the case that the conversion requested by the source code may result in loss of data. In such cases the compiler will refuse to compile the source code and report an error. The programmer must then add some code to indicate that an *explicit* type conversion is to take place.

2.3.1 Automatic type conversion

Automatic type conversion happens in two situations. The first is during the evaluation of expressions, where the compiler will first convert all terms in the expression into the largest data type used in the expression, before actually calculating it. The result of the expression will be of that data type.

The second situation is the assignment to a variable. If the variable that is assigned has a data type that can receive all possible values of the assigned data type, the value is first converted to the data type of the variable and then stored in it. Remember that assignment to variables happen both in assignment statements and in method calls where values are assigned to the parameter variables.

A rule of thumb is that smaller data types can be automatically converted into larger data types, because the larger data type has enough bits to represent all possible values of the smaller one. This is also called a *widening* conversion.

Table 2.2 gives an overview of the allowed automatic type conversions. Some features of interest:

- a `byte` (8 bits) and a `short` (16 bits) can not be automatically converted to a `char` (16 bits), because the `char` is an unsigned data type. It cannot represent negative numbers, while `byte` and `short` are signed, and *can* represent negative numbers.
- a `char` (16 bits) can not be converted into a `short` (16 bits) because of the possible loss of precision. The unsigned `char` can hold the values 32,768 – 65,535, which are out of reach of the signed `short`.
- any integer data type can be converted to a floating point data type, even though the mantissa may not be able to receive all bits of the integer value. The result will be rounded to the nearest possible value.
- no floating point value can be automatically converted to an integer data type, because there may be a loss of precision.

2.3.2 Explicit type conversion

When a value is to be assigned into a smaller data type, is it called a *narrowing* conversion. Often this is practical to do in a program because the programmer is certain that the conversion will be safe, or that a loss of precision is of no consequence.

A *typecast* expression is used to assure the compiler that the conversion is intended, and not a programming mistake. The value is cast into its new data type, by prefixing the expression with the name of the desired data type, in parentheses:

```
int i = 9;
...
byte b = (byte) i;  // The int is cast to a byte
```

When a floating point value is cast into an integer data type, the programmer must first choose if any decimals are to be used to round the value to an integer, or if they can be discarded (truncated). If rounding is desired, it must be explicitly performed by way of a method call. In this example the method `round()` in the standard library is used. The method accepts a float parameter, rounds it, and returns the resulting `int`, so there is no need for a typecast:

```
float f = 1.5f;
...
int i = Math.round(f); // Call to std library routine
```

In the following example, any decimals are simply discarded, and only the integer portion of the floating point number remains:

```
float f = 1.5f;
...
int i = (int) f;  // i will be assigned the value 1.
```

Each explicit type conversion should be considered carefully. An Ariane 5 rocket exploded in 1996 because of an unforeseen data type mismatch and conversion.

2.4 Object data types

In addition to the primitive data types, there are also *object data types.* Object data types consist of *classes* and *interfaces* (a special kind of Java class).

All classes in the Java language, regardless of if they come from the Standard Library or are defined in a small demo program, are part of the same inheritance hierarchy. At the absolute top of this hierarchy is class `java.lang.Object`. Every class that does not explicitly *extend* (inherit from) another class, inherits automatically from `Object`. As a result, all classes have class `Object` as its highest superclass (except class `Object` itself).

Inheritance in Java is strictly singular, a class can only have one immediate superclass (parent). On the other hand, a class can have any number of subclasses. The class hierarchy forms a tree, with class `Object` at the root, and non-inherited classes at the leaves. Some branches of the tree have a very deep structure, with many levels, while many other classes are directly below class `Object`.

Each node in the class hierarchy is a class, and therefore also a data type in the Java language.

2.4.1 Reference variables

Variables of a reference type point, or *refer* to, an object instance. The only exception is when the variable happen to refer to nothing, in which case it has the special value `null`.

Just as with primitive data types and variables of that kind, reference variables have a data type. The data type of a reference variable is the class that it can refer to, and the value that can be assigned to it is a reference to such a class. The declaration of a reference variable follows exactly the same pattern as that of a variable for a primitive data type:

```
String s = null; // s can refer to instances of String
Object o;        // o can refer to any object
Integer i = new Integer(8); // i refers to an Integer
```

Just as with primitive data types, type conversions are possible with object data types. Moving up in the class hierarchy corresponds to a widening conversion and is automatic, while descending towards a leaf corresponds to a narrowing conversion and will require an explicit type cast.

2.4.2 Class String

Class `java.lang.String` is used to hold character strings, like `"Hello world!"`. For each string an object is created with just enough memory to hold the Unicode characters in the string. The string has a length, which can be obtained, and the class defines many other useful methods. For example:

```
String hw = "Hello world!"; // declare and initialize hw
int len = hw.length();      // how many characters in hw?
boolean b = hw.isEmpty();   // is hw empty?
```

> The string variable `hw` in the example above does not contain the string. The variable *refers to the nameless object* that contains the string. The variable can be reassigned to refer to some other string, if so desired.

String creation

In most cases the `new` operator is required in order to create a new object instance. Strings, however, are so common in programs that the compiler allows for a simpler syntax. A string constant in the source code will automatically create the string object instance:

```
String s = new String ("Hello!"); // this works
String t = "Hello!";              // this works too
```

There are also several other constructors available in class `String`, for situations that may require them.

Strings are immutable

Strings in Java are *immutable*, i.e. once created they cannot be modified. To obtain an different string of text, a new `String` object must be created, using a *string expression.*

String concatenation

Strings expressions are often needed to create text, for output to the screen, to file, or to the internet. Since this is another common operation, the `+` operator is *overloaded* to perform string concatenation:

```
String name = ...  // some text is assigned
...
System.out.println("Name: " + name);
```

Since strings are immutable, the concatenation operator must create a new `String` object large enough to hold the text from both argument strings. The new string is created, initialized, and the reference to it is given to the printing method.

Strings are `equal` not `==`

String constants in the program source code are converted to string objects by the compiler. The compiler often tries to save memory, so if it sees two string constants that are equal, it will only create one string object. Two strings are equal if their character sequences are identical.

A common beginner's mistake is to attempt to compare strings using the identity operator `==`. This operator returns `true` if the bit pattern on the left side is identical to the bit pattern on the right side. When applied to strings, it does not look at the text in the strings. Instead, it looks at the references. If the references happen to refer to the same object, then the `==` operator returns `true`, otherwise `false`. This sometimes gives the illusion that the `==` operator can be used to compare the text of strings, *because the compiler has redirected the references to the same string object.* It is however easy to break:

```
String color = "Yellow";
...
if (color == "Yellow") // true, because there is only one
  ...                  // "Yellow" String object

color = "Yell" + "ow"; // new string object created with
                       // the text "Yellow"
...
if (color == "Yellow") // compares false because the object
                       // references are now different
```

The correct way to compare the text of strings, is to ask one `String` to compare itself against some other `String`, using the `equals` method. Thus:

```
if (color.equals("Yellow")) // compares the texts

if ("Yellow".equals(color)) // also works; the compiler
                            // replaces the string constant
                            // with an object reference
```

Chapter 3

Standard input and primitive values

3.1 Basic I/O

Java programs are executed by the Java virtual machine (JVM), a program that emulates an abstract computer. In order to provide useful work, however, this abstract computer must be able to connect to data sources and data targets in the real world. Typical data sources are network connections, files, and the computer keyboard. Typical data targets are network connections, files, and the computer screen.

3.1.1 The streams concept

Just as the JVM abstracts the physical computer, data connections are abstracted into serial *streams*. This makes is possible to read and write data uniformly across different systems. A stream in Java is always a stream of some data type, for example a stream of `char` for text, or a stream of `byte` for raw data. It can also be a stream of objects, if required.

3.1.2 The standard output stream

The most common output for a program is to compose text and present it on the scomputer screen. For this reason, the standard output stream of a program (including the JVM) is directed to the computer screen or console. In Java, the object referred to by `System.out` is connected to the standard output stream, and calling its methods will generate output on the screen.

The standard output stream is open by default and requires no further action by the program to be used. It is a stream of `char` and when the Java program writes Unicode characters to it, the JVM will translate each Unicode character into the character encoding that the data is to have outside of the JVM. The default character encoding is specified by the local system. This arrangement works well on most systems, but sometimes creates difficulties for non-ASCII characters, such as on Windows where the code pages in the general system and the command window are different, for historical reasons.

To print text on the screen, the program calls the appropriate method using `System.out`:

```
System.out.print(42);  // prints 42
System.out.print(" and "); // prints the string
System.out.println(39); // prints 39 and a newline
```

The example generates the following output:

```
42 and 39
```

The methods `print` and `println` are *overloaded* to accept all possible kinds of arguments. This means that there are several versions of the `print` method defined, e.g.:

```
void print(boolean b)
void print(char c)
void print(int i)
...
```

Each method performs the appropriate conversion of its argument to a text, and then sends the characters in that text to the output stream. The compiler will pick the method that matches the argument, and generate a call to it. This is convenient for the programmer, because anything and everything can be printed using the same method name.

3.1.3 Standard input

By default, the standard input stream is connected to the keyboard, and when the window or console that started the JVM has focus, characters typed on the keyboard can be read by the Java program. The input is line buffered to enable the correction of typing mistakes. This means that the Java program will not see the next line of input until the enter key has been pressed.

An object representing the standard input stream is available on `System.in`, but in difference to the standard output stream, it does not by default provide character decoding. All it can do is to present the next byte from the input, and it is up to the program to decode it. There are, however, several classes in the Standard Library that helps with this task.

3.1.4 java.util.Scanner

The `java.util` package contains utilities that are useful in many programs. The class `java.util.Scanner` is used to convert bytes from an input stream into text strings and numbers. It sits on top of an input stream, and when asked to produce the next item of input, it will perform the necessary decoding and parsing, before returning the result.

Here is an example of how to read user input using class `Scanner`:

```
import java.util.Scanner; // Imports must be first
import java.util.Locale;  // (1)
...
Scanner sc = new Scanner(System.in); // Create a new
                          // instance of Scanner that
                          // reads from standard input

sc.useLocale (Locale.US); // accept decimal period (1)

System.out.print("Enter number of days: "); // prompt user
int nofDays = sc.nextInt(); // read an int

System.out.print("Enter amount: "); // prompt user
double amount = sc.nextDouble(); // read a floating pt

// Compute and present total
double total = amount * nofDays;
System.out.println("Total amount due: " + total);
```

The use of the `Locale` package at (1) warrants an explanation. Class `Scanner` needs information on how to interpret some characters as they appear in certain contexts, one being how decimal numbers are formatted when they appear as text. Default rules are available from the local system, and on a Swedish system the default is to use a decimal comma, e.g. 3,1415. But many computer users are accustomed to enter decimal numbers using a period, like they do in on American systems, e.g. 3.1415. By configuring the scanner to apply US rules, the decimal period is accepted.

Characters, strings, and numbers

The scanner sets up the byte input stream to be decoded into Unicode characters. Then it reads the Unicode characters, and depending on what method is called in the scanner, it will attempt to understand the characters differently. The method

```
String Scanner.nextLine()
```

reads characters until it comes to a newline. All characters found before the newline, but not including it, are returned as a string. The newline itself is discarded.

The method

```
int Scanner.nextInt()
```

reads and discards whitespace (blanks, tabs, newlines) until it finds a character that is allowed to begin an integer number. Then it continues to read characters while they are part of the integer number. When no more number characters are found, the scanner stops. The characters found in the number are then parsed to form a binary `int`, and that value is returned.

If the `nextInt()` method comes to the end of the input without having found the start of a number, it will block and wait until input is available. This is why the program appears to stop and wait for the user to type something. When the user hits enter, the next line of input is available for reading and scanning can continue.

The number parsing methods in class `Scanner` expects to find a number that can be parsed and returned. If they find some other character instead, a character that is not whitespace and not the start of a number, they will indicate an error by throwing an `InputMismatchException`. For simpler programs this means that the program ends with an error message.

Parsing numbers

It is important to understand the difference between the text representation of a number, and the binary representation of a number. The text representation is a sequence of characters in some character encoding scheme, that when printed looks like a number. For example:

```
"421"
```

In the Unicode representation used in Java, this character sequence consists of three `char` values:

```
Unicode symbol '4' char value 52
Unicode symbol '2' char value 50
Unicode symbol '1' char value 49
```

Each `char` value requires two bytes of memory, so the whole sequence occupies $3 * 2 = 6$ bytes.

When the character sequence is *parsed* into binary form, each individual character is read and transformed into its corresponding binary number. The numbers are then combined according to the radix (number base) used.

These expressions gives a hint of how number parsing is done (the decimal Unicode for '0' is 48):

$$n = (\text{'4'} - \text{'0'}) * 100 + (\text{'2'} - \text{'0'}) * 10 + (\text{'1'} - \text{'0'}) * 1 \tag{3.1}$$
$$n = (52 - 48) * 100 + (50 - 48) * 10 + (49 - 48) * 1 \tag{3.2}$$
$$n = 4 * 100 + 2 * 10 + 1 * 1 \tag{3.3}$$
$$n = 400 + 20 + 1 \tag{3.4}$$
$$n = 421 \tag{3.5}$$

A more algorithmic approach is given by the following steps:

```
n = 0            // n ==   0
n = n * 10       // n ==   0
n = n + 52 - 48  // n ==   0 + 4 ==   4
n = n * 10       // n ==  40
n = n + 50 - 48  // n ==  40 + 2 ==  42
n = n * 10       // n == 420
n = n + 49 - 48  // n == 420 + 1 == 421
```

The binary number corresponding to decimal 421 looks like this in its `int` form:

```
0000 0000 0000 0000 0000 0001 1010 0101
```

3.2 Expressions

An expression specifies something that can be evaluated to a value. An expression is therefore used when a value is needed, for example in an assignment to a variable or an argument to a method call.

In its simplest form the expression is a constant value, which evaluates to itself. The name of a variable is also simple, the evaluation finds the memory represented by the name and retrieves the value in it:

```
int k = 9;             // the value of the expression is 9
System.out.println(k); // the value of the expression is 9
```

More complicated expressions that compute new values can be formulated by using operators and functions. Operators provide standard arithmetic, while functions are methods that provide more complicated and rarely used computations (e.g. roots, logarithms, powers). For example:

```
i + 1
```

The operator `+` adds the value on its left side (variable `i`) to the value on its right side (constant `1`), and their sum is the value of the expression.

The arguments to an operator are themselves expressions.

A slightly longer example:

```
5 + 7 - 2
```

In this expression, `5 + 7` is evaluated first, giving the result `12`. The work that remains becomes:

```
12 - 2
```

The `-` operator subtracts `2` from `12` and the result is `10`. Since no further evaluations are to be done, this is also the result of the whole expression.

Most operators are *binary*, in the sense that they take two arguments, one on the left side and one on the right side. Two operators are *unary* (arithmetic negation `-` and logical negation `!`) and take only one right-hand side argument:

```
a = -a;  // Arithmetic negation
b = !b;  // Logic negation (boolean)
```

One *trinary* operator exists and it is the operator `?`. The syntax for it is:

test `?` *true result* `:` *false result*

First the *test* (logical condition) is evaluated. If it evaluates to `true`, then the value of the `?` operator is *true result*, otherwise it is *false result*. For example:

```
y = x < 0 ? -x : x;
```

This is evaluated as follows: *if x is smaller than zero, let the value of the* `?` *operator be minus x, otherwise the value is x.* This is equivalent to finding the absolute value of `x`.

3.3 Operations on integers

Integers can be added, subtracted, multiplied, divided, and taken the remainder (modulo) of. There is also unary negation, and a set of bitwise logical operations that can be used to manipulate one or more bits in the integer value. Table 3.1 gives a summary of the integer operators.

Automatic type conversions

The integer arithmetic operators are defined for the `int` and `long` data types. This means that if a component of the expression is a smaller integer data type, like a `byte`, `short`, or `char`, it must first be *promoted* (converted) to an `int`.

Operator	Operation
`*`	Multiplication
`/`	Integer division
`%`	Integer remainder
`+`	Addition
`-`	Subtraction
`-`	Unary negation
`&`	bitwise logical AND
`|`	bitwise logical OR
`^`	bitwise exclusive OR
`~`	unary bitwise complement

Table 3.1: Integer arithmetic operators

Similarly, if any component of the expression has a `long` data type, smaller data types are first promoted to `long` and the operators for the `long` data type are selected.

The promotion process is managed by the compiler, by creating invisible temporary variables and selecting operators. Here is an outline of the process, when the compiler arrives at the following expression:

```
short s ...
long  n ...

long r = s * n; // the expression to compile
```

The compiler then outputs code similar in spirit to this:

```
long temp0 = (long) s; // s is promoted to a long
r = longAdd(temp0 , n); // the long add operator is used
```

Over- and underflow in results

Let us examine the following variables and expressions:

```
byte u = 34;
byte v = 66;
byte w = u + v; // byte valued expression
```

In the assignment to variable `w` the compiler must first promote the values in `u` and `v` to the `int` data type, because there is no `+` operator for bytes. Then the addition can be performed, and the result is an `int` value. Normally an `int` value cannot be assigned to a `byte` without an explicit typecast, but in this case the compiler will simply place as many bits of the result that it can into variable `w`. This works, as long as the result is not greater or less than what a `byte` can hold. However, if the result is greater than 127 an *overflow* occurs, and the value stored in `w` is not the true result. Similarly, if the result is less than -128, there is an *underflow* and the wrong result is stored.

Multiplication can easily overflow a result if the factors are big enough. Multiplying two 8-bit numbers may require 16 bits to hold all of the result, because:

$$2^a * 2^b = 2^{a+b} \tag{3.6}$$

To hammer home the point, consider that the largest positive value of a Java `byte` is 127. Now then, $127 * 127 = 16,129$ which in binary form looks like this:

```
0011 1111 0000 0001
```

It is obvious that at least two bytes are required to hold this result.

Over- and underflows in integers are not detected at runtime. Instead it is the programmer's responsibility to choose data types appropriate for the task and the expected results of expressions.

A good way to protect a program against unwelcome surprises is to check the input data before it is calculated upon. If the program is designed to compute accurate results for a certain range of inputs, then it should verify that the inputs indeed are in within the specified limits before attempting to compute on them. Such range-checking makes for more robust programs, and is absolutely essential in real-time control systems like self-driving cars, airplane autopilots, or medical equipment.

Integer division

One particular thing to remember about integer arithmetic concerns division. Since the integer data types only can represent whole numbers, only the integer part of a result is kept. For example:

```
1 / 3 == 0
2 / 3 == 0
3 / 3 == 1
4 / 3 == 1
```

To find the remainder of an integer division, the modulus operator `%` can be used. In the expression

```
a % b
```

the result is what remains when as many `b` as is possible has been removed from `a`. For example:

```
 5 % 3 == 2 //  5/3 is 1,  5-(1*3)=2
11 % 4 == 3 // 11/4 is 2, 11-(2*4)=3
 7 % 9 == 7 //  7/9 is 0,  7-(0*9)=7
```

A more pragmatic view of the modulus operator can be gleaned from the following list of expressions:

Operator	Operation
`*`	Multiplication
`/`	Division
`+`	Addition
`-`	Subtraction
`-`	Unary negation

Table 3.2: Floating point operators

```
 0 % 5 == 0
 1 % 5 == 1
 2 % 5 == 2
 3 % 5 == 3
 4 % 5 == 4
 5 % 5 == 0
 6 % 5 == 1
 7 % 5 == 2
 8 % 5 == 3
 9 % 5 == 4
10 % 5 == 0
11 % 5 == 1
...
```

Notice how the result increases by 1, and then wraps back to zero on every fifth position. In the modulus expression `a % b`, the result will always be in the range $0 \ldots b-1$ (for non-negative `a`). This has some practical applications, like staying inside a circular list, finding the appropriate column for an element, or extracting digits from a number. For example, a number is even if it can be evenly divided by 2:

```
if (n % 2 == 0)
  // number is even
```

3.4 Operations on floating point numbers

The floating point data types `float` and `double` have only the standard four arithmetic operators, as shown in table 3.2. Additional operations and definitions are available in the standard library classes `java.lang.Float` and `java.lang.Double`. For general trigonometric and mathematical functions, the class `java.lang.Math` should be consulted.

Here are some typical examples of expressions involving floating point numbers:

```
double d = 10;    // (1)
double g = 1.015; // (2)
double r = d * g; // (3)
```

There are a few things to note here:

- On line (1), the constant `10` is an integer constant. The compiler will automatically convert it to a double before it is assigned to variable `d`.
- On line (2), the constant `1.015` is parsed as a double and inserted as the closest approximate value the floating point representation can afford.
- On line (3), the expression contains the multiplication operator. The compiler selects the operation that multiplies two `double` values, a different operation than the integer multiply.

3.4.1 Automatic type conversions

If an expression contains a mix of integer and floating point data types, the integer values will be converted to floating point first, and the data type of the expression will be floating point (usually `double`).

If an expression contains a mix of `float` and `double`, the `float` terms are converted to `double`, and that will also be the type of the expression.

3.4.2 Rounding

Due to the limited precision of the floating point representation, intermediate and final results will be rounded towards the true value. A floating point value should always be regarded as an approximation. In general, very small and very large numbers will suffer more from this lack of precision, but there will often be a small error present.

The performance of a particular program is ensured by confining the error below the specified threshold of significance. One particular instance which affects the programmer is the use of non-integer constants. Always use library-defined constants when they are available. If they are not, see if the desired value can be computed from them. When a constant is specified in the source code, *use as many digits as possible.* For example:

```
double PI_TWICE = 2.0 * Math.PI;

double MX_GAIN = 0.707106781; // incomplete
double MX_GAIN = Math.sqrt(2.0) / 2.0; // better
```

3.4.3 Special floating point values

The floating point number standard IEEE 754 specifies a few special values that are used to indicate a result that can not be managed normally.

Signed zero

Since the floating point format contains a sign bit separate from the mantissa and exponent, the values 0 and -0 are both valid (and equal).

Infinity

A result that exceeds what the floating point number can hold, is represented by the special values `Infinity` and `-Infinity`. You can continue to compute with an infinity, but the result mostly continues to be infinity, so in a long sequence of computations that ends in infinity, it is not obvious in which step the result stepped over the threshold.

Not a number

The special value `NaN` (Not a Number) is generated by attempts to compute expressions that are undefined in standard arithmethic, like $0/0$ or $\sqrt{-1}$. An expression that involves a `NaN` component will generally result in `NaN`, but rare exceptions from this may exist by virtue of functions that ignore it.

Examples

The following provides some examples on special values:

```
double inf = 1d / 0d;      // (1)
System.out.println(inf);   // prints Infinity
System.out.println(-inf); // prints -Infinity
double nan = inf * 0d;     // (2)
System.out.println(nan);   // prints NaN
System.out.println(Math.pow(nan, 0)); // (3) prints 1.0
```

Expression (1) divides one by zero, and the result is `Infinity`. The next expression successfully negates it to get `-Infinity`.

Expression (2) multiplies infinity with zero, but there is no viable definition for that available in standard arithmetic, so the result is not a number, or `NaN`.

Finally, expression (3) calls on the library method `Math.pow` which is exponentation, x^y. Now, by definition, $x^0 = 1$, so when the function sees that the exponent is zero, it does not matter what the base is, and it returns 1.

A result of `Infinity` or `NaN` indicate a problem, because the calculation breaks down. Debugging and range checking of inputs are recommended.

3.5 Operator precedence and parentheses

All things being equal, expressions are evaluated from left to right. However, many operators have different *precedence*[1] in order to provide an intuitive order of evaluation. In practice the following rules of thumb will often be sufficient when writing code:

- Unary operators are evaluated first
- Multiplication is evaluated before addition

In these rules, multiplication includes division and integer remainder, and addition includes subtraction.

For example, consider this expression:

```
2 + 3 * 4
```

Since multiplication is done before addition the evaluation steps will be:

```
2 + 3 * 4
      |
2 +  12
  |
 14
```

Now study the different result achieved by introducing parentheses:

```
(2 + 3) * 4
   |
   5    * 4
        |
       20
```

> It is always possible (and often recommended) to use parentheses and a sequence of expressions to explicitly control the order of evaluation.

3.6 Operations on a variable

Modifying the value of a variable is a *very* common step in a computer program. For this reason there exist a number of syntactic shortcuts in Java that provides direct modification of a variable in certain ways. These are expressed in the form of operators, but are actually compact ways to achieve something that could otherwise be done in the standard manner.

[1] https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html

Operator	Meaning
`+=`	add and assign
`-=`	subtract and assign
`*=`	multiply and assign
`/=`	divide and assign
`%=`	remainder and assign

Table 3.3: Assignment operators (incomplete)

	increment	decrement
prefix	`++v`	`--v`
postfix	`v++`	`v--`

Table 3.4: Increment and decrement operators

3.6.1 Assignment operators

A common computation is to assign a variable the value of an expression in which the variable's previous value is a part. For example:

```
x = x + k;  // x is incremented by k
a = a * g;  // a is modified by a factor g
```

To cater for this, *assignment operators* first evaluates their right-hand side expression, then takes the old value of the variable on the left-hand side, applies the specified operator, and finally stores the result back into the left-hand side variable:

```
x += k; // x is incremented by k
a *= g; // a is modified by a factor g
```

Table 3.3 displays the assignment operators used most often.

3.6.2 Increment and decrement operators

An even more restricted form of direct manipulation of a variable are the *increment* and *decrement* operators. As the names suggest, these add or subtract 1 from the variable's value. In addition to that, the variable still forms an expression in which its value is read (and perhaps used), and for this reason both operators have two forms: the *prefix* form in which the variable's value is modified *before* it is read, or the *postfix* form in which the variable's value is modified *after* it is read.

Table 3.4 shows all four increment and decrement operators. Here is a sequence of statements to demonstrate the operators:

```
int v = 0, u = 0;
```

```
u = ++v; // increment first, then read
// Equivalent code:
v = v + 1;
u = v;

u = --v; // decrement first, then read
// Equivalent code:
v = v - 1;
u = v;

u = v++; // read, then increment
// Equivalent code:
u = v;
v = v + 1;

u = v--; // read, then decrement
// Equivalent code:
u = v;
v = v - 1;
```

All possible ways to increment or decrement a variable:

```
int v = ...

v = v + 1;      // 1 is added to v
v += 1;         // 1 is added to v
v++;            // v is incremented by 1
++v;            // v is incremented by 1

v = v - 1;      // 1 is subtracted from v
v -= 1;         // 1 is subtracted from v
v--;            // v is decremented by 1
--v;            // v is decremented by 1
```

Mnemonic: the prefix operator is written *before* the variable, and updates the variable *before* its value is read.

The postfix operator is written *after* the variable, and updates the variable *after* its value has been read.

3.7 Comparing primitive values

In order to be able to *select* what to do, the computer must be able to make comparisons. There are basically two kinds of comparisons: *equality* and *less than.* When combined with each other and their negations, we arrive at the six operators listed in table 3.5.

Operator	Meaning	Usage	Logical equivalent
`==`	equal, identity	`a == b`	`!(a != b)`
`!=`	not equal	`a != b`	`!(a == b)`
`<`	less than	`a < b`	`!(a >= b)`
`<=`	less or equal	`a <= b`	`!(a > b)`
`>`	greater than	`a > b`	`!(a <= b)`
`>=`	greater or equal	`a >= b`	`!(a < b)`

Table 3.5: Numerical comparison operators

3.7.1 Equal ==

The equality operator is `==`. It takes two arguments of equal type (this is important) and then compares the *bit patterns* on both sides with each other. If the bit patterns are identical, the equality operator returns the boolean value `true`, otherwise it returns `false`[2]

The equality operator `==` can compare *all* possible values in Java, including reference types, but for those it will only return `true` when the two arguments refer to the same object. For this reason the `==` operator should not be used to compare the text of strings, because it cannot do that (see 2.4.2).

3.7.2 Not equal !=

The not equal operator `!=` is simply the negation of the equality operator. The operator is not strictly required, because we could write `!(a == b)` instead, but it provides a nice symmetry to the coding.

3.7.3 Numeric comparison operators

The operators `<`, `<=`, `>`, and `>=` compare *scalar values*, i.e. primitive values that can be ordered on a scale. The appropriate data types are `byte, char, short, int, long, float`, and `double`. These operators can not be applied to reference types, nor to boolean values, because those have no ordering.

3.7.4 Less than <

The expression `a < b` is `true` if the value `a` is smaller than the value `b`. Another good view is that `a` is left of `b` on the number line.

Here are two logical equivalents to the `<` operator:

[2]The operators `==` and `!=` regards the floating point values 0.0 and -0.0 as equal, even though their sign bits are different.

```
a < b         // a is less than b

b > a         // b is greater than a

!(a >= b)     // it is not the case that
              // a is greater or equal to b
```

3.7.5 Less or equal <=

The less or equal operator `<=` combines two alternative conditions into a single operator. The expression `a <= b` is `true` if the value `a` is less than the value `b`, *or* the values `a` and `b` are equal.

Here are some logical equivalents to the `<=` operator:

```
a <= b                // a is less or equal to b

(a < b) || (a == b)   // a is less or they are equal

b >= a                // b is greater or equal to a

!(a > b)              // it is not the case that
                      // a is greater than b
```

3.7.6 Greater than >

The expression `a > b` is `true` if the value `a` is greater than the value `b`. Another view is that `a` is right of `b` on the number line.

Here are two logical equivalents to the `> operator`:

```
a > b         // a is greater than b

b < a         // b is less than a

!(a <= b)     // it is not the case that
              // a is less or equal to b
```

3.7.7 Greater or equal >=

The greater or equal operator `>=` combines two alternative conditions into a single operator. The expression `a >= b` is `true` if the value `a` is greater than the value `b`, *or* the values `a` and `b` are equal.

Here are some logical equivalents to the `>=` operator:

```
a >= b                 // a is greater or equal to b

(a > b) || (a == b)    // a is greater or a and b are equal

b <= a                 // b is less than or equal to a

!(a < b)               // it is not the case that
                       // a is less than b
```

3.7.8 Equality of floating point numbers

When comparing floating point numbers like `float` and `double`, it is important to remember that the values stored in variables are approximations. In addition to that, they may have been subjected to different sequences of arithmetic, and may therefore be different in the least significant bits of their precision, even though their values are similar enough for the program. For this reason, when testing floating point numbers for equality, it is sometimes required to not test for equality, but instead test *if the difference between the numbers is small enough.* For example:

```
double d,e;        // two double variables

// computations happens here involving d and e

if (d == e)        // requires identical values
  ...

double diff = Math.abs(d - e);  // absolute difference
double t = ... ; // the greatest acceptable difference

if (diff <= t)     // t defines the limit
  ...
```

3.8 Logical operators

The numeric comparison operators `<, <=, >=, >` and the equality operators `==` and `!=` all return boolean values, i.e. `true` or `false`. In order to formulate more complicated logical expressions, operators are needed that operate on boolean values. There are three such operators: negation `!`, conjunction `&&`, and disjunction `||`.

3.8.1 Negation `!`

The logical negation operator `!` is unary, and negates the boolean value on its right side:

```
!true      // not true  == false
!false     // not false == true
```

3.8.2 Conjunction &&

The conjunction operator `&&` implements logical AND, i.e. both the left-hand side argument *and* the right-hand side argument must be `true` in order for the `&&` operator to be `true`:

```
true  && true   // true: both arguments are true
true  && false  // false
false && true   // false
false && false  // false
```

3.8.3 Disjunction ||

The disjunction operator `||` implements logical OR, i.e. the left-hand side argument is `true`, *or* the right-hand side argument is `true`, or both arguments are `true`, for the `||` operator to be `true`:

```
true  || true   // true: both arguments are true
true  || false  // true: left-hand side is true
false || true   // true: right-hand side is true
false || false  // false
```

3.9 Logical expressions

Logical expressions are used to express conditions that when evaluated are either `true` or `false`. Based on the result, the program can be directed to take or ignore a certain action, or choose between several alternative actions. Logical expressions occur in `if` and `if else` statements, in the conditions that keep loops like `for, while,` and `do while` going, in the `?` trinary operator, and in the assignment of boolean variables.

Logical expressions are composed from comparison operators, method calls (functions that return a boolean value), the logical operators for negation, conjunction, and disjunction, and parentheses. Here follows some example applications:

3.9.1 Print the larger value

Assume we have two variables, `a` and `b`, and we want to print the value of the larger one. We must select which print statement to execute:

```
int a = ...;
int b = ...;

if (a < b)                // if b is the greater number
  System.out.println(b); // print b
else                      // a is greater or equal to b
  System.out.println(a); // print a
```

3.9.2 Describe the numbers

Assume we have two variables `c` and `d`, and we want to examine them and print some properties of them. This is sometimes useful when debugging a program:

```
int c = ...;
int d = ...;

if (c < d)
  System.out.println ("c is smaller than d");
else if (c == d)
  System.out.println ("c and d are equal");
```

3.9.3 Home automation thermostat

The garage heater is only to be on in the winter, and then only if the temperature drops below 5 degrees Celsius.

```
double temperature = ...; // from thermometer
int month = ...;          // from calendar, 1-12

if (temperature < 5.0 && (month <= 3 || 11 <= month))
  // set heater on
else
  // set heater off
```

3.9.4 Mouse-click in region

Was the mouse clicked inside the region bounded by the coordinates `x`, `y` and `width`, `height`?

```
MouseEvent e = ...;

int mouseX = e.getX(); // Get click position
int mouseY = e.getY();

boolean insideHorizontally =
  (x <= mouseX) && (mouseX < (x + width));
```

```
boolean insideVertically =
  (y <= mouseY) && (mouseY < (y + height));

if (insideHorizontally && insideVertically)
  // mouse was clicked inside region
```

3.9.5 Leap-year detect

Leap-years occur according to the following rules:

- If the year is evenly divisible by 4 it is a leap-year,
- *except* if it also is evenly divisible by 100, in which case it is *not* a leap-year,
- *except* if it also is evenly divisible by 400, in which case it *is* a leap-year.

```
int year = ...;   // from somewhere

// Truth-table analysis
// by 4         by 100       by 400      leap-year
// ----------------------------------------------
// false          *            *         false
// true         false        true        -impossible-
//
// true         false        false       true
// true         true         false       false
// true         true         true        true


boolean by4   =   (year %   4) == 0;
boolean by100 =   (year % 100) == 0;
boolean by400 =   (year % 400) == 0;

if (by4 && (by100 == by400))
  ... // it is a leap-year
```

The truth-table analysis in this example examines the three available predicates: that the year is divisible by 4, 100, and 400. Since the first rule says that the year always must be divisible by 4, we know that all cases where the year is not divisible by 4 cannot be leap-years, and thus if `by4` is false, the whole condition is false. This gives us the idea that we can form a conjunction that requires that `by4` is true:

```
by4 && ...
```

We then turn to inspect the cases when `by4` is true, and the various combinations of `by100` and `by400`. We note that if the number is divisible by 400, then it must be also be divisible by 100, and thus we can eliminate one combination of truth values that is impossible, and will never happen.

We are now left with three viable combinations and we discover to our delight that the right side of the conjunction we have begun writing is true when `by100` and `by400` have the same truth value. We formulate a condition of equality:

```
by100 == by400
```

and insert it as the second condition in the main expression:

```
by4 && (by100 == by400)
```

3.9.6 Short-cut operators

This is an advanced topic and you may want to return here when methods and method calls are fully understood.

The operators `&&` and `||` are defined as *short-cut* operators. This means that they will stop evaluating their logical terms *as soon as the truth value of the operator is decided.* For example, assume we have a conjunction containing the two logical expressions p_1 and p_2:

p_1 `&&` p_2

The expression p_1 will be evaluated first. If it evaluates to `true`, then evaluation continues with p_2, and the value of that will determine the value of the `&&` operator.

However, if p_1 evaluates to `false`, then it is already clear that the value of the conjunction is going to be `false` too, because the `&&` operator requires both arguments to be `true`. There is simply no point in evaluating p_2, because it does not matter what value it will evaluate to. Thus, if p_1 is `false`, then p_2 is not evaluated.

For the disjunction operator `||` the case is similar. Given the expression:

q_1 `||` q_2

if q_1 evaluates to `true`, then the value of the `||` operator is already known, and q_2 is not evaluated. Only if q_1 evaluates to `false` is it necessary to evaluate q_2.

For expressions that only involve constants and unmodified variables, the effect of the short-cut operators is invisible and does not affect the program in any way, save for a very slight speed improvement. However, for expressions that involve *side-effect*, like modifying a variable or calling a method, that side-effect may or may not be performed depending on the evaluation of the whole expression. For example:

```
int a = 1;
int b = 2;
int c = 0;
```

```
if ((a == b) && (c++ == 0))
  ...
```

In this example, the variables `a` and `b` are tested for equality to each other, and variable `c` is compared against zero, and incremented (by the postfix `++` operator) after its value has been read.

The comparison `a == b` will evaluate to `false` because `a` and `b` have different values. Due to short-cut evaluation of the `&&` operator, it will not evaluate the right-hand side, and as a result variable `c` will *not be incremented.* The increment of `c` is has been conditioned on the values of `a` and `b`. This can lead to obscure bugs that are very hard to hunt down.

There are three things to take away from this:

- Write logical expression that are free of side-effect.
- Place conditionally executed code in `if` clauses.
- If side-effect is desired, the operators `&` and `|` can be used instead. They always evaluate both terms.

Chapter 4

Controlling program flow

4.1 Execution flow concepts

A computer programming language becomes powerful when it exhibits the capability to arrange the execution of statements according to the following concepts:

- *Sequence* — Statements are executed one after another, in a well-defined sequence
- *Selection* — Depending on the outcome of a test, some statements are optionally selected for execution
- *Iteration* — Statements can be executed repeatedly.

4.1.1 The statement

A *statement* in Java is the smallest unit of execution. However, due to the generality of the langauge, and the ability to compose long and complex expressions, a single statement can in itself contain the concepts of sequence, selection, and iteration. For the time being, let us assume that we are only dealing with uncomplicated statements.

An *atomic* statement is terminated by the semi-colon character `';'`. For example:

```
int x = 0;
...
distance = Math.sqrt(x*x + y*y);
...
v++;
```

The semi-colon tells the compiler that the end of the statement has been reached.

A *compound* statement defines a sequence of statements by surrounding them by the braces `'{'` and `'}'`. This is also called a *block statement*, or just *block*, for short. For example:

```
{
  x = x + 2;
  y = y + 2;
}
```

The following is important to understand regarding statements:

- Wherever a statement is allowed, a block statement is also allowed.
- The statements in a block can be a mix of atomic statements and block statements, nested to any level.
- Control constructs such as `if, if else, for, while, do while,` and `switch` are also statements.

4.1.2 Sequence

A block statement defines a unit, a set of statements that form a sequence.

The body of a method is a block statement. In particular, the body of the `main` method is the topmost sequence in the whole program.

Statements in a sequence are executed from the top towards the bottom. When the last statement of the sequence has finished executing, that sequence is ended. Consequently, when execution reaches the end of the `main` method the program ends.

4.1.3 Selection

Selection is the ability to optionally execute a statement (or block statement). A test is made to see if a condition holds, and then a statement is executed or not.

Selection lies at the very heart of *computability.* Without selection, computers would be no more than sliderules.

4.1.4 Iteration

Iteration is the ability to execute a statement (or block statement) several times, as long as some condition holds. Iteration makes it possible to repeat a task as many times as are needed. For example, count the number of characters in a string, print all lines in a file, read all records from a database, modify all samples in a sound, accept user commands until it is the quit command ... and so on.

Fun fact: The Spaghetti Monster

CPU instructions implement selection and iteration by making a conditional jump to the instruction (at some memory address) where execution is to continue. Early programming languages, in particular the infamous BASIC language, followed this model and required that each source code line in the program had its own number (address). Selection and iteration was then realized by using the `GOTO` statement, like so:

```
220 IF I < N THEN GOTO 150
```

In any non-trivial program, the use of `GOTO` soon created a mess of jumps all over the source code, and the description *spaghetti programming* was entirely justified.

Machines may be happy with jumps in the sequence of instructions, but humans are soon lost. In modern programming languages there is no use for `GOTO`; it has been weened out in favour of more general constructs.

4.2 Conditional statements

Conditional execution of statements occur when they are guarded by a test. Depending on the logical truth of the test, the guarded statements will be, or will not be, executed.

4.2.1 The if statement

The reserved word `if` begins a conditional statement. The syntax is:

```
if (c)
  s
```

which means: if condition c evaluates to `true`, then execute statement s. If condition c evaluates to `false`, then do not execute statement s. The parentheses around the condition are required, because they help the compiler to see the end of the condition.

In this example, s is the call to the `println` method, including the terminating semicolon `';'`:

```
if (a < 10)
  System.out.println("a is smaller than 10");
```

Remember that a block statement is also a statement. The `if` statement can therefore guard any number of statements:

```
if (b == 2) {
  x = 1;
  n = 2;
  System.out.println("Only two items left.");
}
```

4.2.2 The if else statement

The `if else` statement is used to select between two mutually exclusive statements. The condition is tested and one of the two statements is executed:

```
if (c)
  s_t
else
  s_f
```

If condition c evaluates to `true`, then statement s_t is executed, else statement s_f is executed. For example:

```
// Assign the smaller value to variable k
if (a < b)
  k = a;
else
  k = b;
```

4.2.3 Nested if else statements

The `if` and `if else` statements can be themselves be guarded by conditions. This can be used achieved more complicated logic. For example:

```
// Find smallest value of int variables m, n, o
// and assign that value to variable k
if (m < n)
  if (m < o)     // m < n
    k = m;       // m < n AND m < o
  else
    k = o;       // o <= m AND m < n
```

```
else if (n < o)
  k = n          // n <= m AND n < o
else
  k = o;         // o <= n AND n <= m
```

The `else` clause will attach to the nearest possible `if`. This means that sometimes a block statement is required for the desired control. Look closely at the following example. This is what the programmer intends:

```
if (a == 0)
    if (b == 1)
        System.out.println("a is 0 and b is 1");
else
    System.out.println("a is not zero");
```

But since the `else` clause will attach to the nearest possible `if`, this is what the compiler sees:

```
if (a == 0)
    if (b == 1)
      System.out.println("a is 0 and b is 1");
    else
      System.out.println("a is not zero");
```

The compiler sees an `if` statement that is guarding an `if else` statement, but this is not what the programmer wanted. The program now has a logic error. The programmer must use block statements to make it clear to the compiler which statement is the `if` statement, and which is the `if else` statement:

```
if (a == 0) {
    if (b == 1)
        System.out.println("a is 0 and b is 1");
}
else
    System.out.println("a is not zero");
```

Now the compiler first sees an `if else` statement in which s_t is a block statement that contains an `if` statement.

4.2.4 The operator ?

The trinary conditional operator `?` is not a statement, it is an *operator*, and its place is in expressions (that are part of statements). The syntax of the operator is:

c ? e_t : e_f ;

If the condition c is `true`, the value of the `?` operator is the value of the expression e_t, otherwise it is the value of the e_f expression. For example:

```
// Assign the smallest value of a and b to k
k = a < b ? a : b;
```

4.2.5 The switch case statement

The `switch case` statement is used to select a statement when there are more than two alternatives. The syntax is:

```
switch ($e$) {

case $k_1$ :
  $s_1$
  break;

case $k_2$ :
  $s_2$
  break;



case $k_n$ :
  $s_n$
  break;

default:
  $s_d$
  break;
}
```

The controlling expression e is evaluated and the result is compared against the `case` constants $k_1, k_2 \dots k_n$. When the first match $e = k_i$ is found, execution continues with the corresponding statement s_i. If no matching `case` can be found, execution continues at the `default` case, if it is present. When a `break` keyword is encountered, the `switch case` statement is done, and execution continues with the first statement after the `switch` statement.

The `switch case` statement is useful when there a small number of cases that are selected by an integer expression. For example:

```
 char c = ...;

 switch (c) {

case ’Q’:
   quitFlag = true;
   System.out.println("Quitting.");
   break;

case ’?’:
   printHelp();
   break;
case 'A':
   mode = APPEND_MODE;
   break;

case 'I':
   mode = INSERT_MODE;
   break;

default:
   System.err.println("Unknown command " + command);
   break;
 }
```

The logic of the above `switch case` statement can also be produced by a chain of `if else` statements, as demonstrated here:

```
  char c = ...;

  if (c == 'Q') {
    quitFlag = true;
    System.out.println("Quitting.");
  }
  else if (c == '?')
    printHelp();
  else if (c == 'A')
    mode = APPEND_MODE;
  else if (c == 'I')
    mode = INSERT_MODE;
  else
    System.err.println("Unknown command " + command);
```

However, the rules of the `switch case` are liberal enough to allow for `case` clauses without any statements in them. This makes it possible to create disjunctions, i.e. alternative matches for the same entry point:

```
  char c = ...;

  switch (c) {

 case 'Q':
 case 'q':
 case 'X':
 case 'X':
    quitFlag = true;
    System.out.println("Quitting.");
    break;

 case '?':
 case 'H':
 case 'h':
    printHelp();
    break;
```

```
case 'A':
case 'a':
   mode = APPEND_MODE;
   break;

case 'I':
case 'i':
   mode = INSERT_MODE;
   break;

default:
   System.err.println("Unknown command " + command);
   break;
 }
```

This would be equivalent to the following `if else` sequence:

```
char c = ...;

if (c == 'Q' || c == 'q' || c == 'X' || c == 'x') {
  quitFlag = true;
  System.out.println("Quitting.");
}
else if (c == '?' || c == 'H' || c == 'h')
  printHelp();
else if (c == 'A' || c == 'a')
  mode = APPEND_MODE;
else if (c == 'I' || c == 'i')
  mode = INSERT_MODE;
else
  System.err.println("Unknown command " + command);
```

Finally, under most circumstances each `case` entry point is terminated by a matching `break`. If the `break` is not present, execution will happily continue into the next `case`, until its reaches a `break` or the end of the `switch case` statement. For example:

```
int n = ...;
switch (n) {
case 3:
  System.out.println("three"); // no break, fall through
case 2:
  System.out.println("two");   // no break, fall through
case 1:
  System.out.println("one");
}
```

If this code is executed when `n == 3`, the following printout is generated:

```
three
two
one
```

When `n == 2` the output is:

```
two
one
```

When `n == 1` the output becomes:

```
one
```

The expression in the `switch` clause must evaluate to a scalar value that can be compared against the constants in the `case` clauses. The comparison is performed using the identity operator `==`. However, since Java 8 the compiler also recognises strings as `case` selectors and inserts the proper code to compare strings using the `String.equals` method. For example:

```
// From Java 8 - switch case with String
String s = ...;

switch (s) {

case "foo":
  ...
  break;

case "bar":
  ...
  break;

default:
  ...
  break;
}
```

Switch case remarks

The `switch case` statement is a generalized and structured version of a computed GOTO, a construct inherited from old programming languages such as BASIC and FORTRAN. The value from the `switch` expression results in a jump to the matching `case` constant which therefore acts as a label. The ending `break` then again cause a jump to the end of the whole `switch case` statement.

The `switch case` statement should be used judiciously, it is no silver bullet for value dependent selection of code. As the number of `case` clauses grow, the statement becomes ever more unwieldy to manage and overview. In addition to that, if `case` clauses are not mutually independent and exclusive, the cases easily become bloated with repeated code which makes them even harder to support.

Nested `switch case` statements are not recommended. Let the `case` call a method instead, a method that is written to solve its task in the most appropriate way.

For the above reasons, the `switch case` statement is optimal for a small number (less than 10) of cases. These cases are for the most part mutually exclusive of each other. When data dependent selection of code is required for many scattered values there are other strategies, like array-based schemes, binary trees, or hash-tables. The data structure is used to locate an object which can perform the required operation.

4.3 Iteration

Iteration is the concept of repeating a sequence of instructions until some criterion is met. The Java language contains three general iterative constructs: `for, while,` and `do while`, and a *for each* version of `for` that iterates over all elements in an array or collection.

4.3.1 while

The syntax of the `while` loop is simple:

```
while (c)
  s
```

While condition c evaluates to `true`, statement s is executed. The statement s can be an atomic statement, or a block statement.

From the simple nature of the `while` loop, it follows that when the flow of execution arrives at a `while` loop, condition c is evaluated first, and if at that point it evaluates to `false`, the loop will not be entered, and statement s will not be executed at all. Condition c must be true from the start, if the loop is to be executed.

Another observation is that the loop can only terminate if statement s at some point modifies the state of the program in such a way that condition c evaluates to `false`. There must be a link between what statement s does, and what condition c tests.

Here is an example of a `while` loop:

```
// Print a certain number of dots
int n = 0;                // nof dots printed so far
int nofDots = 10;         // nof dots to print

while (n < nofDots) {     // are we done?
  System.out.print('.'); // print one dot
  n++;                    // and increment the count
}
```

4.3.2 do while

The `do while` loop puts the condition last:

```
do
  s
while (c);
```

The `do while` loop is always entered, and statement *s* is executed at least once. Then condition *c* is evaluated and if it is `true` then statement *s* is evaluated again until *c* is `false`. For example:

```
char c = ' ';

do {
  c = readCommand();
  ...
} while (c != 'Q');
```

The `do while` loop is probably the least used, mostly because there are less situations where it is the natural choice.

Note that local variables declared in statement *s* are not available to condition *c*. The statement and the condition must communicate using variables or state outside of *s*.

4.3.3 for

The *for* loop is a version of the `while` loop with four syntactically distinguished components. The syntax looks like this:

```
for (init ; c ; update)
  s
```

The *init* section is a list of initialized local variables. Condition *c* is tested before each iteration. Section *update* is performed at the end of each iteration. For example:

```
// Print a certain number of dots
int nofDots = 10;

for (int n = 0; n < nofDots; n++)
  System.out.print('.');
```

The components of the `for` loop can be understood in terms of this `while` loop and how it relates to local variables:

```
{
  init
  while (c) {
        s
        update
    }
}
```

The `for` loop is perhaps the most often used loop, because it is versatile and offers a reasonable structure.

4.3.4 for : (for each)

An alternative form of the `for` loop iterates over the elements in an array or in a collection. To be precise, it can be used with any object that implements the interface `Iterable`. The syntax is:

```
for ( v : obj )
  s
```

Where *v* is a variable declaration of the element type provided by the object *obj*. The variable is assigned one element at a time from the object, and statement *s* is executed. Thus, statement *s* will be executed once for each element in *obj*, with variable *v* holding a copy of the element. For example:

```
int [] ar = ...; // an array of integer numbers

for (int number : ar)      // print all numbers in ar
  System.out.println(ar);
```

Assignments to the loop variable will not affect the data structure, because the loop variable only holds a copy of the element. When the loop variable is a reference type, however, it may be possible to follow the reference to the object instance and ask it to modify itself.

4.3.5 Nested loops

It is perfectly possible and sometimes necessary to place a loop within a loop. Consider, for example, when you need to visit each cell in a table of rows and columns. There would be an outer loop for each row, and an inner loop for each column in that row. For example:

```
int [][] mx = new int [7][4];  // a matrix of int

... // table mx is filled with data

// Count the number of zero values in mx
int nofZeros = 0;

for (int row = 0; row < mx.length; row++) {         // (1)
  for (int col = 0; col < mx[row].length; col++) { // (2)
```

```
      if (mx[row][col] == 0)                          // (3)
        nofZeros++;
    }
  }
```

It is important to realise that when loops are nested, an inner loop will be entered once for each iteration of the loop it is in. In the above example, the outer loop (1) will make 7 iterations. For each iteration the inner loop (2) will make 4 iterations. As a result, the innermost `if` statement and test (3) will be executed $7 * 4 = 28$ times.

4.3.6 break

The `break` keyword can be used to break out of a loop at any point. Execution continues immediately after the loop statement. For example:

```
while (c1) {
  ...            // do some work
  if (c2)    // are we done looping?
    break;       // yes
  ...            // no, do some more work
}
```

If condition c_2 is `true`, the `break` is reached and execution continues after the loop. In nested loops, only the loop that issues the `break` is exited. If the inner loop breaks, it is exited and the outer loop continues normally with its next iteration.

For clean, structured programming, one should strive to avoid using `break`, but sometimes it is just the perfect thing.

4.3.7 continue

The `continue` keyword can be used to immediately continue with the next iteration of the loop. It is useful when it is determined that there is no point in doing further work in this iteration, and the control logic is sufficiently complicated to prohibit a structured selection of statements. For example:

```
while (c1) {
  ...            // do some work
  if (c2)    // done with this iteration?
    continue;    // yes, start the next one
  ...            // no, do some more work
}
```

If condition c_2 is `true`, execution immediately jumps to the test of condition c_1 and continues with the next iteration.

4.3.8 break and continue remarks

The `break` and `continue` keywords are in fact structured programming versions of GOTOs, i.e. jumps in the instruction sequence. Correctly used, however, they provide cleaner code while maintaining high efficiency. Their use is rarely needed in trivial cases, and the programmer should carefully consider if there are equivalent choices of selection that does not require `break` or `continue`.

4.4 Examples

Here are a few examples using some of the constructs presented in this chapter. In order to have something to iterate over the examples are using arrays which are described in chapter 5.

4.4.1 Analyze lake water-level data

In the array `waterLevels` are real-valued measurements of the water level in a lake. The task is to find the average water level, the lowest and highest levels, and finally to present the data.

```
double [] waterLevels = ...; // from level meter log file

// Use the first value as the starting point
double sumOfLevels   = waterLevels[0];
double lowWaterMark  = waterLevels[0];
double highWaterMark = waterLevels[0];

// Then process the remaining values
for (int i = 1; i < waterLevels.length; i++) {

  sumOfLevels += waterLevels[i]; // add to sum

  if (waterLevels[i] < lowWaterMark) // check for lower
    lowWaterMark = waterLevels[i];

  if (highWaterMark < waterLevels[i]) // check for higher
    highWaterMark = waterLevels[i];
}

double averageLevel = sumOfLevels / waterLevels.length;

System.out.println("Average water level: " + averageLevel);
System.out.println("Low water mark: " + lowWaterMark);
System.out.println("High water mark: " + highWaterMark);
```

If the above code is applied to the water level readings

```
3.0, 3.3, 3.2, 2.9, 2.9, 2.8, 2.7
```

the following printout is generated:

```
Average water level: 2.9714285714285715
Low water mark: 2.7
High water mark: 3.3
```

4.4.2 Search for the first matching character

The `char` array `ca` is to be searched for a given character. If the array contains such a character, variable `found` is to be set to `true`, and `false` otherwise. The example consists of a common setup part, and several different ways of solving the same task. First the common code:

```
char [] ca = ...;   // array to search
char q = ...;       // character to search for

boolean found = false; // not found yet
```

Version 1, structured programming with a `for` loop:

```
for (int i = 0; i < ca.length && !found; i++)
  if (ca[i] == q)       // check this character
    found = true;       // found
```

The `for` loop will continue as long as the index `i` is less than the length of the array, *and* the element is not found.

Version 2, a `for` loop with a `break`:

```
  for (int i = 0; i < ca.length; i++) {
    if (ca[i] == q) {     // check this character
      found = true;       // found
      break;              // no need to search more
    }
}
```

The `for` loop will continue as long as the index `i` is less than the length of the arry, unless a match is found and `break` is used to terminate the loop prematurely.

Version 3 (advanced), a `while` loop with direct assignment and internal post-increment of the index:

```
int i = 0;
while (i < ca.length && !found)
  found =  ca[i++] == q;
```

The loop will continue as long as the index `i` is less than the length of the array *and* the element is not found. The boolean result of the test is directly assigned to the variable `found`, and that works because the loop should end on the first match. The index variable `i` is incremented by the post-increment operator *after* its value has been read, and that works because it is the last time `i` is read in the loop body.

Version 4, a `for` each loop with a `break`:

```
for (char c : ca)
  if (c == q) {
    found = true;
    break;
  }
```

The `for` each loop will iterate over all elements in the array, so here a `break` must be used to escape the loop prematurely. There is also no need no maintain and check an index, because that is provided by the `for` each loop.

4.4.3 Find first and last characters

The `char` array `ca` is to be searched for a matching character `q`. The integer variables `firstIndex` and `lastIndex` should be set to the indices of the first and last matches, or -1 if `q` is not found at all.

```
char [] ca = ...;
char q = ...;

int firstIndex = -1; // assume not found
int lastIndex  = -1; // assume not found

// Search from the beginning towards higher indices
for (int i = 0; i < ca.length && firstIndex == -1; i++)
  if (ca[i] == q)
    firstIndex = i;

// Only attempt to find the last match if there was
// a first match
if (-1 < firstMatch)
  for (int i = ca.length - 1; lastIndex == -1; i--)
    if (ca[i] == q)
      lastIndex = i;
```

The search for the first match is done from index 0 towards higher indices. As soon as a match is found, the current value of `i` is copied to `firstIndex`, and by virtue of the loop condition, the loop ends.

The search for the last match is only attempted if a first match has been found. If no first match exists, we know that we will not find a match the second time. If a first match exists, we are guaranteed to find at least that one, so we start from the last legal index (`ca.length - 1`) and decrement `i` towards 0.

4.4.4 Count all odd numbers

In the array `numbers` are integer numbers. The task is to count how many odd numbers there are.

```
int [] numbers = ...;
int nofOddNumbers = 0;

for (int n : numbers)
  if (n % 2 == 1)
    nofOddNumbers++;
```

Chapter 5

Arrays

The *array* data structure is a way of allocating and using memory so that you have *n elements*, each of the same *element type.* The elements in the array are accessed by giving their integer *index*, first index always being 0 and the last index always being $n - 1$ (usually referred to as $length - 1$).

The array is useful when:

- several variables of the same data type are needed as a unit
- the name of each variable is irrelevant, and
- the location of each variable in the unit can be computed.

When using arrays, each individual variable in the array is referred to as an *element.* It is one of several identical elements, each of which can be assigned, read, and updated. The element is found by its index.

For example, to create an array of 200 `double`:

```
double [] d = new double[200];
```

Analyzing this declaration statement, we find these components:

datatype `d` `=` *initialization* `;`

A variable named `d` is declared, and initialized to its first value.

The data type of variable `d` is:

```
double []
```

which the compiler reads as *double, array of*, but humans read it as *array of double* because that makes for better English. This is an *array data type.*

The initial value of variable `d` is then presented on the right-hand side of the assignment operator `=`. The expression that initializes `d` is:

```
new double [200]
```

The `new` operator is used to create objects. We are asking for a whole chunk of memory, big enough to hold 200 doubles. Since Java is an object-oriented language, each array that is created in Java is managed by an object. That object contains the memory for the elements and the properties of the array.

When the compiler reads the initializing expression, it will see: *create a new object, of type double, array of, with 200 elements.* The compiler generates the appropriate code to accomplish this at runtime, and to *assign the reference to the new array object to variable* `d`.

So, when the whole declaration is complete, two separate entities have come into being. We have an array type variable `d`, that can refer to arrays with doubles in them. We also have created a new array with 200 doubles in it, and variable `d` is now referring to it.

The array data type of variable `d` is `double []`. The size is not specified. An array data type is also a *reference type.* Variable `d` can refer to any array object, as long as that array object has `double` as its element type.

Here is a different example, an array of 64 bytes:

```
byte [] b = new byte[64];
```

The variable `b` is initialized to refer to an array of 64 elements. Each element in this array is a `byte`, so the element type is `byte`. Variable `b`, on the other hand, has the type *array of byte.* It can refer to byte array objects, and the length of the particular array `b` is referring to does not matter, as long as it is an array where the element type is `byte`.

To use an array, we must have a reference to the array object, and then employ the square brackets `'['` and `']'`. Inside the brackets we put an expression that evaluates to the *integer index* of the element we want to access. For example:

```
double [] d = new double[200];

d[0] = 2.34;    // put the value 2.34 in the first element
d[1] = 9.0;     // put 9.0 in the second element
...
System.out.println(d[2]); // print the third element
```

The combination of a reference to an array object, followed by the square brackets and index expression, is what instructs the compiler. The compiler then generates code that follows the reference to the array object, locates the element by its index, and then performs the requested kind of access, read or assign.

```
variable   refers to              array object
   d -------------------> [length [ ][ ][ ][ ] ... [ ]]
                                   0  1  2  3    length-1
                                         indices
```

Figure 5.1: A one-dimensional array

5.1 One-dimensional arrays

A one-dimensional array is essentially a long box with numbered compartments in it. The first compartment has index 0 and contains the first element. The second compartment has index 1, and contains the second element. The last compartment has index *length* − 1 and contains the last element.

The array object itself has no intrinsic identity other than its reference. An *array data type* variable must therefore be used to hold the reference to the array object. The combination of the reference variable and the array object is what we use to work with arrays.

Figure 5.1 shows one possible way to conceptualize the combination of array variable, reference, and array object:

5.1.1 Index

In Java, the first element always has index 0, and the last element has index *length* − 1. The indices never change position.

An attempt to use a negative index will result in a runtime exception, the `ArrayIndexOutOfBoundsException`. So will an attempt to use an index in excess of *length* − 1.

The index is an integer expression. It must evaluate to an `int`. It is not possible to access more than one element at a time.

5.1.2 Creating and using an array

To create an array, use the `new` operator, specify the element type, and then the number of elements inside square brackets. In all cases, the reference to the array object that is returned by the `new` operator is assigned to a reference variable. This is not a language requirement, but a practical consideration. It serves no purpose to create an array object and then not keep the reference. For example:

```
double [] d = new double [200];
```

The array type variable used to refer to the array object, must be declared to have the appropriate array type. If it has a different type, it may not be able to refer to the array object, and it will not be possible to use it as an array in expressions.

It is, however, quite common to specify the array type variable first, and then the array object itself later. This happens when we know that we need the array variable, but we will not know until runtime exactly how many elements that are needed in the array object:

```
double [] d = null;     // initially d refers to nothing

...                     // the program receives information

int nofElements = ...; // the length of the array
                        // becomes known

d = new double[nofElements]; // an array object of the
                             // correct length is created
```

5.1.3 Array length

The array object has a publicly available, read-only `int` variable called `length`. By following the reference to the array object, and using the access operator `'.'`, the `length` variable can be read. For example:

```
d.length
```

Reading the length of an array object is often necessary, in particular for pieces of code where we do not know how long the array is until the program is running. By reading the length of the array, we can write general code, that can perform its job on arrays of all lengths.

The following example shows a piece of code that will print all the elements in an array, regardless of what length it happens to have:

```
int [] numbers = ...;   // refers to some array object

for (int i = 0; i < numbers.length; i++) {
  System.out.println(numbers[i]);
}
```

The `for` loop starts with `i` being 0, which is the index of the first element. Then it uses array indexing (`numbers[i]`) to read the `i`th element and print it. Finally, `i` is incremented and the loop condition is tested again.

The `for` loop stops when the loop variable `i` no longer is less than the length of the array referenced by variable `numbers`. This means that the last element printed will be `i == numbers.length - 1`, which is the index of the last element. So all elements will be printed.

The length of an array object can not be changed. The length is set when the array is created, and it remains the same throughout the lifetime of the array object.

It is possible to create an array with zero length:

```
double [] d = new double[0];
```

This is both legal and valid. The array object will be created, but since its length is zero, it cannot store any values. It has no elements to store values in.

Creating a zero-length array is sometimes convenient if you expect to replace the reference with a better array later, and you do not want your array variable (`d` in this case) to be `null`.

When you examine the printing example above, you can see that it works as expected with a zero-length array. It prints nothing. On the other hand, if the reference `numbers` is `null`, then the printing example will attempt to read the `length` of an object which is not there, and the program will crash with a `NullPointerException`.

5.1.4 Arrays are objects

Arrays are objects in the Java language. Just like every other object, arrays are created with `new` operator and can be accessed with the access operator `'.'`.

In difference to other objects, however, arrays are so ubiquitously used in programming, that the syntax of the language allows for certain shortcuts when it comes to arrays. One such shortcut is the use of square brackets `'['`, `']'` to specify which element to access. Another such shortcut concerns the static initialization of arrays.

Since big arrays can potentially consume a lot of memory, it is always worth thinking about the size of an array, and how long it needs to be in the program. A large, temporary array, for example, should be released to garbage collection when it is no longer needed. You release an object by removing *all* references to it from the program. When no variable points to an object, then the program can no longer access that object, and the object is eligible be recycled by the garbage collector.

5.1.5 The array variable is a reference type

A reference to an array has the general type `T []`, where `T` stands for some type, and `[]` indicates that it is an array. The type `double []`, for example, is a reference type to an array of `double`.

A variable declared to be of a reference type can refer to objects of that type. A variable declared:

```
long [] myArray
```

can refer to arrays of `long`, and only to arrays of long. A variable declared:

```
byte [] myArray
```

can refer to arrays of `byte`, and only to arrays of byte. The type of the reference variable must match the object.

A particular array object instance has a certain length, but the length is *not* part of the type. These arrays are all of the same array type, the type `byte []`, and the code here is valid (but only meaningful to demonstrate the point):

```
byte [] ba = null;

ba = new byte [18000];
ba = new byte [4711];
ba = new byte [2];
ba = new byte [0];
ba = null;
```

The array type variable is always a reference type, while the element type of an array can be *any* type. This is a subtle but important point. Do not confuse the two:

```
int [] ia = new int [1024];  // ia is a reference type

ia[0] = -77;                 // ia[index] is an int
```

5.1.6 Static initialization of arrays

Sometimes the contents of an array is known already when the source code for the program is being written. Let us assume that we want an array of the first seven prime numbers. We could do like this:

```
int [] primes = new int[7];

primes[0] = 2;
primes[1] = 3;
primes[2] = 5;
primes[3] = 7;
primes[4] = 11;
primes[5] = 13;
primes[6] = 17;
```

This works, there is nothing wrong with it. But there is a lot of typing involved, and when there is a lot of typing programmers sometimes resort to copying, and when you copy code you are apt to make mistakes. So there is an easier way to accomplish the same thing:

```
int [] primes = {2, 3, 5, 7, 11, 13, 17};
```

An array that is initialized in this manner is recognized by the compiler for what it is, because the type declaration on the left-hand side of the initialization says: *int, array of.* Thus, when the compiler comes to the right-hand side, and sees a comma-separated list of values, it knows what it must do. It must count the number of elements in that list, generate code to call the `new` operator with that amount, and finally generate assignments to each of the elements with the given constants. Much like we did explicitly above, except this time it is the compiler doing it for us.

The story gets better though. Each element in the initialization list is actually an *expression.* That means that we can specify expressions that are to be calculated, and then inserted into the new array. These are standard expressions, except they cannot refer to the array that is being initialized. Such self-reference is not possible because the array has not been created yet when the expression is evaluated. For example:

```
double value = ...; // some value

double [] fractions = {value/1, value/2, value/3, value/4};
```

Finally, there is even the possibility of creating a statically initialized array on the fly, for example as an argument to a method call, or the assignment to a variable declared previously. In these cases, for various deep reasons concerning type safety, the compiler will not infer the type of the array from the context. We must add code that explicitly invokes the `new` operator and tells the type of the array.

```
double [] d = ...;

...

d = new double [] {-1.0, -0.7, 0.1, 0.2, 0.22};
```

5.1.7 for and for each loops

The `for` loop is an excellent tool to explore the contents of an array. You can always use a `for` loop:

```
double [] d = ...;

for (int i = 0; i < d.length; i++) {
  ... // process each element
}
```

Sometimes we want the loop to begin with the second element. The standard `for` loop can accomplish this easily:

```
// Find maximum value
double [] d = ...;

double max = d[0]; // the first value is our first candidate

// examine the rest of the candidates
for (int i = 1; i < d.length; i++) {
  if (max < d[i])
    max = d[i];
}
```

When we want to read (but not write) all elements in an array without thinking about indices, the `for` each loop is easy to use:

```
double [] da = ...;

for (double v : da) {
  System.out.println(v);
}
```

5.2 Multi-dimensional arrays

A two-dimensional array is similar to a matrix of elements, much like in a spreadsheet program, the squares on a piece of graph paper, or a table of data with rows and columns. A three-dimensional array is like a stack of graph paper. A four-dimensional array is like a shelf with a row of separate stacks of graph paper. A five-dimensional array is like a corridor of rooms, and in each room there is a shelf with a row of stacks of graph paper. A six-dimensional array is like a street with houses, and in each house there is a corridor... Higher dimensions defy the imagination but generalizes in the same way.

It is rare to use more than three dimensions in arrays, because not many problems have that kind of multi-dimensional data, and when they do the resulting data set often has holes in it, which means that a full three-dimensional array may waste a lot of memory just to satisfy some portions of the data structure. Still, the Java language allows for any dimensionality to be specified.

The best way to think of a multi-dimensional array in Java, is that of a *tree* data structure. A data tree is upside down from natural trees. The data tree has its root at the top and the leaves at the bottom.

At the root of a multi-dimensional array is an one-dimensional array that implements the first dimension. The element type of this array is *reference to an array with one less dimension than its own array type.* Thus, in order to find an element, you pick an index, access the root and follow that reference found

```
  mx
  | refers to
  V
 ---           0     1     2     3
0[ ]-------->[[    ][    ][    ][    ]]
1[ ]-------->[[    ][    ][6.99][    ]]
2[ ]-------->[[    ][    ][    ][    ]]
 ---
```

Figure 5.2: A two-dimensional array

in the element. Now you come to the next level down. Here is another one-dimensional array where the elements also are of the type *reference to an array with one less dimension than its own array type*. Finally, when you approach the leaves of the tree, you find a reference to a one-dimensional array, where the element data type is that of the data you actually want to store.

Luckily, in our everyday programming the data tree model is not something we need to consider. It is only needed to understand how Java implements multi-dimensional arrays.

5.2.1 Matrices — two-dimensional arrays

A two-dimensional array, or matrix, is declared in the following way:

```
double [][] mx
```

This tells the compiler that variable `mx`, is of type *double, array of, array of*, or in English, *an array of array of double*. What this actually means, is that `mx` can refer to an array object, in which the *element* type is *array of double*. The element type is a reference type, and each element refers to a one-dimensional array.

In fact, Java has only one-dimensional arrays, but they are cleverly combined in order to create the very strong illusion of multi-dimensional arrays.

Here is a concrete example, a small matrix of three rows and four columns. In each matrix cell a `double` is stored:

```
double [][] mx = new double [3][4];
```

The declaration instructs the compiler to generate code that creates the data structure shown in figure 5.2. The variable `mx` refers to a one-dimensional array, in which each element in turn refers to a one-dimensional array of `double`. The value `6.99` has been inserted into the cell located at row 1, column 2. That was done by the assignment:

```
mx[1][2] = 6.99;
```

```
    mx
    | refers to
    V
   ---
  0[ ]
  1[ ]
  2[ ]
   ---
```

Figure 5.3: An incompletely filled matrix

To read the value back out again, use the same expression in a reading context:

```
double v = ...;
...
v = mx[1][2];
```

To visit every cell in the matrix, nested `for` loops are needed:

```
// Find maximum value
double [][] mx = ...;

double max = mx[0][0];

for (int row = 0; row < mx.length; row++) {
  for (int col = 0; col < mx[row].length; col++) // (1)
   if (max < mx[row][col])
     max = mx[row][col];
}
```

There is one important thing to notice in this example, and that is the loop condition on the inner loop (1). The loop condition looks up the length of the array that represents the current row, in order to see how many elements (columns) are in the row. The reason for this check, is that it is a convenient way to process any two-dimensional array, without running the risk of landing an `ArrayIndexOutOfBoundsException`.

To further emphasize this point, consider the following code:

```
double [][] mx = new double [3][];
```

This code will initialize `mx` to refer to the data structure shown in figure 5.3.

The elements in the array referred to by `mx` are all `null`. They are referring to nothing. To change that, we can explicitly create arrays of `double` and assign their references, just like we did with one-dimensional arrays:

```
mx[0] = new double[2];
mx[1] = new double[5];
mx[2] = new double[3];
```

```
    mx
    | refers to
    V
   ---            0     1     2     3     4
  0[ ]-------->[[    ][    ]]
  1[ ]-------->[[    ][    ][    ][    ][    ]]
  2[ ]-------->[[    ][    ][    ]]
   ---
```

Figure 5.4: A matrix with different row lengths

The whole data structure referred to by `mx` now looks like in figure 5.4.

As you can see in figure 5.4, the rows are now of different lengths, and this is a perfectly legal two-dimensional array. And, since our maximum value code above *checks* for the length of each row, it will process this matrix correctly, and it will also work in the special case where all rows have the same length.

5.2.2 Cubes

Three-dimensional arrays (and higher dimensions) continue by adding levels of indexing. A two-dimensional array has two indices. A three-dimensional array has three indices. To reach the element data type at the bottom of the data structure *all indices must be specified.*

A three by three cube is easily imagined if one thinks of the toy *Rubic's Cube.* It has three planes, and each plane consists of three by three cells. In total 27 cells. A similar array of integer ands can be declared like this:

```
  int [][][] cube = new int[3][3][3];
```

Figure 5.5 is a conceptual visualization of the corresponding objects and references. In the figure, `r` means a reference, and i means an `int`. Indices go from left to right, and top to bottom. Cell `mx[2][0][1]` is indicated with an asterisk (*).

The tree data structure is quite clear from figure 5.5. What also should be clear, is that just as was the case with two-dimensional arrays, there is no requirement that all arrays in the multi-dimensional array are of the same length. The may even have length zero, or a reference can be `null`. Such degenerate cases should be avoided, if possible. But they are legal and valid.

A final consideration when creating arrays of three dimensions and higher, is how fast memory is consumed. For example, assume you decide to order up something like this:

```
  double [][][] dc = new double [10][10][10];
```

```
                            mx
                            |
                            V
                        ---------
                       [[r][r][r]]
                         |  |  |
      +------------------+  |  +-------------------+
      |                     |                      |
      V                     V                      V
 -----------           -----------            -----------
[[r] [r] [r]]         [[r] [r] [r]]          [[r] [r] [r]]
  |   |   |             |   |   |              |   |   |
  V   V   V             V   V   V              V   V   V
 --- --- ---           --- --- ---            --- --- ---
 [i] [i] [i]           [i] [i] [i]            [i] [i] [i]
 [i] [i] [i]           [i] [i] [i]           *[i] [i] [i]
 [i] [i] [i]           [i] [i] [i]            [i] [i] [i]
```

Figure 5.5: A three-dimensional array

Ten elements in each level is not much. You will be creating 1 + 10 = 11 array objects with 10 references in each, and 100 array objects with 10 `double` elements in each. Assuming the reference takes four bytes, the complete tally is:

$$(10 + 10 * 10) * 4 + (10 * 10 * 10) * 8 = 8\,440 \text{ bytes} \tag{5.1}$$

That is not much, barely over 8 Kbyte. So let us double the size of the dimensions:

```
double [][][] dc = new double [20][20][20];
```

That will require 1 + 20 = 21 array objects with 20 references in each, and 400 array objects with 20 `double` elements in each:

$$(20 + 20 * 20) * 4 + (20 * 20 * 20) * 8 = 65\,680 \text{ bytes} \tag{5.2}$$

Only slightly over 64 Kbyte, nothing to worry about. However, the point is that the relation between the number of elements and the resulting memory requirement is *cubic*.

5.2.3 Static initialization of multi-dimensional arrays

The static initialization of a multi-dimensional array follows the same pattern as that for a one-dimensional array, a list of elements in braces `'{', '}'`. For an array with more than one dimension, the elements of the list will be other one-dimensional arrays. Thus, the whole initialization consists of lists of lists. To initialize a three by four `int` matrix, we can write:

```
int [][] mx = { {1,  2,  3,  4},
                {5,  6,  7,  8},
                {9, 10, 11, 12} };
```

```
  mx
   |
   V
  ---          0   1   2   3
 0[ ]-------->[[ 1][ 2][ 3][ 4]]
 1[ ]-------->[[ 5][ 6][ 7][ 8]]
 2[ ]-------->[[ 9][10][11][12]]
  ---
```

Figure 5.6: The initialized 3 by 4 matrix

Figure 5.6 shows the resulting array data structure.

Higher dimension arrays can be initialized similarly.

5.3 Element data type and array data type

Two important concepts should be fully understood before tackling arrays in any serious manner. They are *element data type* and *array data type.* They have both been covered extensively in this chapter, and here is the summary:

- The element data type is the data type of the array elements. It is the data type of the expression `a[i]`, where `a` is some array and `[i]` is the access to one of its elements. All elements in an array have the same data type, the element data type.
- The array data type is the type of the array object that implements an array. It is always a reference type. A variable of a certain array data type can refer to different array objects, at different times during the execution of a program, provided that they are of the corresponding array data type. The variable may also refer to nothing, i.e. be `null`.

Chapter 6

Methods

A program should be structured in managable and logical units of work. As the program goes through its runtime life-cycle, it will generally advance through these states:

1. initialize
2. collect input
3. select and perform operations
4. generate output
5. clean up and exit.

A loop involving states 2,3,4 is not uncommon for interactive programs.

Each of these states, if sufficiently independent, should be sectioned off to a separate part of the program. The operations available in step 3 could be implemented independently of each other, in separate sections of the source code.

For the programmer it is convenient to be able to create such sections, and give them names, so that they can be called upon when needed, and as many times as they are needed. This is the concept of a *subprogram*, a routine that is invoked by higher level code. The caller sees the subprogram as a procedure or function, available to perform a defined subtask. The subprogram is the implementation of that subtask.

In Java, subprograms (routines, procedures, functions) are collectively called *methods*. With Java being an object-oriented programming language, methods can only exist within classes.

Similar to variables, methods can be declared to be instance methods or static methods. Instance methods belong to an instance of a class, and can manipulate the variables in that instance. Instance methods can only be called via a reference to an object instance.

Static methods belong to the *definition* of a class. They do not require an instance to be called; they are always available. In particular, the `main` method which is the top-most method in a Java program, must be `static` because it must be callable before any object instances have been created in the program.

The careful arrangement of indepedent and efficient methods is a vital part of program structure design, and a key property to extract maximum performance out of the computer, and the people who are set to maintain the software.

6.1 Creating and using methods

Methods are the building blocks of a program, and each method is created for a particular purpose. It can be a pure mathematical function that is generally useful, like `Math.sqrt`, or it can be a routine that performs a specific modification of the object to which it belongs, like `StringBuilder.append`.

Each method should be designed with its role and purpose in mind. A method that performs a well-defined and understandable operation is often a better building block than a large and complicated method that tries to do too much at the same time.

6.1.1 The concept of a subprogram

A subprogram is a named unit of code that performs a well-defined task, usually with one or more parameters. One example of this concept is the function `java.lang.Math.sqrt(double d)`, which computes an approximation of $\sqrt{n}$ for `double` values. The function returns its result as a `double` value. It can be called from any program (because it is part of the standard library) like so:

```
double u = Math.sqrt(2.0);
```

The only purpose of the method is to compute the square root of a number, and that is what it does. It is both general and useful, and it can be called from many different programs.

6.1.2 Method syntax

The syntax for declaring a method looks like this:

1. Access modifier: `private`, `protected`, or `public`. An absent modifier gives the default access.
2. `static` declaration. When absent the method is an instance method.
3. `final` declaration. When present the method cannot be overridden by a subclass.
4. A list of generic type parameters (often omitted)
5. The return value data type, or `void` for no return value.
6. *method name*
7. `(` *parameter list* `)`
8. *method body*

The access modifier determines how other classes are allowed to call the method. A `private` method can only be called from within the class in which it is declared. A method with no explicit access modifier receives the default, which is that it can only be called by code in the declaring class or from classes in the same package. A `protected` method can be called from the declaring class, the declaring package, and subclasses to the declaring class. A `public` method can be called from anywhere.

The `static` declaration makes the method static, i.e. part of the class definition. With no `static` declaration, the method becomes an instance method.

The `final` declaration is rarely used on a method. When present it declares that subclasses are not allowed to override the method with their own versions of it.

A list of generic type parameters is used to generalize the type of the parameters. In most cases this is not used, and it will not be discussed further until chapter 19.

The return value data type is what the method returns. If the method returns an `int` value, then the return value data type is `int`. If the method returns an array, then the type is the corresponding array data type. If the method does not return any value, then the return data type is `void`.

The method name is a standard Java identifier. By convention, method names begin with a lowercase letter.

The parameter list is placed between parentheses. If the method does not accept any parameters, the parentheses are empty: `()`. The parameter list is a comma-separated list of local variable declarations. For example:

```
(char c, String s)
```

When the method is called, the parameter variables are assigned the actual values for that particular method call.

The method body is a block statement. Inside the body static variables and the parameter list variables are available. If the method is an instance method (not `static`) instance variables are also available. Additional local variables can be declared as needed.

If the method returns a value, the method ends by executing a `return` statement, with an expression that matches the declared return type. For example

```
return 0.0;
```

If the method does not return a value, the method ends when the end of the method body is reached, or, when a `return` statement with no expression is executed:

```
return;
```

Here are some sample method declarations with bodies not filled in:

```
private void reset () {
  ...
}

public static area (double side1, double side2) {
  ...
}

boolean isLeapYear (int year) {
  ...
}
```

6.1.3 Method signature

The method's name and parameter list defines the *signature* of the method. Only one method of a particular signature can exist in a class. However, methods with the same signature can exist in *different* classes.

Also, if class `b` extends (inherits from) class `a`, and class `a` has a non-final method *m* with signature *s*, and class `b` also declares a method *m* with signature *s*, then a call to method *m* on an instance of class `b` will go to the method in `b`. This is known as *method overriding* and is very useful to implement the concept of *polymorphism* in object-oriented programming.

When declaring `interface` classes, the method signatures are used. They are essentially method declarations without bodies:

```
boolean isLeapYear (int year);
```

Method signatures are critical to the compiler so that it can understand which method to choose when it encounters a call in the source code. Since the parameter list is part of the signature, it is sometimes convenient to have methods that have the same name but different parameter lists. This is used, for example, in the `print` and `println` methods:

```
  public void print(int n)
  public void print(char c)
  public void print(String s)
  ...
```

The programmer only needs to remember one name, and the the compiler will pick the right method to call from the actual parameter.

6.1.4 The `underline` method example

In the following paragraphs we will develop the `underline` method. This method accepts a string as parameter, it prints the string, and then prints a line of characters to underline the given string. The output could perhaps look like this:

```
Hello proud world!
--------------------------
```

For simplicity's sake, we will make the `underline` method `static`, and assume that it is placed in the definition of some appropriate class. This is the first version of the method:

```
  public static void underline (String s) {
    System.out.println (s);
    System.out.println ("---------------------");
  }
```

The first version of the method has the merit of simplicity, but perhaps it is too simple and inflexible. Look at some sample output:

```
Hello proud world!
---------------------
Where is thy compassionate mood hiding?
---------------------
```

It is obvious from the printout, that all underlines have the same length, while the text can have any length. We would rather have an underline that matched the length of the argument string. We can do that, by printing one dash (`'-'`) for each character in the given string `s`:

```
  public static void underline (String s) {
    System.out.println (s);
    for (int i = 0; i < s.length(); i++)
      System.out.print ('-');
    System.out.println();
  }
```

The output now becomes:

```
Hello proud world!
------------------
Where is thy compassionate mood hiding?
---------------------------------------
```

This looks much better. The underlining now adapts to the parameter string. The method has become more flexible.

6.1.5 Parameters

When a method call is made, the arguments to the method are matched against the method parameter list, on a type-by-type basis. Using the method's signature, the compiler has already decided which method to call. At runtime, the expressions in the argument list are first evaluated, and the result is assigned to the parameter variables in the method. For example, assume we have the method `fill`, which fills every cell in an `int` array with a given value:

```
  public static void fill (int [] ar, int v) {
    for (int i = 0; i < ar.length; i++)
      ar[i] = v;
  }
```

The method `fill` visits every element of the given array and assigns the value in parameter `v` to it.

In use, a call to `fill` could look like this:

```
  int [] counts = new int[32];
  int [] cards = new int[5];
  ...
  fill (counts, 0); // (1)
  fill (cards, -1); // (2)
  ...               // (3)
```

During compilation the compiler checks that the type of the arguments matches the type of the method parameters. If they do not match, or if a narrowing type conversion is required, there will be a compilation error.

At runtime, when the first call to `fill` is reached (1), several things happen. The first is that the position of the next instruction after the method call (2) is saved on the call stack. Then space for the method's local variables (including the parameter variables) is allocated, also from the call stack. The value of the arguments are assigned to the parameter variables. The parameter variable `ar` is assigned the array object reference in `counts`, and variable `v` is assigned the constant value `0`.

With setup complete, execution is now transferred to the body of the method, and it executes with the provided variable assignments. Since the return type is void, there is no value to return, so execution ends when the end of the method body is reached. The space for the method's local variables is reclaimed (popped off the call stack), The return address (2) is popped from the call stack, and execution continues from (2). Here is a second call to `fill`, so the whole thing is repeated, but this time with other parameter values and a different return address (3).

This might all seem technical and complicated, but the good news is that there is no need to memorize it. The important thing to remember, is that each call to a method is a new call. The parameter variables and any other local variables declared by the method only exist for the duration of the method call. In addition, they are unique for that call and cannot be modified or accessed by code outside of the method body.

A second important fact is that the parameters are *copies* of the arguments provided by the caller. This means that primitive data type values cannot be modified by the method. It has received its own copies to work with. References are also provided as copies. But references allows the method to follow the reference to the object (such as the array in the `fill` method example), and if the object allows it, modify it.

Parameter variables can optionally be declared to be `final`. This is an indication to the compiler that it should check that the parameter variables are not reassigned as the method executes. Such a reassignment overwrites the parameter value which is then lost to the method. It is generally good programming practice to always view parameter variables as being `final` (i.e. read-only) and instead declare new local variables as needed.

6.1.6 The function concept

A method with no return value is only useful if it does some work, like printing, writing to a file, or manipulating class variables. It must have some *side-effect.* A method with no return value is sometimes also known as a *procedure*, to distinguish it from functions.

A function, on the other hand, is a method that returns a value. This value can be inspected, or used in expressions, like the `Math.sqrt(double d)` function. A *pure* function has no noticeable side-effect; it can be called again and again without anything actually changing. The only thing that happens is that a value is computed and returned.

Hybrid functions both return a value and has some side-effect. A typical example would be the `new` operator which creates a new object instance and returns a reference to it. It implements a function from class definition to class instance. At the same time it also consumes dynamic memory, and can only be called a limited number of times before memory runs out.

6.1.7 The return value

The return value of a function can be used in expressions. Sometimes the function is a natural and general one, like mathematical or trigonometric functions. Sometimes they provide analysis of a data structure that is only used in the program.

Functions can also be used to encapsulate complicated expressions to reduce clutter and increase managability on a higher level. They are also good to organize computations in a structured fashion.

The return value of a function method is limited to a single value. If more than one value is desired, the function must create an object that can hold multiple values, like an array or a class that fits the purpose.

When a method is returning an object to carry multiple values, it is not uncommon that the method also accepts such an object provided by the caller. This has the advantage that the caller can decide to create one object instance and then reuse it over several calls, to avoid the overhead involved with extra object creation.

Here is an example of this pattern, using an array. The method `centroid` accepts two arrays of coordinates, x and y values, and then finds the average value for both. The return value is the x and y coordinate of the centroid of the polygon, and it is returned in an array. If the caller does not provide an array, a new one is created:

```
public double [] centroid (double [] x,
                           double [] y,
                           double [] rtn)
{
  if (rtn == null)       // if no array was provided
    rtn = new double[2]; // create a new one
  else {
    rtn[0] = 0.0;        // clear to zero so the
    rtn[1] = 0.0;        // sums come out right
  }

  for (double d : x)     // add up all the x
    rtn[0] += d;         // coordinates
  rtn[0] /= x.length;    // find average

  for (double d : y)     // add up all the y
    rtn[1] += d;         // coordinates
  rtn[1] /= y.length;    // find average
```

```
    return rtn;            // return center x,y
  }
```

6.1.8 Expressions in actual parameters

To make a method call, the name of the method is stated and any parameters are given in a comma-separated list of expressions. Each expression must evaluate to a type that matches the type of corresponding parameter variable, so that the value of the expression can be assigned to the parameter variable. For example, assume we have written a method `charColour` which does something clever with a character and a colour. It is declared like this:

```
  public void charColour (char c, int rgb, double alpha) {
    ...
  }
```

In a call to `charColour` we present the following arguments:

```
  int red   = 255;
  int green = 255;
  int blue  =   0;

  charColour ('Z', red*65536+green*256+blue, 0.7);
```

In the setup of this method call, the following happens:

- The character literal `'Z'` is assigned to parameter `char c`.
- The expression `red*65536+green*256+blue` is evaluated and the result assigned to `int rgb`.
- The double literal `0.7` is assigned to `double alpha`.

These evaluations and assignments are no different from ordinary variable assignment, including automatic and explicit type conversions. The thing to remember is that all expressions in the parameter list are completely evaluated *before* the method call is made.

6.2 Managing methods

Methods are the building blocks of programs. They help to structure the work and logic into managable units. A careful and clean design of methods will make the program easier to build, understand, and maintain.

In the Java programming language, object-orientation affords a second important role to methods. Methods are used to implement interface points on an object, and define what can be done to the object and what it can do. Rather than allowing any outside code to manipulate the variables in an object instance, all such requests except the most trivial ones are instead guided to method calls. The methods allows the object to accept or deny a request, and to enforce its design boundaries. This is the concept of *encapsulation*, and when properly designed and implemented it provides structure and increased maintainability to a software system.

6.2.1 Sequences of method calls

With appropriately designed methods, the structure of a superordinate task becomes a (usually short) sequence of method calls. For example:

```
readInput();
translate();
writeOutput();
```

Each method performs a separate and well-defined part of the whole program. By carefully choosing parameters and return values, the method is given a range of capability. It can be instructed what to do, and it can respond back to the caller.

Depending on the complexity of a subtask, it too can often be divided into different parts, some of which can be implemented as further methods. For example:

```
private static void translate() {
  checkInputData();
  normalize(syntaxTable1);
  normalize(syntaxTable2);
  cleanUp();
}
```

6.2.2 Method call chains

The `main` method is the topmost method of a Java program. It is entered when the program starts running. The `main` method then very likely makes a call to a method in the program, and that method makes further method calls, and so on. At any particular moment some method is active and executing, and above it a calling method is waiting for its return, and above that another method is waiting, all the way back up to the `main` method. This chain of waiting methods and one active method is known as a *call chain.* If execution is suspended (like one can do with a stepping debugger), the current call chain can be examined.

Understanding call chains is an important component of understanding what the program is doing — where it is executing and why. Here is an example of a small program to illustrate this concept:

```
public class CallChainDemo {

  public static void c() {
    System.out.print('c');
  }

  public static void b() {
    System.out.print('b');
    c();
  }

  public static void a() {
    b();
    c();
  }

  public static void main (String [] args) {
    a();
    b();
    c();
  }
}
```

Take a moment to examine the code for class `CallChainDemo`. Try to figure out what output it will generate (the answer is in the footnote[1]).

The first time method c is entered, the call chain looks like this:

```
main → a → b → c
```

The second time method c is entered, the call chain is:

```
main → a → c
```

The third time method c is entered, the call chain is:

```
main → b → c
```

And the last time method c is entered, the call chain becomes:

```
main → c
```

6.3 Local variables

Local variables can be declared in the method body as needed, just as in any block statement. The variables in the parameter list are also local variables, but they are of course different in that they are assigned by the caller of the method, and not by the method itself.

[1]The output is `bccbcc`.

Local variables must be explicitly initialized. Memory for them is taken from the call stack, and that memory is constantly being reused as the call chain grows and shrinks. Since local variables are highly likely to receive a value soon, the compiler does not insert code to initialize them. It would mostly be a wasted effort anyway.

Local variables in a method disappear as soon as the method returns. Their values are gone. If data is to be saved between method calls it must be placed in instance variables, files, or somewhere else outside the method.

6.4 Global variables

Global variables are class or instance variables that are accessible from all methods in the class. That is the *global* concept. If the variables are declared with an access not `private`, they may also be reachable from other classes.

Global instance variables keep their content for as long as the their object instance exists. Each object instance has its own copy of the class variable, so if there are 200 instances in the program, there are also 200 such variables.

Static class variables exist in the class definition, and there is only one instance of each such variable in the program. On the other hand, the variable exists during the whole execution of the program, and is available from the first statement to the last.

Class and instance variables are automatically initialized by the compiler, unless the source code explicitly provides an initialization in the variable declaration. The default values are a little different depending on the data type, but always the same. Scalar data types (including `char`) are given the value zero. The `boolean` is initialized to `false`, and reference data types (including arrays) are initialized to `null`.

6.5 Recursive iteration

For a certain class of problems, *recursion* can be used to create solutions that are short, succinct, and sometimes beautifully simple. Recursion is often used in mathematics, to provide definitions and algorithms.

A recursive solution is one in which the problem can be solved by iteration. But instead of using a loop, the recursive solution often follows this pattern:

- Is the problem solved? If so, return.
- If the problem is not solved, solve a portion of the problem, and then apply the solution *to what remains of the problem.*

In programming, a recursive solution almost always involves a method that calls itself, either directly or as part of the call chain. Here is an example of how to print all numbers in an array, recursively:

```
public static void printArray (int [] numbers , int index) {
  if (index < numbers.length) {
    System.out.println(numbers[index]);
    printArray(numbers , index + 1);  // call to self
  }
}
```

The method first checks if the index has reached the end of the array. If that is not the case, then the element at the current index is printed, and the method calls itself with 1 added to the index to print the rest of the array.

Assume that we have the following array:

```
int [] ia = {3, 5, 7};
```

and then some code issues the call:

```
printArray(ia, 0);
```

The we get a call chain that looks like this (arrows are used to show references, and the call chain goes from top to bottom):

```
printArray (numbers → ia, index == 0) // 1st level
  // prints 3
printArray (numbers → ia, index == 1) // 2nd level
  // prints 5
printArray (numbers → ia, index == 2) // 3rd level
  // prints 7
printArray (numbers → ia, index == 3) // 4th level
  // does not print and does not recurse further
```

Recursion works because the variables in the method are local to each call. Each time the method is called, new memory is allocated (from the call stack) for the parameter variables. When the recursive chain reaches the lowest level, there are four different `numbers` and `index` variables active, each one belonging to a separate method activation.

The printing of the array could easily have been accomplished by a `for` loop, and that would be more efficient. The recursive solution consumes a frame of call stack memory for each element in the array. Parameter `numbers` never changes its value, the reference to the array is simply copied to the next method call. Recursion is rarely the preferred solution for one-dimensional arrays.

However, there are other data structures that cannot be navigated by incrementing a linear index. For these, recursive algorithms can be a good choice.

Chapter 7

A class library

A desirable property of well-written code is that it can be reused. This obviously saves work by not having to reinvent the wheel with each new project. In addition, code that has been scrutinized, debugged, and vetted against several projects is likely to be of good quality, with high performance and low risk of bugs.

Routines and data structures that are general enough to be useful in many different programs are often placed in *software libraries.* The programmer can then specify that certain functions in a program are to be performed by the code in the libraries, for example mathematical functions, list manipulation, XML parsing, and so on.

7.1 Distributed code

When assembling a software library, there are often several hundred different source code files to manage. They describe constants, variables, functions, and procedures that are part of the library. They do not, however, contain any main programs.

The software library can be written by many different programmers and evolve over several years. When bugs, security flaws, or missing features are noticed, the library must be fixed, but is not always easy to do that without breaking existing applications that rely on the libraries being exactly as they are. For that reason the choice is sometimes to keep the old routines in, and add new and better routines, with the recommendation that new code only use the new routines. This will keep old programs from suddenly crashing, while newly written software can benefit from the updates. The old library routines are now *deprecated*, meaning that their use is strongly discouraged.

In order to put a program into a state where it can be run, the code for the program must be found and loaded. Any libraries that it relies upon must also be found and loaded. With the program code in one place, and the libraries often installed somewhere else, the code for the program is in practice *distributed*, i.e. it is spread out in different directories.

7.1.1 Class libraries

With Java, there were for several years four kinds of system libraries:

- The standard library, installed with the Software Development Kit (SDK) — this is almost certainly what you have on your personal computer
- The limited library for embedded and mobile devices
- The Enterprise Edition, for industrial strength servers
- A collection of small, special-purpose libraries, sometimes contributed by the Java programming community.

Since every source file in Java generates one or more class files when compiled, a software library in its simplest form is a collection of class files that the compiler and the Java Virtual Machine (JVM) has access to. It is, however, difficult to manage and distribute several thousand small files. Because of this, Java employs an archive file format called `jar` (Java Archive). The `jar` utility program is used to pack any number of class files into a single archive file, which can then be distributed as a complete unit.

Both the compiler and the JVM understands the `.jar` file format, and can use jar files directly. There is no need to unpack them first. They must of course be installed in such a location so that the compiler and JVM can find them.

7.1.2 Packages

The Java programming language has the *package* mechanism to organize classes. A package can contain one or more classes. A class is declared to belong to a package through the `package` clause in the source code file. Here is an example involving a Java source code file named `Quaternion.java`:

```
package hacks.g3d;

public class Quaternion {
  ...
}
```

```
                        ./
                        |
      +-----------------+-----------------------+
      |                 |                       |
    hacks/            prod/                    web/
      |                 |                       |
     g3d/          DBase.java (package prod)    |
      |                                         |
      |                                      server/
Quaternion.java (package hacks.g3d)             |
 Frustrum.java  (package hacks.g3d)        Parser.java
                                       (package web.server)
```

Figure 7.1: A package tree example

The `package` clause must be placed before the first class declaration in the file, because it applies to everything *in that file*. Consequently, all classes generated from this source code file are in the package `hacks.g3d`.

Finally, the file `Quaternion.java` must be placed in a directory called `g3d`, and that directory must be found in a directory called `hacks`. This creates a forced correspondence between where the files are located in the file system, and what it says in their `package` declaration. The compiler expects this correspondence, and will complain loudly if there is a mismatch.

Figure 7.1 shows how the package name corresponds to a directory in the file system for the source code file. The root directory is whatever directory the compiler has been instructed to use as the starting point.

If *no* package declaration is present, the class is placed in the *default package*. This is usually the current directory, and it is often fine for small, monolithic programming projects. There are, however, two caveats to be aware of concerning the default package:

- The default package is nameless. As a result you cannot `import` classes that are in the default package, because there is no name to give to the `import` clause.
- Every hap-hazard unpackaged class goes into the default package. This may create name conflicts between projects if they are using identical class names for different purposes.

7.1.3 Structure provided by IDEs

An integrated development environment may provide its own organization on top of the package system. Eclipse, for example, has the *project* concept. It is important to realise that the project is something provided by Eclipse, and may be called something different in another tool, while the Java *package* is the highest organizing structure defined in the Java programming language.

> Do not use packages until you clearly see the benefit with doing so, and the project is big enough. Compilation without the help of an IDE becomes more complicated with packages, and requires that the documentation for the compiler is consulted first.

7.2 The Java Standard Library

The Java Standard Library contains many packages, and each in turn contains classes. Some packages have many classes, others just a few. The packages present in the Standard Library broadly fall into three categories:

- They contain classes that are needed by the JVM and the Java language, like `java.lang.String` or `java.lang.Array`.
- They contain classes that are very often useful in general programming, like those in the packages `java.util` (utilities) and `java.io` (files).
- Special-purpose packages for various tasks. The majority of the classes in the standard library fall into this category: `java.awt`, `javax.swing` (2D graphics), `java.nio` (modern filesystem I/O), `java.rmi` (distributed systems), `java.security`, `javax.sql` (I/O with relational databases), `javax.sound` (sound and MIDI), `javax.xml` (XML processing) and several more.

> All Java programs require the package `java.lang`, and for this reason it is always imported automatically.
>
> Simpler programs that do not need graphics often use classes from the package `java.util`, and if they do uncomplicated I/O with files, they also need package `java.io`.

7.2.1 Class `java.lang.Math`

The class `java.lang.Math` provides static methods for logarithms, exponentiation, roots, and trigonometric functions. Since package `java.lang` is automatically imported, they can be immediately called from any expression.

The class also defines two public constants, `Math.E`, the base of the natural logarithms, and `Math.PI` (3.14159 . . .).

Remember that the floating point data types in most cases only contain an approximation of the true value. The methods in class `java.lang.Math` also

Method		
abs(a)	$\|a\|$	absolute value
acos(a)		arc cosine
asin(a)		arc sine
atan(a)		arc tangent
atan2(x, y)		rectangular to polar coordinates
cbrt(a)	$\sqrt[3]{a}$	cube root
ceil(a)	$\lceil a \rceil$	smallest integer not less than a
cos(a)		cosine
exp(a)	e^a	
floor(a)	$\lfloor a \rfloor$	greatest integer not greater than a
log(a)		the natural logarithm
log10(a)		the base 10 logarithm
max(a, b)		the greater of a and b
min(a, b)		the lesser of a and b
pow(a, b)	a^b	exponentiation
random()		a random number n, such that $0 \leq n < 1$
rint(a)		the closest integer as a `double`
round(a)		the closest integer as a `long` or `int`
signum(a)		-1.0, 0.0, or 1.0 dep. on the sign of a
sin(a)		sine
sqrt(a)	$\sqrt{a}$	square root
tan(a)		tangent
toDegrees(a)		convert radians to degrees
toRadians(a)		convert degrees to radians

Table 7.1: Some useful methods in `java.lang.Math`

return approximations, and for reasons of efficiency it is not always the closest possible approximation. However, it is close enough unless you are doing lengthy scientific calculations involving physics or extreme numbers.

7.2.2 Class `java.lang.String`

The class `java.lang.String` implements character strings in Java. Strings are used a lot, and receive special treatment by the language syntax and the compiler. In particular, a string object is created for each literal string in the source code:

```
String s = "This is a string.";
```

The operator `+` is overloaded to implement concatenation for strings. The operator can therefore be used in a string expression to create a new string from existing ones:

```
String a = "aaa";
String b = "bbb";
String c = a + b; // a new string object with
                  // the text "aaabbb" is created
                  // and the reference assigned to c
```

In mixed type expressions at least one of the arguments must be a string:

```
int answer = 42;
String d = "The Answer is: " + answer;
```

The compiler inserts code to first convert the `int` value in variable `answer` to a string, and then concatenates the two strings into the final string.

The following does not work:

```
String e = 42 + 38;
```

Since both arguments to the `+` operator are ints, the integer addition operation will be selected. The result will be have an `int` data type, and that can not be assigned to a `String` reference type variable. A compilation error is the result.

Strings are immutable. Once created they cannot be altered. When a mutable and reusable string is needed, the class `java.lang.StringBuilder` should be used instead.

Several methods exist in class `String` to allow the inspection of the string object, or to generate new strings that are modified versions of the original string. Several of these methods require an index or two. String indices are similar to array indices: the first character has index 0 and the last character has index $length() - 1$.

Method	
`char charAt(int index)`	Returns `char` at index
`int compareTo(String anotherString)`	Lexicographic comparison
`int compareToIgnoreCase (String s)`	Lexicographic comparison
`boolean contains(CharSequence s)`	Substring test
`boolean equals(Object obj)`	Equality test
`boolean equalsIgnoreCase(String s)`	Equality test
`int indexOf(String str)`	Substring search
`boolean isEmpty()`	Is the length zero?
`int lastIndexOf(String str)`	Substring search
`int length()`	Nof chars in the string
`boolean matches(String rx)`	Regular expression match
`String replace(char oc, char nc)`	Replace character
`String [] split(String rx)`	Split string at reg. exp.
`boolean startsWith(String pfx)`	Prefix substring test
`String subString(int sta, int end)`	Substring by index
`String toLowerCase()`	Convert to lowercase
`String toUpperCase()`	Convert to uppercase
`String trim()`	Remove leading and trailing spaces

Table 7.2: Useful instance methods in class `java.lang.String`

Table 7.2 shows some of the instance methods that can be called on a `String`. The list is not complete, and some of the methods listed (e.g. `replace`) are overloaded to allow for variants.

Notes on `charAt`

The `charAt` method can be used to extract individual characters from the string. The method accepts an index, and the character at that index in the string is returned. For example:

```
String s = "HelloWorld";

char c = s.charAt(0); // c == 'H'
c = s.charAt(1);      // c == 'e'
c = s.charAt(s.length()-1); // c == 'd'
```

We could also print the string in reverse if we wanted:

```
String s = "HelloWorld";

for (int i = s.length() - 1; 0 <= i; i--)
  System.out.print(s.charAt(i));

System.out.println();
```

Notes on `length()`

While the length of an array is found from the public final variable `length`, the length of a `String` is found by calling the method `length()`. Since the length of both arrays and strings can not change once the object is created, it would seem strange that one has a variable and the other a method.

There is a perfectly good reason though; the method `length()` in class `String` is specified by interface `CharSequence`, an interface that is implemented by not only by class `String`, but also by the standard library classes `CharBuffer`, `Segment`, `StringBuffer`, and `StringBuilder`, some of which have a variable length. It just so happens that `String` is immutable, and a string object cannot change its length.

Notes on `substring`

The substring method returns a new string where the text is copied from a section of the original string. The method is overloaded in two versions:

```
public String substring (int fromIndex)
```

A new string is returned that contains the characters from the given index up to the end of the string. For example:

```
String s = "HelloWorld";   // s → "HelloWorld"
String u = s.substring(5); // u → "World"
```

The other version of `substring` takes two arguments, the start index (inclusive) and the end index (exlusive):

```
public String substring (int fromIndex, int toIndex)
```

For example:

```
String s = "HelloWorld";   // s → "HelloWorld"
String u = null;
u = s.substring(0, 0); // u → ""
u = s.substring(0, 1); // u → "H"
u = s.substring(1, 2); // u → "e"
u = s.substring(5, s.length()); // u → "World"
```

Even though it may appear counterintuitive that the character at `toIndex` is not included in the substring, it actually makes things a little easier. For example, assume that we are reading semi-formatted data from a file. A typical line looks like this:

```
"Age: 32, Name: Nymphadora Tonks, Address: 64 Basil ..."
```

Now we want to extract the name field. We can do this by finding the name field, compute where the name starts, finding the first comma after that position, extract the substring between, and finally trim off leading and trailing spaces:

```
private static final String nameField = "Name:";
...
String line = ...; // a line of text, formatted as above

// Find the index of the name field, and
// add its length to get behind it
int startIndex = line.indexOf(nameField) +
                 nameField.length();

// From that position, find the index of
// the next comma
int toIndex = line.indexOf(",", startIndex);

// Take out the substring, and trim it
String name = line.substring(startIndex, toIndex).trim();
```

If the `toIndex` parameter was inclusive, we would be forced to subtract 1 from the argument, something which can be easily forgotten when there are lots of string manipulation.

7.2.3 The `valueOf` and `format` methods

Class `String` has a static method `valueOf()` which is overloaded for a selection of data types. The method returns a string that is the *text representation of the argument*. For example:

```
String s = String.valueOf(42); // s → "42"

boolean b = true;
s = String.valueOf(b); // s → "true"
```

The `valueOf` method is useful when you need the kind of conversion from value to text that `System.out.print` offers, but instead of printing it, you want it in a string.

It should also be remembered that for this purpose there is also the static `String.format` method, which offers complete control over how the conversion from a values to text is done. The `valueOf` method merely offers the default conversion.

7.2.4 Wrapper classes

Wrapper classes is the collective name of a set of classes in package `java.lang` that include the classes `Integer`, `Long`, `Double`, `Float`, `Byte`, `Boolean`, `Character`, and `Short`. These are classes that provide services related to the primitive data types.

A wrapper class basically offers two kinds of help. The first kind comes in the form of static constants and methods. The constants provide important values for the primitive datatype, like min- and maximum, and special values. The methods implement operations that are relevant to the primitive data type, for example parsing a number from a string of digits, or manipulating the bits in an integer.

The second kind of service offered by a wrapper class, is to provide an object for *one primitive data type instance*. This solves the problem that you cannot create a reference to a primitive value, only to an object. For example, assume that you want to have a dynamic list of `double` in your program. An array could do it, but arrays are not optimal for dynamic data structures that change a lot and vary greatly in size. So you turn to `java.util.LinkedList`. There is just one problem. The `LinkedList` cannot contain a `double` value. It can only contain objects (or more accurately, references to objects).

Here is where the wrapper class `java.lang.Double` comes in handy. You take your integer value, you *wrap* an instance of `Double` around it, and now the value can go into the list, riding inside the wrapper class instance:

```
import java.util.LinkedList;
...
LinkedList<double> dlist; // compilation error

LinkedList<Double> dlist; // OK

dlist = new LinkedList<Double>(); // new list of Double

dlist.add(new Double(0.707259)); // add a Double to list

Double dd = dlist.get(0); // get it out again

double d = dd.doubleValue(); // extract double value
```

Wrapper object instances are immutable, the value they carry inside cannot be altered.

Class `java.lang.Double`

Class `java.lang.Double` is the wrapper class for the primitive data type `double`. The class defines several constants, of which two are worth mentioning:

```
Double.MAX_VALUE
Double.MIN_VALUE
```

`Double.MAX_VALUE` is the greatest positive number a `double` can represent. `Double.MIN_VALUE` is the smallest, non-zero, *positive* number a `double` can represent. Remember that in the floating point format, the sign is independent of the digits in the number. Therefore, the most negative `double` number is:

```
-Double.MAX_VALUE
```

Please note this difference to the constants in the integer datatypes, in which the `MIN_VALUE` constant is the greastest negative number.

Class `java.lang.Integer`

The class `java.lang.Integer` is the wrapper class for the primitive data type `int`. It defines the following two constants:

```
Integer.MAX_VALUE
Integer.MIN_VALUE
```

They are the greatest (most positive) and least (most negative) numbers that an `int` can represent. Please note the difference to floating point numbers.

Here are some useful static methods in class `Integer`:

```
static int parseInt (String s)
```

Attempts to parse the characters in the string as an integer number, and if successful, returns that number as an `int`.

```
static String toBinaryString(int i)  // base 2
static String toHexString(int i)     // base 16
static String toOctalString(int i)   // base 8
static String toString(int i)        // base 10
```

These methods return a string representation of the given number, for the number system bases 2 (binary), 16 (hexadecimal), 8 (octal), and 10 (decimal).

```
static Integer valueOf(String s)
```

This is a short-cut when you want to parse an integer from a string and put it in a wrapper instance:

```
String s = ...; // The number string

// the long way to do it
Integer ii = new Integer(Integer.parseInt(s));

// or we can do
```

```
Integer ii = Integer.valueOf(s);

// in fact, we can also do
Integer ii = new Integer(s);
```

As you can see, there is a fair bit of redundancy available in the creation of an `Integer`.

Class `java.lang.Character`

The wrapper class `java.lang.Character`, in addition to giving a wrapper object instance when needed, provides methods that helps in the categorization of a Unicode character. To this end, it presents a large set of constants that relate to Unicode character classes. It also has some methods that help in the parsing of a number, when this cannot be done using the already available methods:

```
static int getNumericValue (char ch)
```

This method returns the `int` value 0, when given the character `'0'`, the `int` value 9, when given the character `'9'`, and similarly for the digits in between.

To analyse characters in a text, the following methods (and several more) are available:

```
static boolean isDigit(char ch)
static boolean isLetter(char ch)
static boolean isLowerCase(char ch)
static boolean isUpperCase(char ch)
static boolean isWhitespace(char ch)
```

When you consult the documentation[1] you will find that several methods in class `Character` are overloaded to also accept arguments of the type `int codePoint`. There are also several methods that support a concept called *surrogates*.

The reason for codepoints and surrogates is that in Java, the `char` data type is two bytes (16 bits). The classes `String`, `StringBuilder`, and `StringBuffer` use `char` arrays internally to hold the characters. But the Unicode standard allows for characters with codes larger than what fits in 16 bits. Such *supplementary character* codes will not fit into a single `char`. They must be given special treatment.

The solution is to use two adjacent `char`s in the array to represent a single supplementary character. This is called a *surrogate pair*. The code in the first `char` is taken from a Unicode number sequence called the *high surrogate range*. The code in the second `char` is taken from the *low surrogate range*.

[1]`http:docs.oracle.com/javase/8/docs/api`

The surrogate pair solution makes it possible for the character logic to examine a single `char` value and immediately see where it belongs:

- It is a normal, 16-bit character
- It is the first `char` of a surrogate pair
- It is the second `char` of a surrogate pair

If a program wish to manipulate text that contains supplementary characters, `String` and `StringBuilder` can still be used, but they will require the use of different methods and techniques than for a standard program. An alternative solution is to use arrays of `int`. With 32 bits in each `int`, each character fits into a single array element. So, when the program then wants to call methods in class `Character`, the methods that accept a parameter of type `int codePoint` support the full range of the Unicode character set, both ordinary `char` values and the supplementary characters.

7.2.5 Class `java.lang.System`

The class `java.lang.System` gives access to components and properties of the running JVM. For example, class `System` has the final variables `in`, `out`, and `err`, the default streams for input and output. When something is printed with:

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

what that really means is to access the reference held in the variable `java.-lang.System.out`. That reference leads to a `PrintStream` object, and that is the object that has the `println` method.

Further static methods of interest in class `System` are:

```
static long currentTimeMillis()
```

Returns the current time in milliseconds since 1 Jan, 1970.

```
static void exit (int status)
```

Terminate the JVM at once and return to the operating system with the given status. Please note that there are only two good reasons to use this. The first is to do an immediate emergency exit where a normal program exit is not viable. A normal program exit is to end via the `main` method, as dictated by control logic. An exit via an exception explicitly thrown by the program in response to some condition, is also a normal exit.

The second reason for using `System.exit` is that the Java program was launched by some automation, like a shell-script or job controller, that takes an interest in the exit value. In these situations it is good coding practice to use a single exit statement at the end of the `main` method, and manage state in the normal way until it is reached. The by far worse scenario is to have a program sprinkled with exit statements, perhaps in different parts of the code. The program is given no chance to clean up properly, if some far away class with no knowledge of what has been added since it was created, has the authority to kill the whole application.

```
static void gc()
```

Run the garbage collector. This may or may not free up some memory, depending on how many objects that are waiting to be reclaimed. There must be no lingering reference from a program variable to an object, if it is to be recycled.

7.2.6 Class `java.util.Arrays`

The class `java.util.Arrays` holds utility routines for arrays. There are method sets for the following tasks:

- binary search for an element in a sorted array
- copy all or part of an array
- `deepToString` — create a string representation of a multidimensional array
- equality test for arrays
- fill all or part of an array with a value
- sort all or part of an array
- `toString` — create a string representation of a one-dimensional array

7.3 Exceptions

This section is a primer on exceptions. For more, see chapter 12.

Exceptions are objects that represent that an exceptional state has occurred in the running program. It is something that the program was not designed to process, and it cannot perform the operation that lead to the exception. An attempt to divide by an integer by zero, for example, leads to an `ArithmeticException`.

Exceptions are classes that instantiate, or extend, `java.lang.Exception`. They obey all rules that govern classes and objects. They are different from other classes in that they are recognised by the JVM as being part of the exception handling machinery.

An exception is created as usual with the `new` operator. When it is created, it is often configured to contain a small error message, detailing the exact nature of the problem. The exception object is then *thrown* by the operator `throw`, and that is what invokes the exception mechanism. For example:

```
if (n == 0)
  throw new ArithmeticException("Divide by zero");
```

When an exception is thrown, normal execution of statements and method returns are suspended. From the point of the `throw` clause, the JVM now looks if it is inside a context that will *catch* the exception (more about this later). If such a context is found, then control is transferred there, and the code prepared for dealing with this kind of exception is executed. If no such context is found in the current method, the exception is instead thrown upwards, to the point of the method call. Now the calling method has received the exception, and again a search is made for something that will catch the exception.

The search for a place that will catch the exception continues upwards, through the method call chain, until the `main` method is reached. If the `main` method does not catch the exception, then it is again thrown upwards, but this time it goes into the JVM where it always is caught, and the program is halted with an error message. For example:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Examples.fex02(Examples.java:61)
        at Examples.fex01(Examples.java:65)
        at Examples.main(Examples.java:72)
```

Look carefully at the message. The first line tells what exception it was, in this case an `ArithmeticException` with the added text `/ by zero`. This is what caused the exception.

The second line says that the exception happened in method `Examples.fex02`, located in source code file `Examples.java` on line 61. Since the exception was not caught, it continued upwards to where method `fex02` was called. The third line says, that the call was made from method `Examples.fex01`, and that in turn was called from method `main`. From this we know where the problem happened, and how we got there.

7.3.1 The exception hierarchy

The exception classes are organized in a wide hierarchy, with many descendants. There are approximately 70 subclasses that extend class `Exception`, and they deal with many different tings, most of which are never seen. The two most prominent groups of exceptions that a Java programmer sees, are these:

```
java.lang.Exception
  |
  +--IOException
  |    |
  |    +--FileNotFoundException
  |    |
  |   ...
  |
  +--RuntimeException
  |    |
  |    +--ArithmeticException
  |    |
  |    +--ClassCastException
  |    |
  |    +--IllegalArgumentException
  |    |
  |    +--IndexOutOfBoundsException
  |    |
  |    +--NullPointerException
  |    |
 ...  ...
```

7.3.2 Checked and unchecked exceptions

Methods that may throw an exception must declare this so that the compiler properly can check that any attempts to catch it actually matches. From this follows, that if method `a` calls method `b`, and `b` declares that it may throw an exception, then method `a` must either catch it, or itself declare that it may throw such an exception, because it might get handed one from the call to `b`. This check requirement applies to all exceptions *except* `RuntimeException` and its subclasses.

A `RuntimeException` is not checked by the compiler, and there is no need to declare it in the method declaration. The runtime exceptions are *unchecked*. All other exceptions are *checked*.

7.3.3 Create, throw and declare

An exception object is created with the `new` operator as usual, and thrown with the `throw` operator. Below is a small example of a method that only accepts a parameter value in a certain range. Any out-of-range arguments are to be rejected. Since the exception used is a subclass of `RuntimeException`, we do not need to declare it in the method declaration:

```
public void setTurnRate(double rate) {
  if (rate < 0.0 || 5.2 < rate)
    throw new IllegalArgumentException("out of range");
  turnRate = rate;
}
```

The second example is a method that attempts to open a given file for reading. Since it is dealing with files and the filesystem, it may be that a `FileNotFoundException` is thrown by the standard library class it is using. The method will not deal with this, and declares that it may throw a `FileNotFoundException` in turn:

```
  import java.io.*;
  import java.util.Scanner;
  ...
  public void loadFromFile (File inf)
    throws FileNotFoundException
  {
    Scanner in = new Scanner(inf);  // possible exception
    while (in.hasNextLine()) {
      String line = in.nextLine();
      ...
    }
    in.close();
  }
```

It is the creation of the `Scanner` instance that may throw a `FileNotFoundException`, as declared by class `Scanner`.

Chapter 8

Algorithms (1 of 2)

For this chapter, there is a separate document provided. The first portion of that document should be consulted.

Chapter 9

Objects (1 of 2)

Object-oriented programming relies on objects as the building blocks of programs. Objects provide a way to keep code and data together. Operations on a string, for example, are defined by and kept inside class `String`. The only manipulations that can be performed on the text in the string are those allowed by the methods defined in class `String`.

The restriction by methods leads to two good concepts: *encapsulation* and *interfaces*. Encapsulation protects the data inside the object from undue modification, and thus the integrity of the implementation can be upheld. The use of interfaces allows the implementation to be changed, if needed, without breaking other code as long as the old interface remains intact. Both these factors helps with building software systems that are robust and can be efficiently maintained.

9.1 Defining, creating, and using objects

Objects are defined by classes, created by the `new` operator, and used by code that follows a reference and then applies the `.` (period) operator to access methods and fields (variables and constants).

9.1.1 A composite data type

A primitive data type only has a single value. A composite data type combines several primitive and reference data types into a unit. In Java, this unit is the class.

As a place to start, assume that we want to represent a two-dimensional coordinate in our program, a point on a plane. We also expect to have several such points. We need a data type for a point. We can achieve such a data type by declaring a class:

```
public class Point {
  private double x = 0.0; // x coordinate
  private double y = 0.0; // y coordinate
}
```

This is a good start; we have made a public class that any code can make objects of, and we have our two instance variables together as a unit. The variables are protected and can not be reached from code outside the class.

9.1.2 Initialization

Our `Point` class is only useful if we can assign values to the coordinates. Preferably we would like to do this when we create the object instance, so that we immediately can specify the coordinates if we know them.

Each object can have code that is executed when it is created. This code is in a *constructor*. Using overloading, one of the constructors is selected and called as the `new` operator executes. Only when the constructor have completed (including constructors in superclasses, and initialization of variables) is the `new` operator done, and hands over the reference to the new object instance.

A constructor departs from the syntax of ordinary methods in that it *has no name*. It only has a type, and the type is that of the class it is constructing. Another equally valid view is that the constructor has no type, and its name is that of the class. The syntax remains the same.

We add the following three constructors:

```
public Point () {}   // explicit default constructor

public Point (double x, double y) {
  this.x = x;
  this.y = y;
}

public Point (Point p) {
  this.x = p.x;
  this.y = p.y;
}
```

A class may have a *default* constructor, a constructor without parameters. If no other constructors are specified, an empty default constructor is not required in the source code either. However, in this case we have added two non-default constructors, and since we still want to have a default constructor, we must now specify it explicitly, even though we have no code for it to perform.

The second constructor accepts two parameters, x and y. These will be the initial coordinates for the point. But, since we have chosen the names of the parameters to be identical to those of the class variables, we have *shadowed* the instance variables. Inside the scope of the method body, the compiler will assume that x means the local parameter variable, and not the instance variable.

To disambiguate the situation we use the self-reference this. It always refers to the object that the code is in, and by following the reference to ourself we can now access our instance variables and complete the assignments. The values in the local parameter variables are copied to our instance variables and will remain there.

The third constructor is a *copying constructor*. It uses an already existing point to create a new point. The coordinate values are copied from the given point. The this self-reference is used here too for symmetry, even though it is not strictly necessary. The reference variable p provides the required disambiguation.

9.1.3 Methods and behaviour

Public methods define what operations are available on the data in a class. Anyone can call public methods. Non-public methods are internal to the implementation, and perform tasks appropriate for the storage or processing of data.

We are now ready to add some methods to our Point class. They will provide access and define the interface that other parts of the program must negotiate with. The first thing we want is to be able to reassign the coordinates, individually. At this time we have no limits on the inputs, but we may have in the future, so we proceed with methods so that we can add limits checking later if needed:

```
public void setX (double x) {
  this.x = x;
}

public void setY (double y) {
  this.y = y;
}
```

Similarly, we want to be able to extract the coordinates when asked to do so. We provide the appropriate methods for this:

```
public double getX () {
  return x;
}

public double getY () {
  return y;
}
```

Since we are dealing with planar points, we also realize that it may be convenient to offer support for the calculation of the distance between between two points. We add this:

```
public double distance (Point p) {
  double dx = p.x - x;   // Distance on the x axis
  double dy = p.y - y;   // Distance on the y axis
  return Math.sqrt(dx * dx + dy * dy);
}
```

It may also be good to have the point's distance from the origin, it may become practical. We throw that in as well, using overloading:

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

9.1.4 The method `Object.toString`

All objects in Java inherit from class `Object`. This means that methods in class `Object` can be called using our `Point` as well. On such method, defined in class `Object` is the `toString` method:

```
public String toString() {
  ...
}
```

The purpose of the `toString` method is to return a string that describes the object. It is used in printouts and other situations that do not require fancy formatting. Since every class inherits this from class `Object`, you can print all objects. The implementation provided by class `Object`, however, is very generic and not very informative. Therefore, one should always *override* method `toString` with something better. We do this next for class `Point`:

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

This enables us to do this in some other part of the program:

```
Point p = new Point(4, 7);

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

and receive the printout

```
(4.0, 7.0)
```

9.1.5 Encapsulation and interfaces

Class Point demonstrates how it is possible to protect the data inside an object from outside manipulation. It can only be reached through the provided methods, and therefore our implementation remains in control over its integrity. We have enforced encapsulation and provided an interface for users of the class.

For this simple demonstration example, the benefit of encapsulation is not that obvious, but imagine instead that we are managing a more complicated data structure, one in which several variables must be appropriately updated together. In such cases encapsulation is the only way to stay in control, and avoid the dreaded spaghetti of code that is manipulating everything from everywhere.

9.1.6 References to objects

With primitive data types, an assignment to a variable invariably means that the value is *copied*. The values of the x and y coordinates in class Point are safe, because they are copies of anything given to them. Our methods ensure this.

However, let us see what happens when we implement a second class, a class that represents a triangle. Only a partial implementation of Triangle is shown here, because we need it to raise the awareness of a critical issue.

```
public class Triangle {
  private Point a;
  private Point b;
  private Point c;

  public Triangle (Point a, Point b, Point c) {
    this.a = a;  // copy the references
    this.b = b;
    this.c = c;
  }
}
```

Look at the constructor for class Triangle. As previously, we are copying the parameter values to the safety of class variables. But this time we are not copying primitive data type values, we are copying *references*. There are no Point objects inside the Triangle, only references to points outside it. And Point objects are mutable.

This means, that if some other code also holds a reference to one of the points in a Triangle that Point can change position and the Triangle would not know it. Perhaps this is not a problem for our application. Perhaps our implementation of Triangle is designed to allow for this, usually for reasons of efficiency.

But consider now what happens if the code in `Triangle` has inspected the triangle and computed some of its properties, like corner angles, side lengths, area, and more. Further, consider that these properties have been computed once and then were stored in instance variables, so that they could be retrieved quickly and efficiently, *on the assumption that the points in the triangle would not change position.* If that happens, and the points begin to wander around, then the cached properties and the points in the `Triangle` no longer agree, and there is a problem in the program.

What we need, in order to enforce the integrity of our implementation is to make our own copies of the points that are given to the constructor. We must *copy the resources*, not the references.

Handily enough, class `Point` comes with a copying constructor, so we can rewrite the constructor in class `Triangle` to the following behaviour instead:

```
public class Triangle {
  private Point a;
  private Point b;
  private Point c;

  public Triangle (Point a, Point b, Point c) {
    this.a = new Point(a);  // copy the resources
    this.b = new Point(b);
    this.c = new Point(c);
  }
}
```

Now we are safe. We have created our own copies of the points, and the only references that exist to them are in our encapsulated instance variables `a`, `b`, and `c`.

The danger is not over yet though. As development of the `Triangle` class continues, it may be decided that it should have methods that can get and set the corner points:

```
public Point getA () {
  ...
}
```

Now then, what code should go inside method `getA`? If we just return the value of variable `a`, similarly to what we did in class `Point`, then we have just handed out a reference to our own local copy of the `Point`, a reference we want to protect. We have compromised the integrity of our encapsulation.

Well, the answer is quite simple. We do not hand out or own `Point` to callers from other code. We give them a copy to play with:

```
public Point getA () {
  return new Point(a);
}
```

For the set methods, we continue to apply the same pattern, and copy the given resource for our own internal use:

```
public void setA (Point p) {
  a = new Point(a);
}
```

> Copying the resource should be the default design if the resource is a mutable object. This will strengthen the encapsulation of the object, and maintain its integrity in the face of maintenance and evolution of the application.

9.2 Objects that manage strings

There are several classes in the standard library that manage strings. The `String` class offers immutable strings and is also the compiler's choice for string literals in the source code. The `StringBuilder` and `StringBuffer` classes are available for strings that are built up incrementally, edited, erased, and rebuilt.

9.2.1 Class `String`

Class `String` has already been covered extensively in 7.2.2. Suffice to remember that class `String` objects are immutable and when manipulated what actually happens is that new strings are created. This is sometimes not the most efficient way, especially on memory-constrained systems like mobile phones or embedded systems.

9.2.2 The `StringBuffer` and `StringBuilder` classes

The classes `StringBuffer` and `StringBuilder` are identical, in terms of what methods and services they offer. So why are there two of them?

The first version of Java had only class `StringBuffer`. It provides mutable string operations, using dynamic memory management. You can add to the string, change it, erase it, or apply the reverse operation to it. The many `append` methods are overloaded to cater for the basic data types in the same way as `print` and `println`.

The idea behind the `StringBuffer` is to provide the composition of text in an incremental and memory-efficient manner. In order to ensure robustness, `StringBuffer` is also *thread-safe*. This means that if more than one thread is executing in the JVM, and two or more threads are trying to modify a `StringBuffer` object at the same time, then the object is designed to protect

Method	
`append(`a`)`	Append the argument (13 overloads)
`appendCodePoint(int codePoint)`	See 7.2.4, class `Character`
`charAt(int index)`	Returns the `char` at index
`delete(int start, int end)`	Delete text
`deleteCharAt(int index)`	Delete one character
`insert(int offset, `a`)`	Insert at offset (12 overloads)
`length()`	The current length of the text
`replace(int start, int end, String s)`	Replace text
`reverse()`	Reverse the current text
`substring(int start)`	Return a substring
`substring(int start, int end)`	Return a substring
`toString()`	Return the current text as a `String`

Table 9.1: Often used StringBuilder methods

itself from the effects of uncoordinated simultaneous modification. However, this protection comes at a cost, an increase in time which must be paid even if only a single thread is using the `StringBuffer`.

It was realized that the thread-safe property of the `StringBuffer` was a luxuary that was often not needed, because the majority of the programs that used `StringBuffer` were single-threaded. As a result, it was decided to create a new class with identical properties, but remove the thread protection. This class was `StringBuilder` and it was introduced with Java 1.5. The old `StringBuffer` is still there, and can be used when circumstances warrant, but for most programs the `StringBuilder` will provide a more efficient service.

Table 9.1 shows some of the often used methods in class `StringBuilder`. Where start and end indices are supplied, (`delete`, `replace`, and `substring`) they behave just as explained in class `String` (see 7.2.2) i.e. the region includes index `start` and excludes index `end`. For example, to erase the current string completely:

```
StringBuilder sb = ...; // some StringBuilder
...
sb.delete(0, sb.length()); // delete all text
```

The `reverse()` method is not a function, it reverses the current text inside the `StringBuilder` object. Thus:

```
StringBuilder sb = new StringBuilder();

sb.append("ABCDE");  // the current string becomes "ABCDE"
sb.reverse();        // the current string becomes "EDCBA"
```

As an example we will reuse the `toString` method of class `Point`, presented in 9.1.4. It produces a string that looks like this, for example:

```
(4.0, 7.0)
```

The original version uses a `String` expression which is not the most efficient way, because many intermediate strings must be created before the final result is achieved[1]. With a `StringBuilder` the text is built in steps:

```
// In class Point
public String toString () {
  StringBuilder sb = new StringBuilder();

  sb.append('('); // append a char
  sb.append(x);   // append a double
  sb.append(", ");// append a string
  sb.append(y);   // append a double
  sb.append(')'); // append a char

  return sb.toString();
}
```

Admittedly, this code are several lines longer than the original, but we are saving a bit of runtime memory by using just the one `StringBuilder` object. There is a twist to that, though. The `insert` and `append` methods, by design, return a reference to the `StringBuilder` object itself. This means that we can actually do:

```
// In class Point
public String toString () {
  StringBuilder sb = new StringBuilder();

  sb.append('(').append(x).append(", ").append(y).append(')');

  return sb.toString();
}
```

And if we feel really into hacker mode, we could even do:

```
// In class Point
public String toString () {
  return
    new StringBuilder().
      append('(').append(x).append(", ").append(y).append(')').
      toString();
}
```

Study that and see if you understand why everything can be put in the `return` statement. That insight is more important than the code itself. There are almost never any performance gains won from cryptic one-liners.

[1]Some Java compilers are clever enough to realize this, and sneak in a `StringBuilder` anyway.

Method	
`void close()`	Closes the scanner and the source
`boolean hasNext()`	True if there is a next token
`boolean hasNextDouble()`	True if the next token can be parsed as a `double`
`boolean hasNextInt()`	True if the next token can be parsed as an `int`
`boolean hasNextLine()`	True if there is a next line
`String next()`	Get next token as a `String`
`double nextDouble()`	Get next token as a `double`
`int nextInt()`	Get next token as an `int`
`String nextLine()`	Get next line as a `String`
`Scanner useLocale(Locale locale)`	Set the `Locale` to use

Table 9.2: Some `Scanner` methods

9.2.3 Class `java.util.Scanner`

The class `java.util.Scanner` (see also 3.1.4) is used to scan and parse input text. The text comes from a file, an input stream, or a text string already in the program. The source is given to the `Scanner` via the constructor. The `Scanner` reads from the source in response to method calls on it, methods that extract portions of the input. These portions are known as `tokens`.

The `Scanner` can also be configured to recognize certain expected input, for example the format of a decimal number. This is done by providing the `Scanner` with an instance of `java.util.Locale`, if the default will not do.

It is also possible to set the delimiter pattern. The delimiter pattern is a regular expression that the `Scanner` is using to determine where one token ends and the next one begins. The default delimiter is *whitespace*[2], i.e. the characters space, tab, newline (a.k.a. linefeed), form-feed, carriage return, and vertical tab.

The `Scanner` also has predicates that can be used to inspect if there indeed is a next token to read, or a token that matches a given pattern.

Table 9.2 shows a few of the more frequently used methods in class `Scanner`.

Using `Scanner`

This section provides two examples of how to use class `Scanner`. The first is how to create a console type interactive user dialogue, prompting the user to provide input, and then reading the input. The second example shows how to read from a text file.

[2]When you print on paper, there will only be *white space* and no ink from these characters.

When using a Scanner to read user input, we create the Scanner to read from the standard input stream, System.in. This stream is normally attached to the stream of characters coming from the user's keyboard, when the appropriate window has focus. Since the keyboard input is line-buffered to allow for typing corrections, the Scanner receives a new line of input each time the user presses the ENTER key. This line contains the characters that the user typed, including the ending newline character.

If we ask the Scanner for a token, but no input is present, the Scanner will block, waiting for input to appear. Therefore we must make sure, by using suitable prompts, that the user is aware that input is expected.

A second consideration concerns the difference between how the Scanner reads input when numbers are parsed and whole lines of text are requested. For example, if we ask the user to input the number of items, or weight of an article, and press enter, the next line from the keyboard will contain a sequence of digit characters, followed by the newline.

As we call one of the number parsing methods in our Scanner, it starts to read characters in the input. While the characters are whitespace, they are discarded. When it arrives at a number character, it switches over into number parsing, and continues to consume characters from the input as long as they are part of the requested number type. When the Scanner arrives at the newline character at the end of the number, *it will stop, and leave the newline intact.* The number accumulated so far is converted to its data type binary form and returned.

On the other hand, when we call Scanner.nextLine() to receive the whole next line of input from the user, the Scanner will read the characters in the input until it finds a newline. When it finds a newline, it stops, but this time *it removes the newline from the input.* The newline has been read too, but is not included in the return String.

The above differences in behaviour means, that as long as we are reading numbers, things work fine, because the whitespace characters (newline is a whitespace character) separates the numbers. It does not matter if the numbers are on the same line with whitespace between them, or each on a their own line, with newlines between them. The Scanner will return the next number regardless.

Similarly, if we are only reading whole lines of text because it contains things like names, addresses, or recipies for seabass, we obtain the full line each time with the newline at the end automatically read and discarded by the Scanner.

The problem occurs when we use a Scanner and then *mix* reading numbers and lines of text. To see why, imagine that we ask the user for a number:

```
System.out.print("Enter number of items: ");
```

and then we read it using our Scanner sc:

```
int nofItems = sc.nextInt();
```

Since there is nothing to read yet, the `Scanner` blocks, waiting for input. Meanwhile, the user dutifully types `42` and hits ENTER. This input is placed in the `Scanner`'s input buffer as the stream of characters:

```
['4']['2'][newline]
```

The `nextInt` method consumes the `'4'` and the `'2'`, but leaves the newline in the input buffer because it is whitespace and not part of the number. The input buffer now looks like this:

```
[newline]
```

The next step in the user interface dialogue is to ask for the name of the supplier. This is expected to be a full line of text:

```
System.out.print("Enter name of supplier: ");
```

Then the program makes the call to read the input line:

```
String supplier = sc.nextLine();
```

The programmer expects the `Scanner` to block again, waiting for input. However, and here is the issue, since the input buffer already contains the newline left over from the parsing of the number of items, the `Scanner` immediately sees that it has arrived at the end of the input line. It throws away the newline and returns an empty string as the name of the supplier.

Having obtained, what it believes to be the supplier name, the program continues, and goes on to ask for the next piece of input, and as the astonished user begins to read the previous prompt, the next prompt is already showing on the screen:

```
System.out.print("Enter price per item: ");
```

and the program has already gone off to call `sc.nextDouble()` which now will block, because there is no input available.

The entire messed up dialogue would look like this:

```
Enter number of items: 42
Enter name of supplier:
Enter price per item: []   <-- cursor waiting for input
```

The solution to this particular problem is to add an extra call to `nextLine` when switching from reading a number to reading a line. The extra call to `nextLine` will discard the whitespace that ended the number parsing, up to and including the terminating newline. To fix our program:

```
System.out.print("Enter number of items: ");
int nofItems = sc.nextInt();

sc.nextLine(); // discard whitespace after number

System.out.print("Enter name of supplier: ");
String supplier = sc.nextLine();

System.out.print("Enter price per item: ");
double price = sc.nextDouble();
```

In summary: when using a Scanner to read numbers and lines, always insert an extra call to nextLine when going from reading a number that is the last on its line, to reading a whole line. This applies to *all* sources.

User dialogue example

This example shows how to implement a simple user dialogue using a Scanner. It also demonstrates the use of a StringBuilder to compose the final output of the program.

```
import java.util.Scanner;

public class UserDialogueExample {
  public static void main (String [] args) {
    int numberOfItems = 0;
    String supplier = "";
    double price = 0.0;

    Scanner sc = new Scanner(System.in);

    System.out.print("Enter number of items: ");
    numberOfItems = sc.nextInt();
    sc.nextLine(); // eat extra newline

    System.out.print("Enter name of supplier: ");
    supplier = sc.nextLine();

    System.out.print("Enter price per item: ");
    price = sc.nextDouble();

    StringBuilder sb = new StringBuilder();
    sb.append("Total cost from supplier ");
    sb.append(supplier);
    sb.append(": ");
    sb.append(numberOfItems * price);

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

File reading example

This example assumes that there is a text file where each line contains a price followed by a product name. For example:

```
 6.70 Natural Ingredients deodorant
12.90 Natural Ingredients men's perfume
  ...
```

The task is to read the file and display the cheapest product. We assume that the name of the file to read comes from the first commandline argument, found in the array of strings given to `main`:

```
import java.util.Scanner;
import java.util.Locale;
import java.io.FileNotFoundException;
import java.io.File;

public class LowestPriceFinder {
  public void main (String [] args)
    throws FileNotFoundException
  {
    // These two variables hold the lowest price item
    double lowestPrice = 0.0;
    String productName = "";

    // Crate and configure a Scanner
    Scanner sc = new Scanner(new File(args[0]));
    sc.useLocale(Locale.US);  // decimal period

    if (sc.hasNextLine()) {
      // Read the first line of the file
      // to get our initial candidate
      lowestPrice = sc.nextDouble(); // price
      productName = sc.nextLine();   // rest of the line

      // Then read the rest of the lines
      while (sc.hasNextLine()) {
        double price = sc.nextDouble(); // price
        String name = sc.nextLine();    // rest of the line

        // Check and maybe update
        if (price < lowestPrice) {
          lowestPrice = price;
          productName = name;
        }
      }
    }

    sc.close(); // Remember to close

    System.out.println(lowestPrice + " " + productName);
  }
}
```

There are three things that are noteworthy in the program `LowestPrice-Finder`. The first is that even though it is switching between reading numbers and lines, it does not need to insert any extra calls to `Scanner.nextLine`. The explanation for this is that the double number is not the last token on the line. The extra `nextLine` is only required when a number is the last item on a line, and we want to read the next line as a string. In the `LowestPriceFinder` program, the number is the *first* token on each line, and having read it, we then call `nextLine` to read the remaining text on the line, which is the product name.

The second thing to notice is that the `Scanner` is *closed* once we are finished with it. It is good programming practice to close a resource that is no longer being used. It frees resources both in the JVM and in the operating system.

The third feature of interest is that the program is using the *update algorithm* to find the cheapest product. It reads the first product in order to get the first candidate. Then, if there are more products (lines) in the file, those are read in a loop, and as soon as a cheaper product is found, the variables describing the best candidate so far are updated. When all products have been examined, the candidate variables will contain the price and name of the cheapest product (or more correctly, the first cheapest product, as there may be several products with the same lowest price).

9.3 Typical services offered by an object

The methods in an object can be categorized into six different groups, depending on what it is they do, and the kind of service they provide to a caller. It is common to find at least a few of these categories represented in a well-written class. They are:

- Inspectors — methods that allow the inspection of the data inside the object
- Mutators — methods that allow the modification of the data inside the object
- Combiners — methods that combines the object with another object and returns a result that represents the combination
- Comparators — methods that are used to compare two objects with each other, usually for sorting or ordering purposes
- Predicates — methods that report if a certain property of the object is `true` or `false`
- Transformers — methods that transform the object into another representation

The reason for these catagories, and the value of being aware of them, is that they make it easier to design the interface to the object. The design should make every attempt to place each method into one, and only one, of the above categories. This makes the object's interface more clear and easy to use.

We will now exemplify these method categories, using examples picked from classes that we have previously reviewed or created.

9.3.1 Inspectors

Inspectors are methods that allow the inspection of the data inside an object. The inspection does not modify the data. For example, these methods are all inspectors:

```
// In class String and StringBuilder:
char charAt(int index)
int length()
// In class Point:
double getX()
```

9.3.2 Mutators

Mutators allow the caller to modify the data in the object. Here are some mutators:

```
// In class StringBuilder:
append(a)
insert(a)
reverse()
// In class Point:
setY(double y)
// In class Scanner:
useLocale(Locale locale)
```

Class `String` is not mutable, so we cannot find an example of a mutator there.

9.3.3 Combiners

Combiners are methods that combines the object with another object and returns a result that represents the combination. For example:

```
// In class String:
String concat(String s) // implements + for strings
// In class Point:
double distance(Point p)
```

9.3.4 Comparators

Comparators are methods that are used to compare two objects with each other, usually for sorting or ordering purposes. Like so:

```
// In class String:
int compareTo(String s)
```

The method `compareTo` is often used when sorting. The method call `a.compareTo(b)` should return a value less than zero if object `a` is lesser than object `b`, and thus should be sorted before `b` in an ascending sort.

The return value is zero if `a` and `b` are equal, i.e. both objects should be sorted to the same position.

The return value is greater than zero if `a` is greater than `b`, and should be sorted after `b` in an ascending sort.

An additional requirement is that calling `b.compareTo(a)` instead, should return the opposite result, or zero too.

It must be realized, that in this way it is the object itself that decides what goes into the comparison. The actual comparison is encapsulated, and can be evolved to suit the purpose of the program. More importantly, a sorting algorithm does not need to know *how* the objects are compared against each other. All it needs is the result of the comparison.

9.3.5 Predicates

Predicates are methods that report if a certain property of the object is `true` or `false`. For example:

```
// In class Object, and therefore in all classes:
boolean equals(Object o)
// In class String:
boolean isEmpty()
// In class Scanner:
boolean hasNextLine()
```

The `equals` method is inherited from class `Object` and therefore all objects can be compared with it. The default implementation is *really* simple:

```
public boolean equals (Object o) {
  return this == o;
}
```

This is the most general form of equality, and the only one that can apply to all objects: two objects are equal if, and only if, they are the one and the same object.

In practice, there are more involved definitions of equality, like that of two strings. Class `String`, for example, overrides `equals` to return `true` if two different `String` objects have identical character sequences. Such a lexicographic definition is much more useful when working with strings.

We could apply a similar logic to our class `Point`, and define the equality of two points as their coordinates being identical:

```
public boolean equals(Point p) {
  return this.x == p.x && this.y == p.y;
}
```

Just as with the `compareTo` method, the object defines what it means to be `equal` to another object. However, if it overrides `Object.equals`, then it should also ensure that:

```
a.equals(b) == b.equals(a)
```

If the two objects are equal, then it must not matter which one is used to call the `equals` method, and which one is the parameter.

9.3.6 Transformers

Transformers are methods that transform the object into another representation. Some examples are:

```
// In class Object:
String toString()
// In class String:
char [] toCharArray()
```

The `toString` method is probably the most ubiquitously implemented transformer. It returns a string representation of the object.

In the standard library classes it is not uncommon to find transformers that create arrays of various kinds. This is useful if one wants to apply an array-based algorithm on the data.

It should also be noted that the transformer is bit of a misnomer, as it does not modify the original object. Instead a different representation is created, with content based on the data in the object.

Chapter 10

Algorithms (2 of 2)

For this chapter, there is a separate document provided. The second portion of that document should be consulted.

Chapter 11

Objects (2 of 2)

This chapter continues the review of objects and object-oriented programming in Java.

11.1 Object resources

As should be clear by now, an object is not just a container of data. It also carries with it code in the form of methods, code that can be used by other classes to operate on the data in the object. We can view these methods as the *computational resources* in an object.

In the following sections, the concept of object resources is exemplified with selected classes and methods from the standard library.

11.1.1 `String.length()`

The `String.length()` method returns the number of characters in the string. This number is not just the length of some internal array or list, it is a computed quantity, arrived at through the examination of the actual Unicode characters in the string.

The reason why `length()` is computed, is that what the method actually returns is the number of Unicode units in the string. A Java `String` is stored internally in an array of `char`, but since there may be supplementary Unicode characters in the string, and such characters are represented by two char values (see 7.2.4), the relationship between the internal length of the `char` array and the number returned by the `length()` method depends on the contents of the string.

As Java programmers we are not often interested in this technical curiosity, because for most languages the length of the string and the length of the internal array will agree. However, if we are managing a large amount of supplementary characters in a memory-constrained device, then we need to consider the relation between text size and memory use. If you think this scenario is unlikely, consider a mobile app that displays Linear B, ancient Greek, old Italic, old Persian, Egyptian hieroglyphs, musical symbols, or Emojis, to name a few languages and character sets with supplementary characters.[1]

11.1.2 `String.isEmpty()`

The predicate `String.isEmpty()` may appear trivial, because given some `String s`, calling `isEmpty()` it is equivalent to the test:

```
s.length() == 0
```

The reason why the method is there is therefore not a computational one, but a *human* one. If our code compares the length of the string to zero, then we are using numbers and quantities. We therefore think of it in a numerical context, and some reader of the code will have the mind set towards further computation, perhaps involving the length of the string, or avoiding a division by zero.

On the other hand, if we instead call `String.isEmpty()`, then it is obvious that we do not care for the numerical property of the string, but instead *if it is empty or not.* This is a logical condition, and a property of text, and the reader's mind goes instead to the proper composition of strings, or the presence or absence of data, something which probably is closer to the mindset of the programmer.

The resource in this case, is that it helps the programmer to pick the operation that best describes the task that at hand. In addition to that, we should not allow ourselves to be bothered with how an empty string is defined. If it says its empty when we ask, ok then, fine, it is empty. Mucking about with the length only confuses the issue and inserts in our program unneccessary dependencies on how class `String` is implemented.

11.1.3 `Integer.`***type***`Value()`

Class `java.lang.Integer`, the wrapper class for the primitive data type `int`, provides a resource that is the conversion of the `int` inside the `Integer`, to some other primitive data type. Here is the complete list of methods:

- `byte byteValue()`

[1]Go see for yourself at `http://unicode-table.com/en/`

```
                      Number
                        |
    +-------+--------+---+----+-------+-------+
    |       |        |   |    |       |       |
   Byte   Double   Float | Integer  Long   Short
                         |
     +-------------+----+-------+------------+
     |             |            |            |
AtomicInteger  AtomicLong  BigDecimal  BigInteger
```

Figure 11.1: Class Number and its descendants

- `double doubleValue()`
- `float floatValue()`
- `int intValue()`
- `long longValue()`
- `short shortValue()`

The presence of these functions is motivated by the abstract class `java.lang.Number` in which they are defined. Class `Integer`, together with `Byte`, `Double`, `Float`, `Long`, and `Short` inherit from `Number`. However, since class `Number` does not provide implementations for the *type*`Value()` methods (most of them are abstract), the numerical wrapper classes must implement these methods or themselves be abstract.

Figure 11.1 shows all classes that extend class `Number`. The first row contains the wrappers for the numeric primitive data types (note that `Character` for the unsigned `char` is not present). The second row are not wrapper classes. They are included for completeness and not discussed further here.

Ok, so class `Integer` inherits from `Number`. Therefore `Integer` *is* a `Number`. Class `Double` is also a `Number`, and so are the other numerical wrappers. This, and the presence of the *type*`value()` methods, opens the possibility of generalized numerical type computing. For example:

```
public double average (Number [] nums) {
  double sum = nums[0].doubleValue(); // (1)

  for (int i = 1; i < nums.length; i++)
    sum += nums[i].doubleValue(); // (1)

  return sum / nums.length;
}
```

The method average accepts an array of `Number`, and suddenly we can actually have an array of numbers with different primitive data types! The *type*`value()`

methods will take care of data type conversion, doing rounding or truncation as required.

We could set up the following code to call method `average`, like so:

```
Number [] numbers = {
  new Integer(5), new Double(6.4), new Short((short)88)
};

double avg = average(numbers);
```

Is this useful? Well, as always, that depends. At this point it is more important for you to get the first inkling of the power of *polymorphism*, because that is what is at play here. Here it how it works:

When method `average` selects a `Number` from the `nums` array (at (1) in `average`), it calls the method `doubleValue()`. That method is declared in class `Number`, so the compiler accepts this. However, at runtime, with our code calling `average`, the first `Number` will be an `Integer`. Class `Integer` *overrides* method `doubleValue()` with its own version of it, and that is the method that is called.

When the second element in `nums` is accessed, it will be a `Double` that also overrides `doubleValue()`, but with a difference implementation of it. Similarly, the last element is a `Short`, and that will provide a *third* version of the `doubleValue()` method.

This is polymorphism (multiple forms and behaviours). The actual method and code that is being run to perform the work, is not decided by a selection statement like an `if` or a `switch` in method `average`. Instead, the code is *selected by the type of the implementing object.*

The general advantage of polymorphism is not limited to numeric data types. With clever class hierarchy design it can be used in many problem domains. The chief advantage though, is seen by the following scenario:

> You are working for a consulting company, writing a program that operates on data that comes in three different kinds. These kinds consistently require different operations. Your program code therefore has many places where you have to select what to do, and there are many `if else` statements to manage the delicate flow of control. But you put your shoulders to it, and you work it, and debug it and test it, and finally it *works.*
>
> The next day your customer comes and says: *"This is great, we love it! And by the way, we would like to add two more data types, the doofers and ding-dongs too, could you do that? By next week?"*
>
> Now you have to go in and add to all of those carefully crafted `if else` clauses, with a high risk of breaking the program. On the other hand, if you had designed the program to use polymorphism instead of explicit control statements, it is possible that all you would have

> to do would be to add two more classes to the object hierarchy. The objects themselves implement their special treatment, and the general code treats them uniformly.[2]

Returning to the concept of *object resources*, we see that the resource provided is the ability to combine different number types in a uniform computation. The *type*`value()` methods selects the proper data type conversion through the use of polymorphism.

11.1.4 `Integer.equals()`

The class resource provided by the method `Integer.equals` is also an example of polymorphism, just like in the previous section, and the discussion in 9.3.5. It is a method inherited from class `Object` and because of that it can be called in all objects to compare them. The `Integer.equals()` implementation decides what it means for one `Integer` to be equal another `Integer`. Intuitively we understand that they are equal if their internal `int` values are identical.

11.1.5 `Integer.toString()`

The method `Integer.toString()` is yet another example of a method inherited from class `Object`, and specialized by class `Integer` so that each `Integer` object can generate a meaningful text representation of itself. See also 9.1.4.

11.2 Class resources

Object resources belongs to object instances, the resources perform a service that is dependent on what data is stored in the object. Class resources, on the other hand, are static methods that provide a general service related to the data type implemented by the class, rather than a particular value of that data type. This section presents a few examples of this.

11.2.1 `String.valueOf(`***type***`)`

Class `String` has a group of overloaded static methods that provide default text representations of selected data types:

- `static String valueOf(boolean b)`

[2]Yes, this *is* a rosy epic, but the general idea is correct.

- `static String valueOf(char c)`
- `static String valueOf(char [] data)`
- `static String valueOf(char [] data, int offset, int count)`
- `static String valueOf(double d)`
- `static String valueOf(float f)`
- `static String valueOf(int i)`
- `static String valueOf(long l)`
- `static String valueOf(Object obj)`

As can be seen from this list, most of the primitive data types are present. Values that are `byte` or `short` are automatically converted to `int` by the compiler for a call to the `valueOf(int i)` method. Objects can be converted to strings as well; that method simply calls the object's `toString` method.

There are two methods that allows for the text representation of arrays of `char`. The first one converts the whole array to a string, the second one allows a subrange to be converted, to prevent the need to create and copy into a temporary array before conversion.

The composite class resource offered by these methods, is that *any* single value can be converted to a string. You don't even have to look at it, you just call `String.valueOf(`*value*`)`. In addition to that, extra resources are given to `char` arrays, as they often need conversion to strings.

11.2.2 `Integer.parseInt()`

The static method `int Integer.parseInt(String s)` is a class resource that helps in the standard conversion of a string of decimal digits into an `int` value. Similar methods exist in class `Long`, `Double`, and the other number wrapper classes. There is no polymorphism at play here, though, one must turn to the correct class to perform the desired kind of parsing. That does not in any way diminish the usefulness of these methods. If you have a number in a string and want it converted to a primitive data type, these methods can do it for you.

11.2.3 `Integer.MIN_VALUE`, `Integer.MAX_VALUE`

The constant `Integer.MIN_VALUE` contains the bit pattern of the smallest (most negative) value that can be stored in an `int`. Similarly, the `Integer.MAX_VALUE` constant holds the greatest (most positive) value that an `int` can hold.

Note that these `Integer` constants spans the full range of numbers an `int` can hold, from the most negative to the most positive. This is *different* from the floating point constants with the same name.

11.2.4 `Double.MIN_VALUE`, `Double.MAX_VALUE`

The constant `Double.MIN_VALUE` is the smallest, non-zero, *positive* number that a `double` can represent. The constant `Double.MAX_VALUE` is the greatest *positive* number that a `double` can represent.

Note that these `Double` constants only spans the positive range of numbers a `double` can represent. Since the floating point data types use a separate sign bit, the most negative number a `double` can hold is `-Double.MAX_VALUE`, and the greatest, non-zero, negative number in a `double` is `-Double.MIN_VALUE`. This is different from the `Integer` constants with the same name.

11.3 Classes in Java

As has been spoken of liberally in previous sections and chapters, all objects in Java belong to a class. The standard library contains many ready-made classes which a program can create instances from and use. The classes in the standard library also contain numerous static methods and constants which define how a certain class works, and provides support for using the class or the Java language.

A Java program is written as a set of classes. One of these classes contains the static `main` method where the program starts. For very small programs it sometimes suffice to use only static methods in the main program class, possibly with help from objects in the standard library.

A more ambitious program creates additional classes, classes that contain data relevant to the program, and equipped with methods that allow calling code to use it in a defined way. The data objects protect their data from outside manipulation using the idea of *encapsulation*. The public methods in the data object provide the designed interface to the object.

Design by *structured decomposition* helps in the creation of a full program. The idea is that the full task is divided in separate subtasks that can communicate with each other using minimal contact surfaces. Each task in turn is then further subdivided into units, that are as minimally integrated as possible with each other. This modular strategy requires a bit of design effort, but is worth it in the long run, because the modules can be independently maintained. Furthermore, with well defined and minimal interfaces, it is easier to see how a change in one class affects other classes that are using it.

```
                +------+
                | Rule |
                | text |
                +------+
                   |
                   V
+-------+     +----------+      +--------+
| Input |---->| Rewriter |----->| Output |
| text  |     |          |      |  text  |
+-------+     +----------+      +--------+
```

Figure 11.2: The rewriting program example

Here is a short example to illustrate the idea. Assume that you are developing a symbolic rewriting program. The program is to read a text, apply rewriting rules to the strings of characters in the text, and finally write the result out. The rewriting rules come from a separate file. Figure 11.2 gives a schematic example.

We can immediately begin to sketch on a program design in which one part is wholly devoted to finding the input text, be it from a file, from the standard input, from a URL, or something else that we might think of later on. We imagine that there will exist a class that solves all of the particulars on *how* to get the text, as long as it has a method we can call to get the next string of input. We sketch this out as class `RewriterInput`.

Having had that idea, it takes no effort at all to imagine that we could have a similar class, perhaps called `RewriterOutput`, for the output text. A class that we can create an object from, configure, and feed one string after another of rewritten text until we are done.

In the middle then, we will need the rewriter itself. It seems natural that this also is a class. Internally it may use its own instance of a `RewriterInput` to get the rewriting rules. That would save some programming effort. It will need some data structures to store the rules, and of course some methods that performs the actual rewriting, but we can also clearly see that we want it to have a method similar to:

```
public String rewrite(String input)
```

because that would be really neat. At this point we smile gleefully and can even sketch loosely on the main program:

```
...
RewriterInput ri = new RewriterInput(/* input source */);
Rewriter rwr = new Rewriter(/* rule source */);
RewriterOutput ro = new RewriterOutput(/* output target */);

while (ri.hasMoreInput()) {
```

```
+----------------------+
|  class A {           |
|    ...               |
|  }                   |
+----------------------+
           |
           |
           |
+----------------------+
|  class B extends A { |
|    ...               |
|  }                   |
+----------------------+
```

Figure 11.3: Simple class inheritance

```
    ro.write(rwr.rewrite(ri.readInputLine()));
  }

  ri.close();
  ro.close();
```

Such a main program would be short and sweet. In order to have it, we need to design and implement three classes. We begin working on the design of the `RewriterInput` because it will also be used by the `Rewriter` class.

The example ends here, but it should be clear that this top-down approach is a viable one, at least when the task is easy to understand. The difficult thing is to envision the boundaries between the different parts of the program, and as development moves forward it is not uncommon to go back and modify them. In the rewriter example, the boundary between the three working classes was a simple string of text. Real applications are often a little more complicated than that. That does not, however, invalidate the principle.

11.3.1 Inheritance

Inheritance is the mechanism whereby a new class is built by extending a class that already exists. The new *subclass* inherits all non-private methods and class variables of the *superclass*. The subclass can *override* methods in the superclass by providing new versions with identical signatures, and *overload* methods by supporting additional signatures for the same method name. It can also, of course, contribute new methods that are independent of the superclass and appropriate for the task at hand.

Figure 11.3 illustrates the relationship between the superclass and the subclass. Class B `extends` class A (inherits from class A). Class B is a subclass of class A. Class A is the superclass of class B.

Every class in Java has exactly one superclass (except `Object` which is special). In figure 11.3 the superclass of class A is `Object`. A class can only extend (inherit from) a single superclass.

We add some instance variables to the example:

```
// In file A.java
class A {
  public int u;      // Visible to all classes
  int d;             // Visible to package and subclasses
  protected int r;   // Visible to subclasses
  private int p;     // Visible only to class A
}

// In file B.java
class B extends A {

}
```

Look at the listing above; class A declares four `int` variables with different levels of access protection. The variables are `u, d, r,` and `p`. When class B `extends` class A, class B inherits all four variables. When a program executes the statement:

```
B bobj = new B();
```

the object returned by the `new` operator will contain memory for *all four variables.* Code in class B can manipulate `u, d,` and `r` directly, as in:

```
u = r + d;
```

Class B does not have to declare these inherited variables, they are already declared in class A, and since class B extends from A, they are immediately available.

Class B will also inherit and contain memory space for the private variable `p`. Because it is declared as `private`, code in class B cannot access it directly. It is not visible in the scope of class B, but it is there. If class B itself declares a variable `p`, then that will be a different variable. The only way for class B to manipulate `A.p`, is if class A has prepared some non-private methods for that purpose.

This *vertical encapsulation* is useful in that it protects the superclass from inadvertent or incompetent modification. Imagine that class A has been carefully developed, debugged, and tested, and is made ready for use in a greater project. Along comes a programmer who does not fully understand how A works, and extends it with code that breaks A. To prevent this, the writers of class A use the private access protection to say: *"don't mess with this, and if you are trying to, you are approaching the problem the wrong way."*

For private methods, the thinking is similar. A subclass cannot call them and it cannot override them. A non-private method declared as `final` cannot be overriden. A class declared as `final` cannot be extended.

A method declared as `abstract` is intentionally only a signature, it has no body. When a class contains an abstract method, then the class becomes abstract too, and should be declared as such[3]. Abstract methods are placeholders left to programmers. The abstract methods are on purpose left undefined, so that subclasses *must* override them with concrete methods. If they do not, they become abstract too, and one cannot create objects from abstract classes.

A quick example demonstrates the thinking behind abstract classes. Assume a graphics program with the classes `Triangle`, `Oval`, and `Rectangle`. These classes represent the concept of a basic *shape*. Each class also has a method `render(Graphics g)`, which renders that shape. It is convenient to have a common superclass for the basic shapes, so class `Shape` is designed. But a you cannot create a `Shape`, and you cannot render it, so it should be abstract:

```
// In file Shape.java
abstract class Shape {
  protected int x, y, width, height;
  public abstract void render (Graphics g);
}

// In file Rectangle.java
class Rectangle extends Shape {
  public void render (Graphics g) {
    g.fillRect(x, y, width, height);
  }
}

// In file Oval.java
class Oval extends Shape {
  public void render (Graphics g) {
    g.fillOval(x, y, width, height);
  }
}

// In class Triangle.java
class Triangle extends Shape {
  int [] xcor = new int[3];
  int [] ycor = new int[3];
  public void render (Graphics g) {
    g.fillPolygon(xcor, ycor, 3);
  }
}
```

The subclasses to `Shape` all override the `render` method, implementing how to render their particular shape.

[3]The `abstract` declaration in the class informs a reader of the source code that one or more abstract methods are present. It is also a crosscheck that the compiler helps to verify.

```
                  +-------+
                  | Shape |
                  +-------+
                      |
         +------------+---------------+
         |            |               |
     +------+    +-----------+    +----------+
     | Oval |    | Rectangle |    | Triangle |
     +------+    +-----------+    +----------+
```

Figure 11.4: Simple class inheritance

Figure 11.4 shows the class hierarchy of abstract class `Shape` and its concrete subclasses. The abstract class `Shape` is the superclass of all the concrete subclasses. Since every subclass *is* a `Shape`, we can then use polymorphism to create a uniform treatment when drawing mixed shapes:

```
Shape [] shapes = new Shape[3]; // Array of shape

shapes[0] = new Oval(...);      // Constructor parameters are
shapes[1] = new Rectangle(...); // not shown here
shapes[2] = new Triangle(...);

// Draw all shapes using polymorphism
for (Shape sh : shapes)
  sh.render(g);                 // (1)
```

Since the `Oval`, `Rectangle`, and `Triangle` each is a `Shape`, they are valid elements in an array of `Shape`. At line (1) in the listing, a method call is made to the abstract method `Shape.render`. But, since it is not possible to create an object out of an abstract class, *every non-null reference in variable sh must refer to a concrete class — a class that overrides method render with a concrete method.* At runtime, the method call to `Shape.render` will be redirected to the method in the current object.

11.3.2 The class hierarchy

The Java class hierarchy forms a tree that is rooted in class `Object`. All classes are therefore subclasses to `Object` and inherits its methods. The two most interesting ones are `toString` and `equals`.

A class can only extend a single superclass, and if the class declarations omits the `extend` clause, the class will be located (conceptually) immediately below class `Object`. If the class extends some other class, like for example class `Shape`, then its position in the class hierarchy is below class `Shape`, and it will *be* both a `Shape` and an `Object`.

Here are some declarations to illustrate this important relationship further

```
Object
|
+--Shape
|    |
|    +--Rectangle
|    |
|    +--Triangle
|    |
|    +--Oval
|         |
|         +--Circle
```

Figure 11.5: A vertical class hierarchy

(only the important code is shown):

```
class Shape {                // Shape is an Object, by default
  ...
}

class Oval extends Shape {  // Oval is a Shape, and an Object
  ..
}

class Circle extends Oval { // Circle is an Oval, a Shape, and
  ...                       // an Object
}
```

The class hierarchy in figure 11.4 is horizontal. For listings and printings it is often more convenient to use a vertical format, as shown in figure 11.5.

Chapter 17 continues the account of inheritance.

11.4 Autoboxing and unboxing

The wrapper classes in the standard library (e.g. `Integer`, `Double`, `Character`) can be used to *wrap* a single value of a primitive data type in an object. This obviously requires some extra coding:

```
Integer ii = new Integer(42);
```

To get the value back, one also has to reach into the object and retrieve the value with a method call:

```
int i = ii.intValue();
```

The Java compiler is well aware of this relationship between the wrapper classes and the primitive data types they support. The compiler can therefore understand when an instance of a wrapper class is used like a primitive value, and automatically inserts code to achieve the intended meaning. As a consequence it is legal to write:

```
Integer ii = 42;
```

When the compiler encounters this, it sees an attempt to assign an `int` value to a reference variable. This is normally illegal. However, since the reference type is of class `Integer`, the wrapper type for `int`, the compiler automatically inserts code equivalent to:

```
Integer ii = new Integer(42);
```

This is called *autoboxing*; the compiler pulls up a box of `Integer` and wraps the `int` value 42 inside it.

Similarly, when an attempt is made to use a reference to a wrapper instance as if it was a primitive value, the compiler can automatically insert the required code to find the expected value. This is called *unboxing* and it means that the following code is allowed:

```
int i = ii;
```

Only simple assignments are shown here, but in general autoboxing and unboxing works as intuitively intended, and can be used in parameters, expressions, and when using collections[4].

11.5 Arrays of references to objects

As was shown in the example of the shapes (figure 11.4), it is quite possible to have an array where the element type is a reference type. In such cases each element of the array contains not an actual object, but a reference to an object. This is an important distinction from arrays of primitive data types.

We can, for example, create an array of strings that does not contain any string at all:

```
String [] words = new String [4];
```

The array is created, memory is allocated, but so far each element in the array is `null`. We need some (references to) strings that can be placed in the array:

```
String alan = "ALAN";
String takes = "TAKES";
String a = "A";
String cookie = "COOKIE";
```

[4]The package `java.util` has a number of *collection* classes. These classes provide lists, sets, and maps.

```
words-->[ ][ ][ ][ ]
         |  |  |  |
         |  |  |  |
         |  V  V  |
         +>"ALAN"<+
```

Figure 11.6: Array elements that refer to a single string

These references to strings can then be assigned to elements in the `words` array:

```
words[0] = alan;
words[1] = takes;
words[2] = a;
words[3] = cookie;
```

If we were to print the strings in the `words` array, we could use the following code that inserts a space between words:

```
for (String w : words)
  System.out.print(w + " ");
```

So, with the original assignments we get the message:

```
ALAN TAKES A COOKIE
```

It should also be clear that we can move elements around and create different printouts (this is a roll left by one):

```
String first = words[0];
for (int i = 0; i < words.length - 1; i++) {
  words[i] = words[i+1];
}
words[words.length - 1] = first;
```

After this code the printout becomes:

```
TAKES A COOKIE ALAN
```

Since the array elements just contain references to string objects, any element can refer to any string. Thus, it is trivial to arrange the assignments of the elements so that a word is repeated. Figure 11.6 shows the reference structure that generates the following printout:

```
ALAN ALAN ALAN ALAN
```

11.5.1 Matrices

As was described in 5.2, multidimensional arrays are basically one-dimensional arrays where the elements refer to other one-dimensional arrays. Since an array is an object, multidimensional arrays are also arrays referring to objects.

Matrices consist of a one-dimensional vector where each element refers to a one-dimensional vector of the leaf data type. If we remember the example with strings in the previous section, we can begin to see the wider possibilities offered by a network of references. Consider, for example, the following:

```
  Object [][] mox = Object [2][2];
```

The matrix referenced by `mox` is a matrix of reference to `Object`, i.e. any class since each object instance *is* an `Object`. It can be visualized like this:

```
  mox
   |
   V
  ---
 0[ ]--->[[null][null]]
  | |
 1[ ]--->[[null][null]]
  ---
```

We apply the following assignment:

```
  mox[0][0] = "foo";
  mox[0][1] = "bar";
```

and the structure is changed like so:

```
  mox
   |      "foo"  "bar"
   V        ^      ^
  ---       |      |
 0[ ]--->[[    ][    ]]
  | |
 1[ ]--->[[null][null]]
  ---
```

The element type of `mox` is reference to array of object. As a result two elements can contain the same reference, just like the `words` array above contained several references to `"ALAN"`:

```
  mox[1] = mox[0];
```

The resulting matrix, however, is non-standard and degenerated:

```
  mox
   |      "foo"  "bar"
   V        ^      ^
  ---       |      |
 0[ ]--->[[    ][    ]]
  | |    ^
 1[ ]----+
  ---
```

While the length of `mox` is two, there is now only a single row in the matrix. A change made to `mox[0][0]` or `mox[1][0]` is immediately reflected in the other. The same holds for `mox[0][1]` and `mox[1][1]`, because there are now two routes to the same element.

Finally, with the element type of `mox[]` being `Object`, we can perform yet another perversion of the data structure and do:

```
mox[0][0] = mox;
```

This gives the reference structure:

```
 mox
  |+--------+     "bar"
  VV        |       ^
 ---        |       |
0[ ]--->[[     ][     ]]
 | |    ^
1[ ]----+
 ---
```

The matrix `mox` now contains a reference to itself, and not only in `mox[0][0]`, but also in `mox[1][0]`. This is called a *circular* reference and is usually considered to be a *bad thing*, because it easily invites path traversal to get stuck in an infinite loop.

Hopefully it is obvious that these examples are taken from the horror cabinet of programming. They are, however, a consequence of the possibilities allowed us by the use of references. References and graph structures are a very powerful tools, and they must be employed with care. When properly utilized, they serve us best. When abused they turn on us, and the program becomes a beast.

11.6 An array as a parameter

When passing parameters to a method, we can assign a single value to each parameter variable. But it is often the case that we want to the method to process a variable number of arguments. We can solve this by passing a reference to an array as a single parameter. The array contains all the objects to be processed.

11.6.1 `java.util.Arrays.sort`

The standard library class `java.util.Arrays` contains a static method `sort` that sorts an array in ascending order. The method is overloaded to care for arrays of the primitive scalar data types, because these can be easily compared. In addition to those, there is also a version of method `sort` that can sort an array of `Object`. The only requirement is that *all elements in the array implements interface* `Comparable`.

The `Comparable` interface is, simply put, a promise that the implementing class contains the method:

```
int compareTo(T o)
```

When a class implements this for some type `T`, it can compare itself against some other object `o` and then return a value smaller than zero if it is less than `o`, a value greater than zero of it is greater than `o`, and zero if it considers itself equal to `o`. So, when all elements in the array implement interface `Comparable` then `java.util.Arrays.sort` can sort them without knowing *how* they are compared. It is sufficient that the objects themselves can decide which of two is the lesser one.

There is yet another version of `sort`, and that is for those situations when all objects do *not* implement `Comparable`. In this case the caller must provide a `Comparator`. This is, in simple terms, an object that implements interface `Comparator`. By implementing the interface, it provides the method:

```
int compare(T o1, T o2)
```

and again the return value is less than zero if `o1` is lesser than `o2`, and so on. The sorting algorithm can now pick two elements from the array to be sorted, ask the provided `Comparator` which one is the lesser one, and continue with the sorting.

For more information about interface classes, see 19.2 and 20.

11.6.2 Parameters to main

The `main` method has the following signature:

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

The main method receives commandline parameters as an array of `String`. Commandline parameters are given to the `java` program when it is launched and are those not recognised and consumed by the JVM itself. Commandline parameters are easy to provide if the Java program is run from a commandline. Just add them after the `java` command:

```
java Rewriter rules.txt Greek.txt TXGreek.txt
```

The program Rewriter would then receive in the `main` method the following array of strings:

```
args
 |
 V
---
[ ]--->"rules.txt"
[ ]--->"Greek.txt"
[ ]--->"TXGreek.txt"
---
```

It should be remembered that the Java language has no way of knowing how the commandline strings are composed, or where they are split into different strings. This is something the invoking shell does *before* starting the program.

In the Eclipse IDE the commandline parameters are set in the *Run Configuration*; a set of parameters accessible from the Run menu. This is a bit trickier to manage, but on the other hand allows for a more consistent control. In any event, Eclipse is the development environment, and should not be used to run production software.

11.7 Algorithms and objects

Algorithms exist for many different purposes, and with varying degrees of simplicity and efficiency. Usually the applicability of an algorithm on a certain dataset is not predicated on the particular dataset, but instead if it *exhibits the properties that are required by the algorithm.* A very good example of this is the array sorting algorithm briefly reviewed in section 11.6.1. That same sorting algorithm can applied to many different data types and objects, as long as they have the property that they can be ordered relative to each other. That is all that a general sorting algorithm needs.

Other algorithms may require a different set of properties from an object. Sometimes it is trivial to equip the objects with such properties. At other times an intermediate data structure must be employed. The problem is exacerbated by algorithm descriptions that for reasons of simplicity often are expressed using what appears to be very native data representations. In such cases the programmer must fully understand the algorithm in order to see how it can be applied to the dataset at hand, or indeed, if a different data representation works better.

Consider for example the update algorithm for finding the smallest element in a dataset (below is pseudocode):

```
let c = $d_0$   // Initialize c to the first element
let i = 1    // Search from the second element
while i < sizeOf($d$) {
  if $d_i$ < c
    c = $d_i$   // Update to smaller element
  i = i + 1
}
```

The algorithm is very simple: the first element is chosen as the initial candidate, because if there only exists one element in the dataset then it must also be the smallest. Then the remaining elements are examined, and as soon as an element smaller than the current candidate is found, the candidate is updated to remember the smaller one instead. When all elements have been examined, the candidate must be one of the equally smallest elements in the data set.

There are actually two aspects to consider when we want to use the update algorithm in a program:

- Can the dataset be accessed by an index?
- Can the elements be pair-wise compared?

If those two conditions are fulfilled, we can use the algorithm[5]. Let us consider some ways of using it. First out is an array of integers:

```
int [] nums = ...; // an array of integers numbers

int c = nums[0];   // init candidate to first element
for (int i = 1; i < nums.length; i++)
  if (nums[i] < c)
    c = nums[i];
```

This is a fairly straight-forward implementation of the update algorithm. The array provides the indexing, and the operator `<` provides the comparison of elements. It is also fairly trivial to modify it to deal with the other primitive scalar data types.

Now let us consider an array of `String`. The definition of a smaller string is likely to depend on the needs of the application, but for our purposes we will rely on the `Comparable` implementation in class `String`. Thus:

```
String [] sa = ...; // an array of strings

String c = sa[0];   // init candidate to first element
for (int i = 1; i < sa.length; i++)
  if (sa[i].compareTo(c) < 0)
    c = sa[i];
```

It is the same algorithm, except that we had to change the comparison from the `<` operator to a method call.

Let us now see if we can make a general implementation. We establish the requirements that the data structure must implement the interface `java.util.List`, and that the elements in the list implement the interface `java.lang.Comparable`:

[5]Actually, we do not strictly need an index to find the smallest element; all we need is a guarantee that we can read every element. But that is a slightly different algorithm.

```
import java.util.List;

List<Comparable> lst = ...; // a list of comparable objects

Comparable c = lst.get(0); // init to first element
for (int i = 1; i < lst.size(); i++) {
  Comparable e = lst.get(i); // get element i
  if (e.compareTo(c) < 0)    // compare
    c = e;                   // update
}
```

This implementation is still the same algorithm, but it is no longer tied to a particular data type, as long as our requirements are fulfilled. One way of viewing this, is that instead of writing many different implementations of the algorithm, each adapted to a particular data type, we present a general solution to which *the data must adapt.* Now, that said, this viewpoint is only fertile for complicated algorithms that take time to implement and debug. The update algorithm is so small that it should be implemented where it is needed; there no gain in generalizing it. But for other algorithms that require more lines of code, like sorting and searching, the gain in having a general implementation that can be reused is far better than lots of specialized cases.

Chapter 12

Exceptions

Exceptions were previously covered in section 7.3. This chapter repeats and extends this discussion.

12.1 Managing exceptions

Exceptions are used when a program enters an *exceptional* state. This is a state which the program was not designed to handle, or from which it cannot continue safely. A typical example could be that the program tried to read from a file, but the file is not found. This would indicate that the program itself is correct, but was given the wrong filename, or that the file is missing from the environment in which the program operates.

Another typical exception is the `NullPointerException` which is thrown by the JVM when the program attempts to follow a reference to an object, but the reference is the `null` reference. This usually indicates a logic problem in the program; a failing assumption on behalf of the programmer.

Exceptions can often be placed in two categories: those that are anticipated, and if they are thrown they can be dealt with and the program can continue to execute. A typical example is that the user enters some mal- or misinformed data: a file that does not exist, a host that can not be found, a string of characters that cannot be parsed into a number.

With fallible human beings it is more or less expected that mistakes will happen, so the aware programmer prepares extra code to catch the exception, present the problem to the user, and ask again. The *defensive* programmer goes a step further, and instead writes code that *checks* the input before attempting to use it. Does the file exist? Can that host be reached? Is there a number in that string? With defensive coding, expected and trivial data errors do not need the exception machinery. They are processed normally by the program, and exceptions are used for truly exceptional states.

Truly exceptional states then, can be traced back to these situations:

- A bug in the program
- Unexpected non-interactive input (data in a file)
- An error in the JVM or the environment, like a missing disk drive, or hardware failure.

The first two cases are connected to each other in subtle but interesting ways, that reflects upon how they should be treated, and how exceptions can help with that.

A bug in the program means that it does not do what the programmer intended. This is almost always because the programmer has made a mistake or did not fully understand the programming language. The computer will normally do exactly what it was told to do. For example, perhaps the programmer wants to see if a string contains the text `"yellow"` and is using the `==` operator (incorrect) instead of the `equals` method (correct). This may lead to an inconsistency in the program where one part concludes that a dataset contains no yellow elements, while there actually are some in there. Such inconsistencies between programmer assumption and actual program state, may lead to corrupt output or exceptions like the `NullPointerException`.

A less obvious problem is if a program has been designed to read input data from a file of records, and expects a certain data field to contain `"SE"`, `"DK"`, or `"NO"`. Then one day comes a data field that contains `"FI"`, and the program does not know what to do. Is this an error in the input data, a typo, or is it a new kind of data record which the program has not been updated to handle? This cannot be resolved by the program, and it must report the situation, and stop, or if it can, proceed with the next data record.

Errors in the JVM are impossible to protect against. Luckily, they are also very rare.

When an exception happens the program probably wants to end with an error report. But before it can do so, it may want to finish up neatly. There may be files open, or database connections that are not involved in the exceptional state. These should be closed normally, to avoid the risk of resource leaks. The program may want to make a logfile entry, detailing the state at which it closed, so that the operators can learn the most from the situation. Consider that many programs are run automatically, as part of larger systems, with no human at the console. Logfile forensics is an art in itself, which we will not go into here.

12.1.1 Creating, throwing, declaring

Exceptions are created with the `new` operator, just like ordinary objects. Exceptions *are* ordinary objects, with the distinction that the exception machinery recognise them.

To create an exception, we can use exception classes from the standard library:

```
throw new IllegalArgumentException("value too large");
```

It is often more useful to create new exceptions that fit the program requirements better:

```
public class TemperatureException extends Exception {
  public TemperatureException(String s) {
    super(s);
  }
}
```

12.1.2 Catch and manage

When executing code that may throw an exception, the programmer is faced with two options. Either, the exception is not managed, and is allowed to continue up the call chain, possibly all the way up and out of the `main` method, resulting in the termination of the program. Or, the exception is *caught* and managed. This is done with a `try catch` clause.

Let us assume that we have a program that is reading records from a file. The file comes from a remote source so it is possible, but not very likely, that the data in it contains all sorts of syntax errors and inconsistent information. We call our method `parseNextRecord` to process the next record from the file. If the method finds any bad data, it throws an exception that we have declared for this very purpose and named `AppDataException`. This exception we want to catch, so that we can make a log entry before attempting the next record.

Since our method `parseNextRecord` reads data from a file, it is possible that some kind of `IOException` will also be thrown, unrelated to the contents of the data file. This is both unusual and unexpected and we do not want to catch that, but rather pass it on up the call chain.

In the example below, `AppData` is a class we have created to represent the input data file. One of its services is to keep track of how many records we have read. The `PrintWriter` object referred to by parameter variable `logfile`, is an open output stream to the current logfile. We arrive at the following code:

```
import java.io.IOException;
import java.io.PrintWriter;
```

```
void parseAllRecords (AppData ad, PrintWriter logfile)
  throws IOException
{
  while (ad.hasNextRecord()) {  // (1)

    try {
      parseNextRecord(ad);      // (2)
    }
    catch (AppDataException aex) { // (3)
      logfile.println("Record " + ad.currentRecord() +  // (4)
                      ":" + aex.toString());
    }

  } // while
}
```

The calling code opens the input data file and creates the `AppData` object. It also opens the logfile and wraps a `PrintWriter` around it. Then it calls `parseAllRecords`, passing the references to the `AppData` and `PrintWriter` objects.

Method `parseAllRecords` will attempt to read all records from the input data file. It does this, using a `while` loop (1). Inside the `while` loop, it expects that there may be an `AppDataException` coming from the call to `parseNextRecord` (2). For this reason, the call to `parseNextRecord` is wrapped inside a `try catch` clause. The code in the `try` block can be any number of lines, but it is just the one method call in this example.

When an exception is thrown inside the `try` block the exception machinery immediately begins to scan the `catch` clauses for one that matches the exception thrown. There can by any number of `catch` clauses, to cater for different exceptions. If a match is found, the local variable in the `catch` clause is assigned the current exception object, and the code inside the `catch` block is executed (4). Note how the `toString` method is used to retrieve the message from the exception instance, a message that was inserted by method `parseNextRecord` when it created the exception.

In our example only instances of `AppDataException` can be matched. The exception has now been *managed*, and when the `catch` block exits, execution will continue normally from after the whole `try catch` clause. In the case of our example, this point is the end of the `while` loop, so the program will continue at the top of the loop.

If no match for the exception can be found among the `catch` clauses, the exception will continue upwards to the caller. This is indeed the desired behaviour in our example if there is an `IOException`.

12.1.3 Catch at the right level

Dealing with exceptions can be messy. In practice, the example above is as complex as you ever would like it to be. Longer clauses, nested clauses, all quickly contribute to hide the logic of the working code behind the trees of the exception management.

The example also illuminates a division of labour that is connected to units of work. We only manage the `AppDataException` because it concerns a single record; a single unit of work. An `IOException`, on the other hand, concerns a different unit, the whole file. Therefore it is better handled on a higher level, the level where files are selected, opened, and closed.

The exception system is a mixed blessing. It is far better to have, than not to have, like in the programming language C. On the other hand, it quickly gets clunky when dealing with more complicated exception patterns. For this reason it is advisable to attempt a vertical separation of exception management, and if possible keep the code inside the `try catch` clause to a minimum.

One particular important point is illustrated like so:

```
try {
  $s_1$
  $s_2$
  $s_3$
  $s_4$
}
catch ($ex$) {
  ...
}
```

Here, when execution enters the `try` block, statements will be executed sequentially. First s_1, then s_2 and so on. If no statement throws an exception, then all statements will be executed. If a statement does throw an exception, then *the remaining statements will not be executed.* For example, if statement s_2 throws an exception, then statements s_3 and s_4 will *not* be executed.

This leads to subtle dangers. Assume that statements s_2 and s_3 both are capable of throwing the same type of exception, an exception that is handled in a `catch` clause. When the exception is thrown, and the `catch` clause is entered, it is not obvious if it was s_2 or s_3 that threw the exception. This could make it very difficult for the managing code in the `catch` clause, especially if a proper recovery depends on knowing if s_2 was executed or not. In such situations the program must rely on additional state variables to be able to determine what to do. It is proabably better to define separate exceptions and have separate catch clauses to immediately be able to distinguish the two cases.

```
java.lang.Object
  |
  +--java.lang.Throwable
       |
       +--java.lang.Error
       |    |
       |    ...
       |
       +--java.lang.Exception
            |
            +--java.io.IOException
            |    |
            |    +--java.io.FileNotFoundException
            |    |
            |    +--java.net.UknownHostException
            |    |
            |    ...
            |
            +--java.lang.RuntimeException
            |    |
            |    +--java.lang.ArithmeticException
            |    |
            |    +--java.lang.IllegalArgumentException
            |    |
            |    +--java.lang.NullPointerException
            |    |
            ...  ...
```

Figure 12.1: The exception class hierarchy root

12.2 The exception hierarchy

Exception classes are ordered by a hierarchy. The root of this hierarchy is shown in figure 12.1. Class `Object` sits at the top (as always) and below that is `Throwable`. This is the common superclass of `Error` and `Exception`. A program is not expected to declare or catch an `Error`; they are too severe exceptions. Below `Exception` several subtrees can be found. Of these only two are shown, `IOException` and `RuntimeException`. A `RuntimeException` is *unchecked* by the compiler and can therefore be left undeclared by methods. The other exceptions are *checked*, and the compiler will require that throwing methods declare them.

12.2.1 Sequnce ordering in catch

When an exception has been thrown and the exception machinery is searching through `catch` clauses for a match, it will select the first one that matches. The search is done from top towards bottom. Comparison is done by type. If the thrown exception is of the same type *or a subtype* as the type declared in the `catch` clause, then it is considered a match.

This means that in general, `catch` clauses should start with the specialized types and use more general types towards the bottom of the `catch` clause list.

As an example, assume we want to catch all `IOException`, but with special treatment for an `UnknownHostException`. Look at figure 12.1 and confirm that an `UnknownHostException` is a subclass of `IOException`. Here his how *not* to write this code:

```
try {
  ...
}
catch (IOException iox) {
  ...                             // (1)
}
catch (UnknownHostException uhx) {
  ...                             // (2)
}
```

When an `IOException` is thrown, the first comparison will be against (1), and since it is a match, that clause will manage the exception. However, when an `UnknownHostException` is thrown, it will also match (1), because the `UnknownHostException` *is* an `IOException`. The code under clause (2) will never be reached.

The correct code is therefore to place the special case before the general case, like so:

```
try {
  ...
}
catch (UnknownHostException uhx) {
  ...                             // (1)
}
catch (IOException iox) {
  ...                             // (2)
}
```

In this code, the `UnknownHostException` will be tested against the more special condition first, and if there is a match then (1) is executed. Otherwise, if it is some other kind of `IOException`, clause (2) will manage that.

From Java 7 it is possible to use disjunctions in `catch` blocks:

```
try {
  ...
}
catch (FileNotFoundException | NoSuchHostException ex) {
  ...
}
catch (IOException iox) {
  ...
}
```

The vertical bar character is used to separate the exception classes to match against, and the variable is placed at the end. The advantage of the disjunctive pattern is that it removes the need to repeat code in specialized `catch` clauses, code that is different from the catch-all general case.

12.3 Nested try catch clauses

It is possible, but therefore not recommended, to place a `try catch` clause inside another `try catch` clause. For example:

```
try {
  $s_1$
  try {
    $s_2$
  }
  catch ($ex_2$) {
    ...
  }
}
catch ($ex_1$) {
  ...
}
```

In this code, statement s_2 is guarded by the inner `try catch` clause. If statement s_2 throws an exception that is caught by `catch` ex_2, then the outer `try` block will continue to execute normally. It will not see exception ex_2, since that was caught and managed by the inner `try catch` clause. If the inner `try catch` clause does *not* catch the exception, then it will continue to be thrown outwards, and reach the outer `try catch`. The exception will be compared against ex_1 and possibly caught there.

The reason why nested `try catch` clauses are not recommended, is that the code overhead quickly grows beyond what can be easily appreciated. It becomes difficult to write, to read, and to understand what is actually happening.

12.4 try catch finally

There is an optional third component available with the `try catch` clause. This is the `finally` section, a block of code that if present will *always* be executed, regardless of if an exception was thrown or not. The purpose of the `finally` block, is to allow for things that needs to be done when the operations inside the `try` block has finished, regardless of if it finished normally or because of an exception. Typical examples involve the release of resources that were allocated before the `try` block was entered, such as open files, open database connections, or large blocks of memory.

Here is an example using `finally`:

```
import javax.sql.Connection;
...
Connection myDB = null;

try {
  myDb = ...; // Create and open database connection
  ...         // Make transactions
  myDB.commit(); // Commit changes
}
catch (SQLException sqx) {
  myDB.rollback(); // Undo incomplete transaction
}
finally {
  myDB.close();    // Always close the connection
}
```

The `finally` block always executes. If the `try` block exits normally, control is transferred to the `finally` block. If there is an exception, control is transferred to the first matching `catch` block (if there is one), and then to the `finally` block. If there is no matching `catch` block, then the `finally` block is executed before the exception is thrown further outwards or upwards.

12.5 try-with-resources

A recent (Java 7) addition to the `try` block is a construct called *try-with-resources*. Rather than using the procedural `finally` block, try-with-resources takes a declarative stance and allows the use of resources (object instances) that can be automatically closed. Such objects implement the interface `AutoClosable`:

```
void close() throws Exception
```

The `AutoClosable` interface was made the superinterface of many preexisting interfaces, so most of the things in the standard library are, as a result, usable in a try-with-resources.

In a try-with-resources block, the `try` block is entered with `AutoClosable` resources. When the `try` block exits, the resources are automatically closed. This means that the method `close` is called in each resource (in reverse declaration order). This happens regardless of if the `try` block exits normally or due to an exception.

Here is the schematic view:

```
try (resource list) {
  ...
}
```

The *resource list* is a sequence of local variable declarations with initializations. For example:

```
import java.util.Scanner;

try ( Scanner tx = new Scanner(new File(infile));
      PrintWriter pw = new PrintWriter(outfile) ) {
  ...
}
```

It should be noted that the try-with-resources clause does not by itself manage exceptions. All it does is to ensure that the resources used in it are closed when the `try` block is exited. To manage exceptions, `catch` blocks can be added after the try-with-resources:

```
import java.util.Scanner;

try ( Scanner tx = new Scanner(new File(infile));
      PrintWriter pw = new PrintWriter(outfile) ) {
  ...
}
catch (InputMismatchException imx) {
  ...
}
```

12.6 Rethrown exceptions

Having caught and managed an exception does not necessarily mean that all is well and done at that point. It may be that catching the exception was important to make a log entry, clean up, show an error message, and so on. The program is still not able to recover, and it needs the exception to be thrown upwards, even though it has already managed it. This is easy to do, simply `throw` the exception again, or perhaps throw a different exception:

```
try {
  ...
}
catch (IOException iox) {
  logger.severe(iox);
  throw iox;
}
```

The `throw` statement will rethrow the exception from that point, out- and upwards as appropriate.

12.7 Defensive coding

As was stated initially in this chapter, in section 12.1, one should as far as possible avoid using the exception machinery as a normal part of the operations in a program. Exceptions should be rare, and truly exceptional. The reason for this is that the exception machinery comes with an overhead; normal execution is suspended, `catch` clauses must be searched, and normal execution resumed. If this is done for every other record in a file of one million records, the extra time adds up, and the program runs slower than necessary.

In addition to the overhead incurred by exceptions, the code becomes cluttered with `try catch` blocks, and the sequence of control is more difficult to see. Just like with strong drink, exceptions should be used responsively, to catch those situations that are not normal, and that you really want to know about when they happen.

An interactive program that communicates with a human user should expect, and be prepared to deal with, malformed or inconsistent input. The software should be written to verify that input is useful and sensible, before it attempts to work on it. A graphical user interface should guide the user towards the legal alternatives. If two choices are mutually exclusive, then the one selected disables all controls that belong to the other alternative. If a text-field is supposed to contain an email address, then verify that it is a syntactically legal email address before accepting it. If another field allows the input of an integer number, only accept digits, and so on.

This strategy of scrutinizing the input before computing on it is called *defensive coding.* It brings a little more effort to the programming, but it makes for a more robust program. When something is wrong, the program knows it, and it can tell the world about it, or it can substitute a working default or approximation, rather than falling over on its virtual back like a cadaver struck by lighting. At the same time, judicious use of exceptions makes it possible to gracefully recover and retreat when something truly exceptional happens.

Chapter 13

In- and output

13.1 Streams

Input and output (I/O) in Java is built around the concept of serial streams. When receiving input, you are reading sequentially from a stream of uniform items, one after another. For example, you read the characters in a text file, from the beginning of the file towards the end.

When doing output, you write to an output stream. If it is an output stream of characters that goes to a file, then each new character is appended to what was written before it.

Streams separate the sequential reading and writing of data units, from the units themselves. You read one entire unit, or you write on entire unit. There are never any partial units read or written. The stream takes care of this, for you.

This discrete behaviour of a stream, makes it possible to have streams, not only of bytes or characters, but also of objects. An entire object instance can be saved in a file, and then later read back, using an object stream.

13.1.1 Buffered streams

Hard disks and network connections are from a communications point of view, *block* devices. This means that they have a lower limit on the smallest unit of data they operate on. For hard disks this block size is often 4 Kbyte (4096 bytes). For networks it can be as small as 56 bytes.

A block device is more efficiently used when data going towards it, is *buffered* in main memory until a complete block is filled. This greatly reduces the number of exchanges between the operating system and the device.

If you like a real-world analogy to the block device, consider a small ferry used to cross a river. The ferry can carry 20 people. In the morning, people arrive randomly but consistenly at the jetty on the residential side of the river, in order to travel to work on the other side. If the ferry was to cross the river as soon there was someone waiting, it would have to run without pause, taking only a few people on each trip. This would obviously be very wasteful. On the other hand, if it waited until 20 people or more were assembled on the jetty, each trip would utilise the ferry to its maximum efficiency, resulting in fewer trips and less total fuel consumption.

Because of block devices, *buffered streams* are often used when writing to files. Buffered streams reduce the number of disk operations needed to write the file, and therefore increase overall efficiency. The only caveat is that the program must remember to `flush` (write the buffer to disk *now*) the stream when it wants to make sure that data has been committed to disk.

A buffered stream is an ordinary stream with a buffer wrapped around it. Data is sent to the buffered stream object, which then stores data in its internal buffer. When the buffer is full, or an explicit `flush` is requested, the data in the buffer is sent to the underlying stream as a block of data. This often improves performance downstream, at the possible cost of a slight delay between when the program writes the data and when it reaches the end of the stream.

13.1.2 Byte streams: writing binary data

A byte stream is, as the name implies, a stream of bytes. The unit is the primitive data type `byte`. This is the smallest possible unit in Java, and it is also the smallest common denominator when communicating raw data between applications.

When reading from a byte stream, the program assumes full responsibility to interpret the bytes correctly. Neither the stream, nor the bytes themselves offer any guidance as to what they mean. It *could be* ASCII characters from an old textfile, or it *could be* 128 bit double values from a supercomputer: the programmer must know how to interpret the bytes.

Similarly, when writing a byte stream, the program must write the bytes in a sequence that can be understood by the intended recipient.

The Standard Library offers several helper classes that operate on top of a byte stream to write more complex units.

Figure 13.1 shows the class hierarchy below class `java.io.OutputStream`. Class `OutputStream` is the *abstract superclass* of the Standard Library classes that are, or use, byte streams. As can be seen from figure 13.1, there are four subclasses to `OutputStream`:

```
java.io.OutputStream
  |
  +--java.io.ByteArrayOutputStream
  |
  +--java.io.FileOutputStream
  |
  +--java.io.ObjectOutputStream
  |
  +--java.io.FilterOutputStream
  |     |
  |     +--java.io.BufferedOutputStream
  |     |
  |     +--java.io.DataOutputStream
  |     |
  |     +--java.io.PrintStream
```

Figure 13.1: Output streams

- `ByteArrayOutputStream` — This class writes bytes to an in-memory array of byte. The array expands automatically as bytes are added to it. No files or network connections are involved.
- `FileOutputStream` — This class writes bytes to a file, with no buffering.
- `ObjectOutputStream` — This class can write both primitive data types and objects to an `OutputStream`. Only objects that implement the marker interface `Serializable` can be written to a stream.
- `FilterOutputStream` — This is the superclass for a group of classes that *filter* or *transform* data in some way, before writing to an `OutputStream`.
  - `BufferedOutputStream` — This class provides buffering for an `OutputStream`.
  - `DataOutputStream` — This class writes the primitive data types to an `OutputStream`.
  - `PrintStream` — This class provides two services to an `OutputStream`, the `print` and `println` methods, and buffering. Characters written by the `PrintStream` are converted to bytes using the platform's default encoding but should still be considered to be binary data. Use a `PrintWriter` when writing a text file.

The classes below `OutputStream` make up a toolbox for writing binary files. Text is written with other classes. The `OutputStream` classes are used with the combination of three choices:

- Should the byte stream go to memory or to a file? This dictates if one creates a `ByteArrayOutputStream` or a `FileOutputStream`.

- Should the stream be buffered or not? If the stream goes to a file, buffering is highly recommended and a `BufferedOutputStream` should be used.
- Are we writing objects, primitive data types, or text in binary form? These choices select between `ObjectOutputStream`, `DataOutputStream`, or `PrintStream`.

Example: savepoint

Let us assume that we have number-crunching program that runs for days. At suitable times we want to save our current results to a file, so that we have a savepoint to restart from. We want write a binary file with our numbers efficiently saved. Our data is in the form of two large arrays of double. We want to write to a file, with buffering, and we want to write primitive data types (this example does not include exception management):

```
import java.io.*;

double [] aData   // The arrays to be saved
double [] bData

// (1)
FileOutputStream fos = new FileOutputStream("savepoint.dat");
BufferedOutputStream bos = new BufferedOutputStream(fos);
DataOutputStream dos = new DataOutputStream(bos);

// Write the first array

// Length before data helps when reading back
dos.writeInt(aData.length);

for (double d : aData)
  dos.writeDouble(d);

// Write the second array
dos.writeInt(bData.length);

for (double d : bData)
  dos.writeDouble(d);

// Remember to flush out buffered data
dos.flush();
// And we are done
dos.close();
```

Look at section (1) of the program code. First a `FileOutputStream` is created. Second, that stream is given to a `BufferedOutputStream`, so that we do not harass the file system. Third, the `BufferedOutputStream` is given to a `DataOutputStream` so that we can use its methods to write our primitive data type values.

```
      +------------------------------+
dos-->|  DataOutputStream            |
      |                              |
      |   +------------------------+ |
      |   | BufferedOutputStream   | |
      |   |                        | |
      |   |  +-------------------+ | |
      |   |  | FileOutputStream  | | |
      |   |  |         +--------------------->savepoint.dat
      |   |  +-------------------+ | |
      |   |                        | |
      |   +------------------------+ |
      |                              |
      +------------------------------+
```

Figure 13.2: Nested `OutputStream` instances

Figure 13.2 illustrates how the three `OutputStream` objects cooperate, providing different services.

When we call methods in the `DataOutputStream` instance to write `long` or `double` values, the `DataOutputStream` first converts the value into bytes. Then it writes those bytes to the `OutputStream` it was given when it was created, the `BufferedOutputStream`.

The `BufferedOutputStream` then puts the bytes it receives into its internal buffer, until the buffer is filled to capacity. When that happens, the bytes accumulated so far are given as a whole chunk of bytes (an array) to the `OutputStream` it was given when it was created, the `FileOutputStream`.

The `FileOutputStream` in turn, when it receives bytes to write, sends them to the operating system which in turn writes them to the file, using whatever block size the filesystem requires.

13.1.3 Object streams

The `ObjectOutputStream` is used to write both primitive data values and objects. Primitive values are encoded as bytes, using as many as are required to store them: `int` four bytes, `double` eight bytes.

The process of transforming an object to a sequence of bytes is called *serialization*. The only meaningful purpose of serialization, is to be able to read it back at some point in the future, to *de-serialize* it, and reconstruct the object. To this end, an object stream can only be meaningfully interpreted by another Java program.

Not all classes can be serialized. The programmer must explicitly decide if a class is to be serializable or not, and if so, make the class implement the interface `java.io.Serializable`. This interface is a *marker* interface; it has no methods. Its only purpose is to indicate the property that the implementing class is allowed to be serialized. For example:

```
import java.io.Serializable;

class PlotMarker implements Serializable {
  private double longitude, latitude;
  ...
}
```

When an object instance is serialized, what actually goes onto the byte stream, is the fully qualified class name of the object, and the contents of its instance variables. Primitive data type variables are encoded directly. Reference type variables that are not `null` are followed, and if the referred object is also serializable, it goes on the byte stream as well. This makes it possible to save an entire *object graph*, at least the components that are `Serializable`.

Later on, when the object stream is read, the objects are reconstructed by reading the class names off the stream, finding the class in the codebase (classpath), creating an object instance and restoring the variable values. This has the immediate consequence that all the required class files must already be available to the reading program. The object stream only tells the story of how some object instances happened to be configured when the object stream was written. The object stream never contains the full objects.

Example: object stream

This is an example of how to create and write an object stream to a file:

```
import java.io.*;
import java.util.Date;
...
FileOutputStream fos = new FileOutputStream("obj.dat");
ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(new PlotMarker(59.4202, 18.8931));
oos.writeobject(new Date());

oos.close();
```

The same example, this time with buffering:

```
import java.io.*;
import java.util.Date;
...
FileOutputStream fos = new FileOutputStream("obj.dat");
BufferedOutputStream bos = new BufferedOutputStream(fos);
ObjectOutputStream oos = new ObjectOutputStream(bos);
```

```
java.io.InputStream
  |
  +--java.io.ByteArrayInputStream
  |
  +--java.io.FileInputStream
  |
  +--java.io.ObjectInputStream
  |
  +--java.io.PipedInputStream
  |
  +--java.io.SequenceInputStream
  |
  +--java.io.FilterInputStream
  |     |
  |     +--java.io.BufferedInputStream
  |     |
  |     +--java.io.DataInputStream
  |     |
  |     +--java.io.PushBackInputStream
  |     |
  |     ...
  |
  ...
```

Figure 13.3: The `InputStream` class hierarchy

```
oos.writeObject(new PlotMarker(59.4202, 18.8931));
oos.writeobject(new Date());

oos.flush();
oos.close();
```

13.1.4 Byte streams: reading binary data

The abstract class `InputStream` is the superclass of the classes that help with reading byte streams, i.e. binary input. The class hierarchy is shown in figure 13.3. Not all classes are shown: deprecated classes, classes related to zip-files, security, and audio are omitted, but can of course be found in the documentation for the Standard Library.

The classes are:

- `ByteArrayInputStream` — This class allows the program to read from an array of bytes as if it was a byte input stream.
- `FileInputStream` — Creates an input stream from the bytes in a file in the file system.

- ObjectInputStream — Used to read objects from a byte stream that was created by an ObjectOutputStream.
- SequenceInputStream — This is a sort of meta-stream: several different input streams are concatenated by the SequenceInputStream. The program reads bytes, and the SequenceInputStream switches to the next stream when the current one is exhausted.
- FilterInputStream — This is the superclass of the various classes that perform modification or filtering of the bytes from the input stream.
  - BufferedInputStream — This class wraps a buffer around the byte input stream. This is useful in that the reading code can *mark* a position in the stream, and then later *reset* the read position back to the mark. This is useful when the byte stream must be examined a few bytes ahead before it is clear how to interpret them.
  - DataInputStream — This class is used to read primitive data type values from the stream.
  - PushBackInputStream — This class gives the capability to push back (unread) a single byte onto the input stream. In essence, this is a limited version of the functionality offered by BufferedInputStream.

Example: reading the savepoint

This example shows how to read primitive data values using a DataInputStream wrapped around a FileInputStream. The example illustrates how to read back the savepoint created earlier:

```
import java.io.*;

double [] aData;
double [] bData;

FileInputStream fis = new FileInputStream("savepoint.dat");
DataInputStream dis = new DataInputStream(fis);

// Read the length of the first array
int len = dis.readInt();
// Create the first array
aData = new double[len];
// Read first array data
for (int i = 0; i < len; i++)
  aData[i] = dis.readDouble();

// Read the length of the second array
len = dis.readInt();
// Create the second array
bData = new double[len];
// Read second array data
for (int i = 0; i < len; i++)
  bData[i] = dis.readDouble();
```

```
java.io.Writer
  |
  +--java.io.BufferedWriter
  |
  +--java.io.CharArrayWriter
  |
  +--java.io.FilterWriter
  |
  +--java.io.OutputStreamWriter
  |    |
  |    +--java.io.FileWriter
  |
  +--java.io.PipedWriter
  |
  +--java.io.PrintWriter
  |
  +--java.io.StringWriter
```

Figure 13.4: The Writer class hierarchy

```
dis.close();
```

Reading an object stream

This example reads the object stream created earlier:

```
import java.io.*;
import java.util.Date;
...
FileInputStream fis = new FileInputStream("obj.dat");
ObjectInputStream ois = new ObjectInputStream(fis);

PlotMarker pm = (PlotMarker) ois.readObject();
Date dt = (Date) ois.readObject();

ois.close();
```

13.1.5 Writing text — Character streams

To write text on character streams, you use a (subclass of) `Writer`. Figure 13.4 shows the abstract `Writer` class and its subclasses. These are used to write character streams, i.e. *text*. This is different from the byte streams that deal with binary data in the form of bytes. In a character stream the smallest unit is a Unicode character.

With overloading in mind, the `Writer` class basically has these methods:

- `append` — append a single character or character sequence to the writer (write to the stream)
- `flush` and `close`
- `write` — write one or more characters or a string to the character stream

The subclasses of `Writer` then implement the abstract methods and override the non-abstract methods to provide their particular behaviour.

The `Writer` subclasses offer the following services:

- `BufferedWriter` — Wraps around another `Writer`, applying buffering.
- `CharArrayWriter` — Adds characters to an in-memory array that grows as required.
- `FilterWriter` — Abstract superclass for classes that would like to apply some modification or translation of the characters sent to the stream. The Standard Library does not contain any such classes, so this is for applications that want to subclass `FilterWriter` themselves.
- `OutputStreamWriter` — The `OutputStreamWriter` is a bridge from a character stream to a byte stream. It maps from Unicode characters to bytes, using a specific character set (`java.nio.charset.Charset`). The character set used can be the platform default, or it can be specified explicitly. This mapping is done because all Java programs have the same internal character representation (Unicode). However, outside the program, text can be encoded in many different ways: UTF-8, ISO-8859-1, and so on.
- `FileWriter` — This is a convenience class for writing text to a file using the default character set and buffer sizes.
- `PipedWriter` — A `Writer` for piped character-output streams. Included here for completeness, but will not be discussed further.
- `PrintWriter` — This class offers the `print`, `println` and `printf` methods for easy and formatted printing of text. It can be constructed directly with a given file and character set, and thus offers a very convenient tool for writing text files.
- `StringWriter` — This class collects its output in a buffer that can later be retrieved as a `StringBuffer`.

Example: write a text file

The easiest way of writing a text file in the default encoding is to use a `PrintWriter`.

```
import java.io.*;

PrintWriter pw = new PrintWriter("report.txt");
pw.println("Hourly report");
...
pw.flush();
pw.close();
```

Example: write a text file with UTF-8 encoding

To specify a character set, use the alternate constructor of the `PrintWriter`:

```
import java.io.*;

PrintWriter pw = new PrintWriter("nodes.txt","utf-8");
pw.println("Nodes found:");
...
pw.flush();
pw.close();
```

13.1.6 Reading text from character streams

The abstract class `java.io.Reader` is the superclass of classes that help with reading character streams. They are shown in figure 13.5 and have the following behaviours:

- `BufferedReader` — This class wraps a buffer around the input character stream and supports the *mark* and *reset* operations. This is useful when the program needs to look ahead at some characters, before it can decide how to read them.
- `CharArrayReader` — This class allows a character array to be read as if it was a character stream.
- `InputStreamReader` — This class maps from bytes to characters, using some character set. This means that for each character read by the program, the `InputStreamReader` may have to read one or more bytes from the underlying byte stream. The Standard Library documentation recommends wrapping a `BufferedReader` around the `InputStreamReader`. That way several character conversions can take place at once, as the buffer is filled, rather than just one conversion at a time.

```
java.io.Reader
  |
  +--java.io.BufferedReader
  |
  +--java.io.CharArrayReader
  |
  +--java.io.InputStreamReader
  |     |
  |     +--java.io.FileReader
  |
  +--java.io.PipedReader
  |
  +--java.io.StringReader
  |
  +--java.io.FilterReader
        |
        +--java.io.PushbackReader
```

Figure 13.5: The `Reader` class hierarchy

  - `FileReader` — This is a convenience class for reading characters from a file with the default character set and buffering.

- `PipedReader` — Reads character that were placed in a `PipedWriter`.
- `StringReader` — This class makes it possible to read from a `String` as if it was a character stream.
- `FilterReader` — The superclass of character streams that perform some kind of filtering or modification of the characters.
  - `PushBackReader` — Allows a single character to be pushed back onto the character stream (compare with `BufferedReader`).

It is important to understand that the `Reader` classes, with the help of the `InputStreamReader`, converts from character encodings *outside* the Java program, to the Unicode character set used *inside* the program. For this reason it is important to know what character encoding that is used in the files that the program needs to read.

First off, if left unspecified, the JVM will pick the default encoding for the system it is running on. This may be different on different systems, like between Apple iOS, Microsoft Windows, Linux, VMS, or other systems. It may even be the case that several encodings are used on the same system, because of legacy systems or compatibility requirements.

Because of this, and the general portability of Java, there are two special cases where one can generally disregard the character encodings used by the environment:

- The writing and reading Java programs are on the same computer. In this case, they can probably safely use the default character encoding for files, because it will be the same for both programs.
- The writing and reading Java programs are on *different* computer systems with different operating systems. In this case, the exchange character encoding for the applications should be in the requirements, and both writers and readers should explicitly specify a character encoding to use between them.

Remember, if a program can write a text in one character encoding, it can also write it in another. If compatible text output is required in several directions, it may be necessary to write the file several times, with different character encodings.

Back to reading: once the bytes have been decoded into Unicode characters inside the Java program, the next level of interpretation is entered, and that is to understand what the text means. Counting character positions, finding delimiter characters, extracting fields, parsing digits into numbers, and so on. The `Reader` classes support this activity minimally, but luckily there are other classes in the Standard Library that can do a lot more for general text, like `java.util.Scanner`.

Example: reading default encoded text

The example below reads a text file using a `FileReader` that provides the default character decoding and byte-buffer size. The characters read are returned as positive `int` values. When the end of the file has been reached, `-1` is returned. The example puts the characters in a `StringBuilder` just to do something meaningful with them.

```
import java.io.*;
...
FileReader fr = new FileReader("Report.txt");
StringBuilder sb = new StringBuilder();

for (int c = fr.read(); c != -1; c = fr.read())
  sb.append((char) c);

fr.close();
```

Example: reading specifically encoded text

This is the same as the previous example, but with a specified encoding. The `FileReader` cannot help with this, so extra steps have to be taken:

```
import java.io.*;
...
```

```
FileInputStream fis = new FileInputStream("Report.txt");
InputStreamReader isr = new InputStreamReader(fis, "utf-8");
BufferedReader bfr = new BufferedReader(isr);

StringBuilder sb = new StringBuilder();

for (int c = bfr.read(); c != -1; c = bfr.read())
  sb.append((char) c);

bfr.close();
```

Example: reading lines of text

The `BufferedReader` has the ability to return full lines of text from the file. This is often both desired and convenient. The following example reads a text file line by line and prints them on the console, adding line numbers:

```
import java.io.*;
...
FileReader fr = new FileReader("text.txt");
BufferedReader bfr = new BufferedReader(fr);

int lineNumber = 1;

for (String s = bfr.readLine(); s != null; s = bfr.readLine())
  System.out.println(lineNumber++ + ": " + s);

bfr.close();
```

13.2 Standard input and standard output

The variable `java.lang.System.in` is a reference to an `InputStream`. This is the standard input stream, which by default is connected to the computer keyboard. When keys are pressed there, bytes can be read from `System.in`. To allow for the correction of typing mistakes, keyboard input is line-buffered, and bytes only become available when the user has pressed the Enter key.

To transform the bytes into characters, an `InputStreamReader` must be used. It is also possible to use some other object, like a `java.util.Scanner`, that performs the mapping from bytes to characters internally.

The variable `java.lang.System.out` is a reference to a `PrintStream` object, connected to standard output, which by default goes to the command or console window in which the program is running. The `PrintStream` offers the `print` and `println` methods for converting primitive data values and strings into characters, and then converting those characters to bytes, in the default character encoding of the platform.

The default character encoding of the `PrintStream` sometimes leads to unexpected characters in the output. The most common example is when running a Java program in a command window (`cmd.exe`) on Microsoft Windows. The default character encoding on a Windows system is WINDOWS-1252. However, due to historical reasons, the most frequent default character set in the command window is WINDOW-858, which is different from WINDOWS-1252. This has the effect that when a Java program is making output to a command window, the system default specifies that characters are encoded into bytes for WINDOWS-1252, but then they are displayed as if they were WINDOWS-858. The result is of course gibberish. This mistranslation does not affect the ASCII characters, but is obvious for Swedish national characters.

13.2.1 Reading characters from `System.in`

To read characters from `System.in`, an `InputStreamReader` is used. You wrap it around the `InputStream` referred to by `System.in`:

```
import java.io.*;
...
InputStreamReader isr = new InputStreamReader(System.in);

int c = isr.read();
```

13.2.2 Reading lines of text from `System.in`

Reading input character by character is tedious, and it is often more useful to read a whole line at a time when reading from standard input. That is, after all, how the user is typing. To do this, the `InputStreamReader` is wrapped inside a `BufferedReader`:

```
import java.io.*;
...
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader bfr = new BufferedReader(isr);

String s = bfr.readLine();
```

13.2.3 Reading tokens and text from `System.in`

The class `java.util.Scanner` is useful when reading from standard input, as it can convert bytes to characters, buffer them into lines, and even parse the input into numbers when required. Using a `Scanner` is easy because a single object performs so much:

```
import java.util.Scanner;
...
```

```
Scanner sc = new Scanner(System.in);

String s = sc.nextLine();
int k = sc.nextInt();

sc.close();
```

Please see 3.1.4, 9.2.3, and 12.5 for more about the `Scanner` class.

13.2.4 Summary: the output stream hierarchy

Java programs use the concept of *streams* to write data. Streams can be either *binary* or *character* streams.

With binary streams the program has full control, and assumes all responsibility for how the bytes are written and if they will be meaningful when read again later.

The binary streams in Java are represented by the `OutputStream` class and its subclasses.

Character streams, on the other hand, are meant to produce text that is readable and where the identity of each character is preserved (modulo output encoding limits). The character stream contains text that can be stored in text files and perhaps processed further in documents, emails, or databases. Character streams do *not* produce *formatted* text, but they can be used to produce text embedded in generalized markup, like HTML or XML, which in turn may be rendered as formatted when read by a suitable interpreter.

The character streams in Java are represented by the `Writer` class and its subclasses. It is important to realise, that at the lowest level all streams are binary. Therefore, when the characters in a character stream leave the JVM, they must be converted to bytes according to some encoding standard.

13.2.5 Summary: the input stream hierarchy

The binary input streams reads bytes from the environment. The various subclasses of `InputStream` provides bytes from files, buffers them, and assemble them into primitive data types or objects. Reading from a binary stream requires that the program knows exactly what to expect and what conversion from bytes to data that is to be performed.

The `Reader` class hierarchy reads text from the environment, and performs the decoding from external encoding to internal Unicode. Again, at the bottom the program sees a stream of bytes, the bytes are converted into characters per the expected external encoding, the characters are converted to Unicode, and are finally made available to the program as a stream of Unicode characters.

The `java.io` package provides streams of Unicode characters. To extract additional meaning from the characters, like strings or numbers, a process called *parsing* must be invoked. The `java.util.Scanner` class gives basic support for parsing, but for more serious applications like a compiler, interpreter, or text analyzer, a lot of programming is usually required.

13.2.6 Formatted output - printf

In this section, the term *formatted* does not refer to properties of rendered text like bold, italic, underlined and so on. Instead it refers to the process of producing monospaced text in defined *fields* (columns). Typical applications are report pages in console listings or in files, to help help the human eye to correctly and quickly find the relevant information.

The C programming languages has for long offered a set of library routines called *printf* for *print formatted*. The Java language therefore has a similar library method. It actually appears in several places:

- `java.io.Console.format`
- `java.io.Console.printf`
- `java.io.PrintStream.format`
- `java.io.PrintWriter.printf`
- `java.io.PrintStream.format`
- `java.io.PrintWriter.printf`
- `java.lang.String.format`

The method names `format` and `printf` are just aliases for each other, even though it seems likely that `format` is the Java name, and `printf` is just present to make old C programmers happy. We will first focus on the `format` method, because that it the only one present in class `String`, and this is very much a string processing business.

The `format` method comes in two flavours. The simple one looks like this:

```
format (String fmt, Object ... args)
```

This means that you provide a formatting string (more about that below), followed by a variable number of object arguments. The formatting string determines how to convert the elements in the object list to text.

The second flavour of the call looks like:

```
format (Locale loc, String fmt, Object ... args)
```

In this version of the method, the caller can specify which locale to use. The difference between locales could be to use a decimal comma or a decimal period. The first version of the call uses the default, which in turn is determined by the JVM from the system it is running on.

Although this text has up to this point not discussed methods with a variable number of arguments, the short story is that the type of parameter variable `args` is actually `Object []`, i.e. an array of `Object`. The three periods `...` instructs the compiler to count the arguments in each call and insert code to generate the appropriate array. In addition, since the format call also works with primitive data types, we can deduce that *autoboxing* is at work, creating instances of wrapper classes around each element. The following legal call:

```
format("%6s %6d", "Gain:", 123)
```

will thus be massaged by the compiler into:

```
format("%6s %6d", new Object[]{"Gain:",
                               new Integer(123)})
```

which of course saves a lot of tedious typing. The output generated by the call is to print both values right-justified in a field of 6 columns, separated by a single whitespace:

```
 Gain:    123
```

The `format` call parses the formatting string from left to right, and then picks arguments from the args array for the corresponding conversion. If the second conversion in the formatting string specifices an integer value, then the second element in the array (the *third* parameter to the format call) must be an integer expression. If the conversion and the argument cannot be made to match, an `IllegalFormatConversion` is thrown.

The full documentation of the formatting string can be found in class `java.util.Formatter`. Some common ways to use it are shown below.

The formatting string

The formatting string contains ordinary text that is printed, together with *conversion sequences* that define where and how one element of data from the arguments is to be converted to text and inserted in the output. It is of course possible to have a formatting string that only contains conversion sequences, just as it is equally possible to have no conversion sequence at all, and just text to output.

Conversion		Description
`b`	`B`	general boolean
`s`	`S`	String
`c`	`C`	char
`d`		integer, decimal number
`o`		integer, octal number
`x`	`X`	integer, hexadecimal number
`e`	`E`	floating point, scientific notation
`f`		floating point, decimal number
`g`	`G`	floating point, e or f for best fit
`%`		the `%` character
`n`		a newline

Table 13.1: Common conversions

A conversion sequence begins with the percent sign `%`. Then follows a sequence of optional fields which end in a single mandatory character that specifies what kind of conversion is requested. The general syntax is:

```
%[argument_index$][flags][width][.precision]conversion
```

Very briefly, the fields are:

- argument index — Which element in the *list of arguments* to select. The first element is numbered 1, and selected with `1$`. If this field is omitted, the first conversion sequence selects the first argument, the second conversion selects the second, and so on.
- flags — Characters that modify the output, depending on what kind of conversion is selected.
- width — A non-negative number specifying the minimum number of characters to be written by this conversion. This field can also be seen as column width. When converting a number, all digits in the integer part are always written, possibly overflowing the field.
- precision — For number conversions, this is the number of decimals to print. The decimal period, and the decimal digits are *included* in the field width, not appended to it.

Table 13.1 shows the more commonly used conversions. The uppercase conversion character prints uppercase letters in the output.

It should also be remembered that the argument list as seen by the formatting code is actually an array of `Object`. This means that the elements in that array are references to other objects, like instances of `String` or `Integer`, inserted by the compiler. But since the elements are references, the possibility exists that one or more of those array elements contains the value `null`, the reference to nothing.

The treatment of `null` values is that for most conversions the string `null` is printed if the argument is `null`. The `b` conversion, however, will always print `false` or `true`. It will print `false` if the argument is `null`, or a `Boolean` with a `null` value. It will print `true` if the argument is a `Boolean` with a `true` value, or any other non-null reference. This special behaviour of the `b` conversion makes it difficult to see what the actual argument was.

Here are some examples:

```
double d = Math.PI;
System.out.printf("The value of PI: %f%n", d);
```

This generates the output:

```
The value of PI: 3.141593
```

The conversion string contains a leading text that ends in a colon and a space. These are all printed to the output. The comes the `%f` conversion, which takes the first argument from the argument list (the variable `d` in this case), converts it to a string and prints that. Finally, there is the conversion `%n` which consumes no argument but just outputs a newline.

The advantage of formatted printing comes when you want to line up things neatly in columns. Let us assume that we have a table of two columns and three lines to print:

```
double [][] table =
  {{42, -9.2342},
   {-0.0732, 15.5},
   {0, -100}};

// Labels for columns
System.out.printf("%-8s%10s%10s%n",  // (1)
                  "Item", "Today", "Yesterday");

// The table rows
for (int i = 0; i < table.length; i++) {
  System.out.printf("%8d%10.2f%10.2f%n", // (2)
                    i+1, table[i][0], table[i][1]);
}
```

The output becomes:

```
Item         Today Yesterday
       1     42.00     -9.23
       2     -0.07     15.50
       3      0.00   -100.00
```

The column labels are printed with formatting string (1). It contains no text, just four conversions. The first one, `%-8s` prints a string of text, in a field of 8 characters. The flag `'-'` requests that the text is left-justified within the field. The second and third conversions are both `%10s`, they specify a string in a field of 10 characters. The last conversion `%n` outputs a newline.

The data lines are printed with formatting string (2). This also has four conversions. The first is `%8d` asking for an integer printed in a field of 8 characters. The second and third conversion are identical, `%10.2f` and specifies a floating point number, printed in a field of 10 characters, with two decimal places. The formatting string ends with a newline conversion.

13.3 Managing files

Apart from the immediate interactions with the user through the keyboard, console, and graphical user interfaces, a computer program must also be able to manage files in the filesystem. Files contain temporary and permanent data that can be read or written, but the program can also manipulate files as whole units. Files can be deleted, renamed, and moved around.

Since a filesystem very often is organized in *directories* (often shown as paper folders in a user GUI), the program must also be able to inspect and manipulate directories.

13.3.1 The `java.io.File` class

A file is represented within the Java program by an instance of the class `java.io.File`. Just because a Java program has created an object instance of `File`, it does not follow that a corresponding file exists in the filesystem. There are three distinct possibilities:

- The file does not yet exist but it will be created
- The file does exist, and will be used by the program
- The file did exist, but is now gone, because it was deleted, or the filesystem on which it resides is no longer present (for example, a USB memory stick was removed)

The `File` objects therefore allows the program to reason about files that may or may not exist.

In reality though, the `File` class provides three important services:

- It provides a mapping between abstract pathnames (used inside the Java program), and the pathname syntax used on the current host

- It allows the program to query and manipulate properties of the file
- It gives the program access to the pathname syntax of the current host

The concept of a *pathname* is a string of symbols that indicates the position of a file in a filesystem. An absolute pathname begins at the top of the name space specifying a host, a device on that host, followed by a sequence of directories that ends either with a directory or a file. For example, an absolute Unix/Linux pathname:

```
/home/fk/id1018/a.txt
```

An absolute Microsoft Windows pathname:

```
C:\USERS\FK\ID1018\A.TXT
```

An absolute Microsoft Windows UNC pathname:

```
\\GOY\PUBLIC\FK\ID1018\A.TXT
```

Relative pathnames are those that are specified relative to the *current directory*, which usually is the directory from where the program was started. A relative pathname specifies how to find a file or directory in relation to the current directory. For example, a file in the current directory:

```
a.txt
```

A file in subdirectory `assets`:

```
assets/a.txt
```

A Java program is designed to be portable, and the use of abstract pathnames in the `File` class supports this. A program that uses an absolute path *must* consider how to match the specific syntax of the host file system. A program that only refers to files in a relative way can use abstract relative paths and is likely to be more portable.

13.3.2 Verify that a file exists and is readable

To avoid a `FileNotFoundException` a program may want to verify that a file actually exists and can be read before attempting to do so. Let us assume that a file is called `savepoint.dat` and is expected to be in the current directory:

```
import java.io.File;

File f = new File("savepoint.dat");

if (f.canRead()) {
  ... // open the file
}
else
  System.out.printf("File not found: %s%n", f.getName());
```

13.3.3 Directories and files

The last name in a path can refer to either a directory or a file. When a `File` object has been created for the path, it can be queried to determine which:

```
import java.io.File;

File f = new File("assets/old"); // relative path

if (f.isDirectory()) {
  // a directory: get paths to content
  File [] content = f.listFiles();
  ... // process the array
}
else if (f.isFile()) {
  // it is a file
  if (f.isHidden()) {
    ... // it is a hidden file
  }
}
```

For another example, assume our program wants a temporary directory while working. It is called `tmp` and is relative to the current directory. The logic is that on a succesful run we create `tmp`, execute the work, and then delete `tmp`. However, if `tmp` already exists we must do nothing, because another instance of the program may already be working in it.

```
import java.io.File;

File tmpdir = new File("tmp");

if (tmpdir.exists()) {  // name exists
  if (tmpdir.isDirectory())
    throw new Exception("tmp dir already exists");
  else
    throw new Exception("tmp already exists - not a dir");
}

// create the temporary directory
if (!tmpdir.mkdir())
  throw new Exception("could not create tmp dir");
```

```
doWork(tmpdir); // do work, using this tmp directory

// remove tmp files
File [] tmpFiles = tmpdir.listFiles();
for (File f : tmpFiles)
  if (!f.delete())
    System.err.printf("Failed to delete file %s%n",
                      f.getPath());

// tmp directory is hopefully empty, delete it
if (!tmpdir.delete())
  System.err.printf("Could not delete %s%n",
                    tmpdir.getPath());
```

It is not allowed to remove a non-empty directory. Therefore we must first remove all files in `tmp` before we attempt the removal of the directory itself.

Obviously there is a lot more to be said about files in Java, and how to manage them. The package `java.nio` (new I/O) contains additional tools for modernized and advanced file operations, such as watching a directory for changes, or the efficient reading of binary data in units larger than a byte (e.g. 16 bit audio samples).

13.4 Random access files

The streams concept, while general, has one serious limitation, and that is that it is serial. In order to update a record somewhere in a file, you have to read and write the whole file. While block oriented, disk devices have *random access*; any randomly selected block on the disk can be accessed in the same time (on average).

For applications that need to read or update records all over a file (or a set of files), serial access is not a practical solution. Such applications need random access.

The class `java.io.RandomAccessFile` offers a solution for this. The class presents a model of the file as a sequence of bytes. The current position is called the *file pointer*, and it can be set anywhere between 0 and the maximum file length supported by the underlying filesystem.

There are methods that support reading or writing of the primitive datatypes. Both will advance the current position towards the end of the file, by the size of the datatype. If the file pointer moves or is set beyond the end of the file on writing, the file grows automatically. Holes are filled with zero bytes.

Using the `RandomAccessFile` is easiest with records of fixed length. This is because the address (the byte position) of a certain record can be easily computed by multiplying the record number with the record size (in bytes). However, if the data contained in each record varies much in size, it may be difficult to decide on the best record size. Too small, and data will often not fit completely. Too large, and there will be many records with unused space.

More advanced solutions use records of variable length, but then some kind of management structure is required to ensure that records do not overlap, and that they can be located reasonably fast. This quickly gets complicated, and the computer science literature has a wealth of information on data structures and algorithms for this particular problem.

For an example, let us assume that we have a fixed record size that contains an identity number (`int`), a name field (a string of 64 characters), and a data field (`double`). The byte-size of the record will be 4 bytes for the `int`, 8 bytes for the `double`, and 128 bytes for name string (each character is stored in two bytes). To make things a little easier for us, we create a class that we can use to manage the record internally in our program:

```
import java.io.*;

class AppRecord {
  public static final long NAME_SIZE = 64;
  public static final long NAME_BYTES = NAME_SIZE * 2;
  public static final long RECSIZE = 4 + NAME_BYTES + 8;

  int id = -1L;
  String name = "";
  double data = 0d;

  public AppRecord (); // default constructor

  // parameterized constructor
  public AppRecord (int id, String name, double data) {
    this.id = id;
    this.name = name;
    this.data = data;
  }

  // Construct by reading from random access file
  public AppRecord (RandomAccessFile raf, int recNum)
    throws IOEXception
  {
    read (raf, recNum);
  }

  public read (RandomAccessFile raf, int recNum)
    throws IOEXception
  {
    raf.seek(recNum * RECSIZE);

    id = raf.readInt(); // Read id

    // Read the string into an array of char
```

```
      // then construct a string from the array
      char [] ca = new char[NAME_SIZE];
      for (int i = 0; i < ca.length; i++)
        ca[i] = raf.readChar();
      name = new String(ca);

      data = raf.readDouble(); // Read data
    }

    // Write this the record to the random access file
    // at the given record position.
    public void write (RandomAccessFile raf, int recNum)
      throws IOException
    {
      raf.seek(recNum * RECSIZE);

      raf.writeInt(id); // Write id

      // If the name is too long it must be truncated,
      // if too short it is padded with blanks to ensure
      // that old characters are erased from the record.
      if (NAME_SIZE < name.length())
        raf.writeChars(name.substring(NAME_SIZE));
      else {
        StringBuilder sb = new StringBuilder(name);
        while (sb.length() < NAME_SIZE)
          sb.append(' ');
        raf.writeChars(sb.toString());
      }

      raf.writeDouble(data);
    }
  }
```

An application program can then be written to take advantage of the `AppRecord` class. A random access file can be opened in four different modes, the mode is specified by a string of characters:

- `"r"` — read only
- `"rw"` — read and write
- `"rws"` — read and write, with synchronous commit to storage
- `"rwd"` — read and write, with delayed commit to storage

The synchronous or delayed commit to storage means that any call to a method that writes to the random access file will not return until data has actually been written to the storage medium. The synchronous mode waits for both file data and meta-data to be updated, and the delayed mode only waits for file data (this is slightly faster).

Here is the example application, with several patterns shown:

```
import java.io.*;

RandomAccessFile raf = new RandomAccessFile("db.dat", "rw");

// Create and write some data

AppRecord ar = new AppRecord(42, "Domestic", 12.02);
ar.write(raf, 0); // Record 0

// Record 1, done differently
new AppRecord(199, "International", 27.33).write(raf, 1);

// Record 2, reusing ar
ar.id = 12;
ar.name = "Logistics";
ar.data = 17.6;
ar.write(raf, 2);

...

// Update record 1
ar = new AppRecord(raf, 1);
ar.data *= 1.2;
ar.write(raf, 1);

// Update record 2, reusing ar
ar.read(raf, 2);
ar.name = "Logistics and transport";
ar.write(raf, 2);

...

// done
raf.close();
```

13.4.1 Record keys

In this example the position of each record in the file is chosen arbitrarily, but this is rarely the case in real applications. Even with fixed-size records, some kind of logic is required to determine where to put each record. We need a record *key* so that we can go fast from data to a position in the file.

The id number could perhaps be used, but id number sequences are often thinly spread over a huge number range, and can rarely be used directly. Id numbers can also contain letters, and other characters like hyphens and dashes, that are not immediately translated into a numeric ordinal.

13.4.2 Personal numbers as keys

Just think for example of the Swedish personal registration number *(person-nummer)*, which while unique for each Swedish citizen, is a non-trivial record key. For example:

```
720322-1237
```

The personal number consists of a six digit date on the form YYMMDD, a dash (`'-'`) or a plus (`'+'`)[1] in the seventh character position, followed by a three digit serial number. The final *check* digit in the last position is computed from the previous digits, so it is not strictly necessary to preserve the uniqueness of the number.

It would be possible to concatenate the first nine digits and arrive at a unique integer, but this is a number in the region of 10^9. If we knew the *lowest* personal number, we could perhaps subtract that and achieve a smaller number. However, we would still be left with approximately 70 unused 'days' for each month, and 88 unused 'months' each year, so a direct mapping to a record number in a file would generate a lot of unused space.

To solve the sparseness problem, we could attempt to remap the date into a daynumber relative to some start date and calendar, and then multiply by 1000 before adding in the serial number. Apart from the added computation required, that strategy would still leave a good number of holes, because on most days only a portion of the available serial numbers are occupied.

In addition to all of the above, the three digit serial number is odd for men and even for women, which means that for any given day, only 500 men and 500 women can be given personal numbers accurate to their actual date of birth. In the situation where all serial numbers on a day have been taken, the date in the personal number when issued, is set to an adjacent date where there still are numbers left (this is also done for people born on the leap-day, 29 February). As a result, the personal number is always unique, but does not always represent the actual date of birth.

13.4.3 Hash functions

One solution (there are several) in cases where the unique key cannot be used to directly point to a file record, is to use a *hash function* to map the key string down to a chosen number of record positions in the file. A hash function takes a string of characters and performs a sequence of arithmetic operations similar to those of a pseudo-random number generator, with the difference that the same input string always generates the same output value, the *hash-value*.

[1]A plus sign is used when the person was born more than 100 years ago.

For example, assume that keys are personal number strings, and we have a hash function that maps to one million records. For any input key, the hash function will always return a value in the range $0 \ldots 999\,999$.

The hash function is usually fast and efficient, and if good it will distribute the hash-values fairly over the output range. The disadvantage of using a hash function is that it must be expected that two or more different keys may generate the same hash value. The terminology is that the keys *hash to the same bucket.* To deal with this, we must perform some additional programming.

One way is to increase the *bucket size.* We multiply the hash value with some number n, so that for each hash value we can actually store n records. But, this does not prevent the situation where we need to insert record number $n + 1$ in a bucket. So, we must allow for an *overflow chain* at the end of the file, beyond the region of the last bucket, and we are back in dynamic record management with lists and links. Again, there is a huge body of literature on the subject of hash functions and record management.

13.4.4 Binary search

If the contents of the random access file is mostly read, and rarely updated, it may be possible to keep the (fixed-size) records sorted on the value of the key. Binary search can then be used to efficiently locate the wanted record.

13.4.5 Index files

If the records are updated frequently, or are large, or vary much in size, it may be possible to create a separate file that consists of small, fixed-size records. Each record contains a copy of a record key, and the file pointer to the corresponding record in the primary file. This *index file* can then be updated, sorted, and searched more efficiently than the main file.

13.4.6 Random access files — final comments

Random access files efficiently solve the problem with serial access, but comes with their own complications. Any non-trivial application will soon find itself on the road towards to creating an embryonic database management system. Using an external database system instead, may be a better alternative.

Random access files are best suited to applications that operate on large numbers of small, fixed-sized records, which can not be held in memory, and where practical lookup can be maintained. Careful design is required, and there are many chances to reinvent a lot of already existing wheels.

Chapter 14

Creating new object types (1 of 2)

Data needed by an application is represented by variables. Sometimes a group of variables are needed, as a unit, just like the `AppRecord` example in section 13.4. In Java, the solution is to create a class. When we create a class, we also create a new datatype.

14.1 Composite data types

Composite datatypes are typically aggregates of other variables that are treated as a unit. The colocated variables describe something that has more aspects or dimensions than what fits into a single primitive variable. For example, the coordinates of a planar point:

```
class Point {
  double x;
  double y;
}
```

Another example, a complex number:

```
class Complex {
  double r;
  double i;
}
```

A third example, an address record:

```
class Address {
  String name;
```

```
    String street;
    String postCode;
    String city;
    String country;
  }
```

14.1.1 Arrays

All of the above examples could actually be represented by arrays. The planar point and the complex number could just as well be stored in:

```
  double [] da = new double[2];
```

The address record could be stored in:

```
  String [] sa = new String[5];
```

However, while the array representation works equally well for the computer (because all the fields have the same datatype), the programmer must now remember what data goes into which index. Defining constants may help with this:

```
  final int NAME_X     = 0;
  final int STREET_X   = 1;
  final int POSTCODE_X = 2;
  final int CITY_X     = 3;
  final int COUNTRY_X  = 4;

  ...
  sa[NAME_X]      = "Boffington Publishing";
  sa[STREET_X]    = "South Meandering Street 42";
  sa[POSTCODE_X] = "QZB987";
  sa[CITY_X]      = "Fancyville";
  sa[COUNTRY_X]   = "Ogria";
```

This style of programming is doable for very short and simple programs, but is in practice painful both to do and debug.

In addition, as soon as we need to store different datatypes together, like integers, floating point numbers, strings, dates and more, the array solution is no longer efficient (because then everything has to be represented by strings). We need the ability to freely combine different datatypes into a composite aggregate.

The concept of a data record goes a long way back in computer programming, and many programming languages that are not object-oriented support the record abstraction. With object-orientation, we can also introduce *code* into the record, and make the record active and clever. In Java, this is the concept of a class: data and code stored together. The data fields (variables) in an object instance represent the *state* of the object, and the code in the object defines and controls exactly how that state is allowed to be changed.

14.1.2 The definition class

When we define a class we create two things: a blueprint for object instances, with initialization of the instance variables, and resources in the form of static fields and methods. It is sometimes not obvious if a method should be an instance method or a static method. However, there are some easy cases:

- If the method operates on the instance (non-static) variables of an object, then it should be an instance method.
- If the method does not require an instance to do its work, and it is useful to the code that is using the class, then there is a good chance it could be a static method.
- Constants that define properties of *all* object instances from the class, like constant values, sizes of arrays, or directions on a map, should be static to avoid redundant duplication.
- The creation of a complex object is sometimes too much for a constructor, especially if the object needs extensive configuration from parameters, or the attachment of additional objects resources. In such cases, a static *factory* method can be considered to help with the creation of new objects.

The definition class is not limited to use only primitive datatypes in its variables. Any datatype is of course allowed. For example, imagine that we are writing an application in which we model some simple planar geometry. We create the definition of a two-dimensional point:

```
class Point {
  int x;
  int y;
}
```

Initially the point has no code, just two variables that go together (but we will develop it below).

We are also interested in triangles, so we decide to model a triangle based on our definition of a Point:

```
class Triangle {
  Point p1;
  Point p2;
  Point p3;
}
```

Now we have a triangle, represented by three points. The development of these two classes will be used as examples in the coming sections.

14.1.3 The Standard Library

The Standard Library is full of definition classes. Many of these (but not all) can be instantiated and used, some can be subclassed to create specialized and adapted versions, and most of them contains static resources in the form of constants and methods that can be called when required.

14.2 Objects with data

Objects can contain data in its variables. From the moment an object is created, to the end of its life, we want to have full control over the contents. It should never be undefined, and it should never be allowed to reach an internally inconsistent state. All operations on the data is to be controlled and orchestrated by us. We do this by careful and skilled design and implementation of the definition class.

14.2.1 Variables

The fields of an object (the instance variables) are associated with computer memory. That memory is never absent. The bits in an `int` or a `double` always has *some* value. When our expectations or assumptions about the value of a field do not match reality, we have a bug in the program. We can prevent such bugs from appearing by understanding how variables are initialized, and how they are updated. We maintain authority over the instance variables by not allowing any outside modification than what we specify.

14.2.2 Initialization

Instance variables in an object are created by the `new` operator. By default, the `new` operator assigns zero, `false`, or `null` to the variables. For example:

```
class Point {
  int x;
  int y;
}
```

When we do

```
Point p = new Point();
```

we receive a point with the coordinates (0, 0), because both variables were initialized to the value 0. We could declare a different default, if that is useful, by initializing the variables to their respective values. We could also, explicitly initialize the instance varibles to zero *just to show that this is what we expect.*

```
class Point {
  int x = 0;
  int y = 0;
}
```

Now, anyone who reads our code sees that (0, 0) is the default coordinate, and this is indeed what we have specified.

However, at this time we must consider if code outside class `Point` is allowed to modify the coordinates. Will they be allowed to do this?

```
Point p = new Point();
p.x = 4711;
p.y = -66;
```

For the sake of demonstration, we will introduce the requirement that the position of a `Point` can only be moved by providing both coordinates together. This helps the user of class `Point` to remember both coordinates every time an update takes place. To do this, we must prevent assignments from code outside the object. We make the variables `private`, and begin building the wall of *encapsulation* that will protect the instance variables:

```
class Point {
  private int x = 0;
  private int y = 0;
}
```

14.2.3 Constructors

Since the variables in class `Point` no longer can be assigned directly, we must provide methods for it. The initial step is to consider what happens at construction time. We provide a constructor in which the initial position can be given. Having done that, we no longer have the implicit default constructor available, so we make it explicit. Finally, we also put in a *copying* constructor, a constructor that uses (a reference to) another `Point` as a template:

```
class Point {
  private int x = 0;
  private int y = 0;

  public Point () {} // default constructor

  public Point (int x, int y) { // parameterized constructor
    this.x = x;
    this.y = y;
  }

  public Point (Point p) { // copying constructor
    this.x = p.x;
    this.y = p.y;
    }
  }
```

14.3 Operations

This is all very well, we can make points, but we still need to add more. We want to be able to update the point, and we want to be able to retrieve the coordinate values. We need *operations* on the point. So we add relevant methods for this:

```
  class Point {
    ...

    public int getX() { // return the x coordinate
      return x;
    }

    public int getY() { // return the y coordinate
      return y;
    }

    public void set(int x, int y) {
      this.x = x;
      this.y = y;
    }
  }
```

We also realize that since we are dealing with 2D geometry, and two points define a straight line, it may be a good idea to provide an object resource in the form of a method that returns the distance between a point and another point. We add this method:

```
  class Point {
    ...
    public double distance (Point p) {
      double dx = x - p.x;
      double dy = y - p.y;
      return Math.sqrt(dx * dx + dy * dy);
    }
  }
```

We can also easily overload method `distance` to return the distance to the origin (0, 0). A first, naive implementation might look like this:

```
  class Point {
    ...
    public double distance () {
      return distance (new Point());
    }
  }
```

We create a new comparison object using the default constructor, because we have alread decided that the default point is located at the origin. However, creating and discarding a new object with each call is a bit wasteful, so we later change the implementation:

```
class Point {
  ...
  public double distance () {
    return Math.sqrt(x * x + y * y);
  }
}
```

We are almost done with the first version of class `Point`. All that remains is to override method `toString` which we inherit from class `Object`. This method is used to return a string of text that describes the instance:

```
class Point {
  ...
  public String toString () {
    return "(" + x + ", " + y ")";
  }
}
```

With the `toString` method in place, we can, for example, use a `Point` instance in the following way:

```
Point p = new Point(-5, 8);

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

and receive the printout:

```
(-5, 8)
```

14.4 Encapsulation and interfaces

What we have done, is to create a model of a planar point. Class `Point` is the definition class for the model, and from it we can create as many instances as we like. We have also protected the internal variables from outside manipulation, by *encapsulating* them inside the object. The variables are declared to be `private`, and the only way to read their values or assign new values, is through the public methods that we have specified.

Together, the public methods specify the *interface*[1] for a `Point` object. The public methods are:

[1]The term *interface* is used in its generic sense here, and does not mean a Java interface class.

- `Point()` — The default constructor
- `Point(int x, int y)` — The parameterized constructor
- `Point(Point p)` — The copying constructor
- `int getX()` — The x coordinate accessor
- `int getY()` — The y coordinate accessor
- `void set(int x, int y)` — The joint coordinate mutator
- `double distance(Point p)` — Distance between two points
- `double distance()` — Distance from this point to the origin
- `String toString()` — The text representation transformer

The `Point` class is so simple that we do not need any internal methods. But just for the sake of the example, here is how we could use an internal private method to perform a recurring computation:

```
class Point {
  ...
  private double distance (int x0, int y0, int x1, int y1) {
    double dx = x0 - x1;
    double dy = y0 - y1;
    return Math.sqrt(dx * dx + dy * dy);
  }

  public double distance (Point p) {
    return distance(x, y, p.x, p.y);
  }

  public double distance () {
    return distance(x, y, 0, 0);
  }
}
```

14.4.1 Access levels

Access levels apply to classes, class variables, and methods. They are:

- `private` — Visible only to the defining class
- `protected` — Visible to subclasses
- `(default)` — Visible to package and subclasses
- `public` — Visible to all classes

In practice, encapsulated components should start out as `private`, while openly visible components should be declared `public`. As the class evolves, it may be necessary to change the declaration of some components, e.g. from `private` to `protected` in order to achieve the desired design. This is usually done in order to allow for groups of different classes to work efficiently together. Even so, as a project grows in complexity the benefits of encapsulation and defined interfaces usually grows faster. Quite often, the key to achieve the desired interoperability between different parts of a program is not to give access to private members, but instead to modify and extend the interfaces.

Here then, finally, is the first complete code for class `Point`:

```
public class Point {
  private int x = 0;
  private int y = 0;

  // Creates a new point positioned at the origin
  public Point () {}

  // Creates a new point at the given coordinates
  public Point (int x, int y) {
    this.x = x;
    this.y = y;
  }

  // Creates a new point at the coordinates in the
  // given point.
  public Point (Point p) {
    x = p.x;
    y = p.y;
  }

  // Returns the x coordinate
  public int getX() {
    return x;
  }

  // Returns the y coordinate
  public int getY() {
    return y;
  }

  // Sets the coordinates of this point
  public void set(int x, int y) {
    this.x = x;
    this.y = y;
  }

  // Returns the length of a straight line between this
  // point and the given point
  public double distance(Point p) {
    double dx = x - p.x;
    double dy = y - p.y;
    return Math.sqrt(dx * dx + dy * dy);
  }

  // Returns the length of a straight line between this
  // point and the origin
  public double distance () {
    return Math.sqrt(x * x + y * y);
  }

  // Returns a string describing this point
  public String toString () {
    return "(" + x + ", " + y + ")";
  }
}
```

Chapter 15

Creating new object types (2 of 2)

15.1 Instance resources and class resources

A class offers two kinds of resources, *instance* resources and *class* resources. Instance resources refers to public fields and methods that perform some operation or service related to a single object instance. The result or effect of the resource *depends on the (state of) the instance.* If the same resource is used through a different object instance, the value returned or the result obtained is likeley to be different too. As an example, let us review the following code (class `Point` is defined in chapter 14):

```
Point p0 = new Point(-5, -7);
Point p1 = new Point(11, 13);

System.out.println(p0.getX()); // use the instance
System.out.println(p1.getX()); // resource method getX
```

In this example we have two different objects, with different values in their internal variables. The result of using the instance resource method `getX()` on each object, therefore gives different results. The printout becomes:

```
-5
11
```

A class resource, on the other hand, is a static variable or method, available in the class definition. Because it does not need any object instance to be used, the resource should always give the same result for the same parameters[1].

[1]Contrary examples *can* be constructed, but those are generally not desirable.

Class Point does not have any class resources, so we must look elsewhere for an example. Three such examples are chosen, all from the standard library and class java.lang.Math:

```
double a = v * 2d * Math.PI;
```

In this example variable v contains a fraction, and the result stored in variable a is that fraction of the number of full circles (in radians). All that is beside the point, however, because what the example shows is the constant Math.PI. It is a class resource in class java.lang.Math. Its value never changes.

The second example:

```
double y = g * Math.sin(a);
```

Here we are using the static function Math.sin (multiplied by a scaling factor g) to find out the height (y component) of the angle in a. Again, this is not relevant except for one thing. Method Math.sin is a static resource in class java.lang.Math, and it always returns the same result for the same argument.

The third example:

```
double u = y + h * Math.random();
```

Here we add a bit of random noise to the value in y. The noise is fetched from the random number generator in class Math, and scaled by the factor h. The method Math.random is also a class resource, in that it does not require an object instance, and always returns the same result. The result in this case is a random number.

Actually, we can now create a class resource for class Point. We introduce the static method getRandomPoint which returns a new point, with a randomized position somewhere within a rectangle centered around the origin. This is how it looks:

```
public class Point {

  public static Point getRandomPoint(int width, int height)
  {
    // Find random coordinates in the positive quadrant
    int x = (int) (width  * Math.random());
    int y = (int) (height * Math.random());

    // Then translate relative to the origin
    x -= width  / 2;
    y -= height / 2;

    // Return a new point at those coordinates
    return new Point(x, y);
  }
  ...
}
```

It is important to understand the following with respect to method `getRandomPoint`:

- The method is a class resource. No previous `Point` instance is required to call it.
- The variables `x` and `y` inside the method are local variables. They are not the class variables of the same names.

The method is used like this:

```
Point p = Point.getRandomPoint(100, 100);
```

15.1.1 Class variables

All class (static) variables presented so far has been declared `final`, i.e. they can only be assigned once. After that they are constant, just like `Math.PI`. It is, however, legal to declare a class variable that is not `final`, and that can be updated any number of times. Such a variable may be useful for a small and simple program that does not require objects, or it could be used to remember dynamic properties of the class as the program is running.

However, from a puristic view towards object-oriented programming, a definition class should not be dynamic. It should not have an internal state that depends on its history in the execution of the program. There are two arguments for this:

The first argument is that if the definition class is immutable, then each time an object is created from the class, that time is *no different* from any other time an object is created. All object creations are equal, and all objects are created under exactly the same conditions. The behaviour of the class is thus *predictable*.

The second argument is that if it is deemed necessary to introduce such a state anyway, then it is better to do so via a separate object, rather than using class variables. The reason for this is that with class variables, there can only be *one* such state, because there is only one instance of each class variable. This restriction locks the design into a singular state.

15.1.2 Class methods

Class (static) methods supply services and resources that are independent of any pre-existing object. They can be accessed through the definition class, and they generally do something which is relevant to the class. This is different from an instance method, in which the method's computation depends on the state (values of instance variables) of the instance in which it is called.

15.2 References to other objects

An object can have (and often do have) instance variables that refer to other object instances. A class may have fields like:

```
class Person {
  String name;
  String address;
  ...
}
```

The variables `name` and `address` are references to `String` objects. The structure looks like this:

```
+-Person------+
|             |
| Name----------------->[String ... ]
|             |
| Address-------------->[String ... ]
|             |
|             |
|             |
+-------------+
```

Now, in the interest of consistency, we would like to ensure that once we have assigned a name to a person, that name will remain assigned and unmodified. We can do this by preventing outside manipulation of the `name` variable, and only allow manipulation through a method call:

```
class Person {
  private String name;
  private String address;

  public void setName (String name) {
    this.name = name;
  }
}
```

At this point the method `setName` just sets the name, but we could easily add code to inspect the name and throw an exception if it does not meet our requirements.

15.2.1 Dependencies

Since references are just that, *references*, it is easy to assign the same reference to multiple variables. Remember, for example, class `Triangle` from section 14.1.2:

```
    a---d
   / |  /
  /  | /
b---c
```

Figure 15.1: Two triangles sharing some corners

```
    a
   /| \
  / |   d
 /  |  /
/   | /
b---c
```

Figure 15.2: Moving point *a* alters both triangle

```
class Triangle {
  Point p1;
  Point p2;
  Point p3;

  public Triangle (Point p1, Point p2, Point p3) {
    this.p1 = p1;
    this.p2 = p2;
    this.p3 = p3;
  }
}
```

Assume further, that we want to create two triangles, and that they happen to be arranged as shown in figure 15.1. The left triangle is defined by points *a-b-c* and the right by *a-d-c*. Since the two triangles share some corners, we could write the following code to implement this model:

```
Point a = new Point(4, 3);
Point b = new Point(0, 0);
Point c = new Point(4, 0);
Point d = new Point(8, 3);

Triangle leftT  = new Triangle(a,b,c);
Triangle rightT = new Triangle(a,c,d);
```

Both triangles now have references to points *a* and *c*. It should be immediately obvious that if we move, for example, point *a*, then *both triangles will change their shape.* Figure 15.2 illustrated this. The two instances of `Triangle` depend on the same references.

In general, co-dependent references between object instances are not desirable. It complicates the program, and references can become a tangled web of spaghetti. We therefore would like to separate the *definition* of the triangle from the resources that *implement* the triangle.

15.2.2 Independent references

To ensure that our object instances have independent references, we could supply each triangle different object instances when we create them. That would work, but it does not guarantee independence. A single slipup would violate the independent property. It is therefore far better if the triangle class itself can ensure that every triangle instance created is independent of any other. We do this in the constructor:

```
class Triangle {
  private Point p1;
  private Point p2;
  private Point p3;

  public Triangle (Point p1, Point p2, Point p3) {
    this.p1 = new Point(p1);
    this.p2 = new Point(p2);
    this.p3 = new Point(p3);
  }
}
```

The updated constructor is using the copying constructor of class `Point`, and instead of simply copying the references, makes its own copies of the corner points. Now it does not matter what further happens to the objects passed as the constructor parameters. The triangle instance has its own corners, and *no other object have any reference to them.*

15.2.3 Preserve independence - return copies

It would be quite natural for an object like the `Triangle` to have methods that allow the inspection of its corner points. For example, we might choose to implement:

```
class Triangle {
  private Point p1;
  private Point p2;
  private Point p3;

  ...
  public Point getPoint1() {
    return new Point(p1);  // return a copy
  }
}
```

Similar methods also exist for the other two corners. In order to preserve indepedence, we must make sure that the references to the points that implement the triangle are not revealed to outside code. Again, the copying constructor in class `Point` becomes useful. Rather than handing out or precious reference to point `p1`, we create a copy and return the reference to that. What the caller does to it we do not know, and do not care for. We have maintained independence and encapsulation.

15.2.4 References to immutable objects

The whole need for copying valuable resources, springs from the fact that `Point` instances in the triangle are *mutable*. They can be changed, and if a point is changed all contexts that share a reference to that point will change as well.

The problem would go away if the resource could not be altered. Immutable resources can be shared liberally, because they can never change. A `String` instance, for example, holds exactly the same text throughout its life. Likewise, wrapper classes like `Long`, `Double`, and `Integer` are also immutable.

While immutable objects can be safely shared, it is not trivial to decide if a data holding class like `Point` should be immutable or not. The price for immutability increases rapidly with the number of instance variables in the object. The program designer must evaluate, if the copying of mutable resources is preferable over the creation and dropping of immutable ones.

Finally, it should be acknowledged that sharing references to resources is an efficient way for objects to communicate and to share a common data model as it is being updated. It does, however, require careful planning and management. Sharing references should not be the default decision, as it easily leads to errors and bugs that are difficult to trace down and find.

15.2.5 Instantiation in the declaration

Instance variables in a class are always given default values when they are created. If the source code provides no value in the variable declaration, then the compiler inserts an initialization value of zero, `false`, or `null`. Even when these defaults are acceptable, it may be advantageous to explicitly initialize the variables, to let readers of the source code see what the initial value is intended to be.

For reference type variables, the default value `null` can be problematic. Assume that we have an instance variable that refers to a `String`. It could be anything, like a name, or an address, or the title of a favourite movie. Assume further that, as the program runs, this variable is only assigned in some object instances. This is quite common, especially if the variable describes some property that not all objects have.

So, now we have a design decision. What should be the default for this string variable? We could choose `null`, that would work, as long as we remember not to reference the string without checking first if the value is `null` or not. Another possible choice is to initialize it to the empty string `""`. This buys us some advantages:

- We can write the processing code under the well-founded assumption that the variable will never be `null`.
- In listings and printouts it will not print anything. Printing a `null` reference will most often print the text `null`.
- There is no memory penalty (at least not for class `String`) because the compiler will only create one empty string and then reuse that. This is safe because class `String` is immutable.

While the empty string is an easy case, other classes may not have object states that are obviously empty, or unused, or whatever kind of characterization one would like to apply. For example, an instance of class `Point` which is not a valid point, is difficult to imagine.

In more complicated instances, an approach called *lazy* or *late* instantiation can be used. Remember that we can hide instance variables behind methods (and we should do so in many cases). We therefore have the design option to defer initialization of one or more instance variables until they are actually needed.

As an example, imagine that you have a program with 10000 object instances. During a normal run, some 2500 of these objects will contain a small list of integers. We do not know in advance which object instances, and it would be a waste of memory to insert a list object in each instance by default. So what we do is this:

```
import java.util.LinkedList;

class Item {
  private LinkedList<Integer> nums = null; // no list yet

  // A number is added to our list
  public void addNumber (int n) {
    if (nums == null)  // if we have no list, create one
      nums = new LinkedList<Integer>();

    nums.add(n); // add the number to the list
  }

  // Return the size of the list
  public int size () {
    if (nums == null)
      return 0;
    else
      return nums.size();
  }
  ...
}
```

In the above example, the actual list object is not created until a number is to be added to it. Likewise, when the size of the list is requested and we have no list, we just return zero, even though we do not actually have a list. The caller will not be able to tell the difference.

Similar techniques exist for other common methods. If someone wants a reference to our list, we return a copy if we have a list, and a new empty list otherwise. The same tactic can be used to serve a request for an iterator.

15.3 Classes within classes

Classes are usually defined in source files with the same name as the class. However, the Java language offers the possibility to place a class definition *inside* another class. There are two possibilities, *nested* classes which are ordinary classes except that they are defined inside another class, and *inner* classes where the inner object instances has a special relationship with the object that spawned them.

15.3.1 Nested classes

A class often needs a resource that is one or more instances of another class. If this resource is of a general nature which can be useful to several other classes, then it makes sense to create this as an ordinary class.

However, if the desired resource is highly specialized and only relevant to the class using the resource, then it may be prudent to create it as a nested class.

A nested class is a static class declared inside another class.

Here is an example of a nested class:

```
public class Person {
  String name = "";
  String personNumber = "";
  Address address = new Address();

  public static class Address { // nested class
    String street = "";
    String postNumber = "";
    String city = "";
    ...
    public String toString() {
      return String.format("%s,%s,%s",
                  street, postNumber, city);
    }
  }
```

```
    ...
    public String toString() {
      return String.format("%s,%s,%s",
        name, personNumber, address);
    }
  }
```

The outer class is `Person` and the nested class is `Address`. Class `Address` is declared `public` which means that any class can create an instance of it if they want to. The name of the nested class is `Person.Address`.

Class is `Address` is declared as `static`. For a nested class the keyword `static` means that objects can be created from it independent from a particular instance of the surrounding class. A new `Address` instance can be created without having a `Person` instance around.

If class `Address` had been declared `private`, then only class `Person` can create instances from it. It would be a `private` resource to class `Person`, and hidden from outside view.

Since the nested class is `static`, it cannot refer to non-static fields and methods in the surrounding class. It needs an explicit object reference in order to do so.

15.3.2 Inner classes

Inner classes are declared just like nested classes, with the difference that the `inner` class is not declared `static`:

```
public class Person {
  ... // as above

  public class ResidentialPerson {
    public String toString() {
      return String.format("%s,%s", name, address);
    }
  }

  public class SocialPerson {
    public String toString() {
      return String.format("%s:%s", pnr, name);
    }
  }
}
```

This is a contrived example in order to demonstrate the mechanism. Class `Person` has (in addition to the members shown further above) two inner classes called `ResidentialPerson` and `SocialPerson`. These provide, by virtue of their `toString` methods, different views of a `Person` instance. If that was all we wanted, it could have been done with ordinary methods. But consider if the view required some additional computation, like summary statistics, or database lookups. Then the inner class is much more motivated for this kind of application.

It is not possible to create instances of an inner class from outside the surrounding class. Only an instance of the surrounding class can do so. In the above example, an instance of class `Person` is required to create an instance of class `ResidentialPerson` or `SocialPerson`.

Furthermore, when an inner class instance is created, it receives a reference back to the object that created it. The reference is (mostly) invisible and is managed by the compiler. The reference to the creating object instance enables the inner class to refer to variables and methods in the surrounding class. The reference goes to the object instance from which the inner object was created.

We could use this in the following way. Assume we have created a person (suitable constructors must be assumed):

```
Person p = new Person("Eva Asp", "720823-1234",
                      new Person.Address("Fasangatan 31",
                                         "123 45",
                                         "Lund"));

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

The above code prints:

```
Eva Asp, 720823-1234, Fasangatan 31, 123 45, Lund
```

We now wish to create and print a `SocialPerson`. In order to do so, we need the reference to `Person` from which to extract the view:

```
Person.SocialPerson rp = p.new SocialPerson();

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

We follow the reference in `p` to the object before using the `new` operator. The returned object will execute in the context of `p` and print:

```
720823-1234, Eva Asp
```

15.3.3 Internal dynamic data structures

Even though the Standard Library offers a lot off commonly used data structures for lists and sets, an application sometimes find it appropriate to define their own. A dynamic data structure is one that grows and shrinks over time. As an example, we will show three simple lists of `int`.

An array-based list

An array-based list uses an internal array to store the list elements. The advantage of using an array is that reading from the list is efficient, and that memory usage is efficient. The disadvantage is that extending the list requires some extra effort. Here is (the outline of) the array-based list class:

```
public class Alist {
  private int [] ar = new int [0];

  public void addLast (int n) {
    int [] h = new int[ar.length+1]; // Create a new array
                                     // one element longer

    for(int i = 0; i < ar.length; i++) // Copy from old
      h[i] = ar[i];                    // to new array

    h[h.length - 1] = n;            // insert new element
                                    // last

    ar = h;                         // use the new array
  }
}
```

As you can see from the listing, each time we add an element to the array we must create a new array one element longer, copy all the old elements to the new array, and then add the new element. This means that as the list grows, it will be increasingly more time-consuming to add a new element, because of all the copying that must be done. Similar considerations apply to insertion and deletion of elements.

We also want to offer a `size` method, that tells the caller how many elements there are in the list. With an array-based list, this is trivial and fast. We just return the length of the current array:

```
public int size () {
  return ar.length;
}
```

An array-based list with capacity

If we modify the array-based list slightly, we can get a lot of better performance in exchange for a little extra complexity. What we do, is that if an added element does not fit in the array, we allocate a new array with *extra* space for new elements. We allocate some extra capacity, so that when more elements are added to the list, we can simply insert them. With a properly selected capacity, the unused extra memory will be fairly low, and the performance gain on list manipulation fairly high. This is how it looks:

```
public class AClist {
    private int extra = 16;
    private int [] ar = new int[extra];
    private int index = 0;

    public void addLast (int n) {

      // If the old array is full, allocate a longer array
      // and copy from old to new
      if (index == ar.length) {
        int [] h = new int [ar.length + extra];
        for (int i = 0; i < index; i++)
          h[i] = ar[i];
        ar = h;
      }

      ar[index++] = n;  // Insert new element
    }

    public int size() {
      return index;
    }
  }
```

It should be realized, that with an extra factor of 16 elements, we will only have to allocate a new array and copy on every 16th addition. That is obviously faster than doing it on each addition.

The capacity-based array-list is, however, not any better when it comes to operations that insert an element or remove an element before the last element. In such cases elements must still be moved around.

A second issue is how to deal with a list that initially grows to a very long length, and then shrinks again down to, say, 10% of its maximum size. If all we do upon deletion is to move down the `index`, we now have 90% of that memory unused. We could add some extra code to replace a long array with a short one, when the unused capacity of the array becomes too large, that would work. But we cannot generally foresee what size of `extra` is optimal. For that reason, we may allow the user of class `AClist` to hint about a good value. For example:

```
  public void capacityHint (int capacity) {
    extra = Math.max(16, Math.min(capacity, 32000));
  }
```

This code allows the using code to set the capacity between 16 and 32000 elements.

A linked list

A linked list is constructed from a set of *nodes*. Each node contains the payload (an `int` in this case) and a reference to the next node. If the node is the last

```
first:null
last:null
```

Figure 15.3: The empty linked list

one in the list, the next reference is `null`.

The list is modelled by class `Nlist`. A node in the list is modelled by the nested class `Node`:

```
public class Nlist {

  private static class Node {
    public int value; // payload
    public Node next; // next node

    public Node (int value, Node next) {
      this.value = value;
      this.next = next;
    }
  }

  private Node first = null;
  private Node last = null;
}
```

With class `Node` being `static`, each instance of `Node` has no special relationship with the object that created it, but because it is declared `private` class `Nlist` is the only class that *can* create instances of `Node`.

To manage the whole list two variables are used, `first` and `last`. These refer (point) to the first and last node in the list. When the list is empty both are `null`. Figure 15.3 illustrates the empty list.

A linked list generally has better performance than an array-based list because the time it takes to add an element first or last does not depend on the size of the list. Insertion and deletions in the middle of the list, requires a search which *does* depend on the size of the list, but once the spot is found the actual insert och delete operation is fast. The penalty for using a linked list is a slight memory overhead, because each element requires one or more extra variables to manage it.

When adding a new element (integer) to the end of the list, there are two different situations that need to be managed:

- If the list is empty, both `first` and `last` must be updated.
- If the list is not empty, only `last` is updated.

```
public void addLast (int value) {
```

```
first--->[Node next:null]
         ^
         |
last-----+
```

Figure 15.4: The one-element linked list

```
first--->[Node next]
                |
                V
         [Node next]
                |
                V
         [Node next]
                |
                V
last---->[Node next:null]
```

Figure 15.5: The linked list with four elements

```
    Node n = new Node (value, null); // create new node

    if (first == null) // empty list, special case
      first = last = n; // both list vars updated
    else {
      last.next = n; // link in new node last
      last = n;      // and remember where new end is
    }
  }
```

When adding a new element to the end of the list, we know that the new node will be the last one. Therefore when we create the new `Node`, we know that the `next` reference must be `null`. Figure 15.4 shows the list after the first element has been added to it.

If `first` is null, the list was empty, and so we make both `first` and `last` point to the single node. Otherwise, we update the previous last node to point to the added node via its `next` field, and finally set `last` to point to the `n`, the new last node in the chain.

A common question is how long the list is. The method `size` counts the number of nodes in the list (see also figure 15.5):

```
  public int size () {
    int count = 0;

    for (Node n = first; n != null; n = n.next)
      count++;

    return count;
```

```
}
```

Note how the standard for loop is using a reference variable. The reference variable is updated by following the next variable to the next node. When the end of the list is reached, we know that it will be null and the loop exits. It also works for the empty list, because then first will be null and the loop never gets going.

The method insert allows us to insert a new value before a given position (first position is zero). Since we do not trust the calling code to get the index right always, we check it first. Note that the implementation expects the list to be non-empty:

```
public void insert(int index, int value)
  throws IndexOutOfBoundsException
{
  if (index < 0 || size() <= index)
    throw new IndexOfOfBoundsException("" + index);

  if (index == 0) // special case, var first is modified
    first = new Node(value, first);
  else {
    Node n = first; // hop index-1 jumps down the list
    for (int i = 1; i < index; i++)
      n = n.next;

    n.next = new Node(value, n.next);
  }
}
```

Dynamic lists are often used in programs. The choice between an array-based list or a linked list should be based on the needs of the program. The Standard Library includes both kinds of lists. In general one can say, that memory considerations and the expected dynamics of the lists used, should determine the choice of implementation. In addition to that, if the list is used like a stack (mainly addition and deletion at the end), with minimal lookup inside the list, then the array-based list works well. If the list is used like a queue, with addition to the head and deletion at the tail, then a linked list is preferable.

If locating a particular element in the list is a frequent operation, then it probably is a good idea to use a sorted list and binary search, or an ordered tree data structure. Both of these can locate an element in logarithmic rather than linear time.

Chapter 16

Developing new object types

This chapter is about creating new object types. Another way of putting it, is to say that this chapter is about designing and implementing new classes. In doing so, the chapter also has a significant taste of software engineering to it.

16.1 Defining a new type of object

The definition of a new type of object should begin, just as for any type of software, with requirements. These requirements may be informal, some notes on a piece of paper, or they may be highly structured and vetted by peer review.

The whole point is, think first, write it down, *then* program. The reason for this is that while you can think of a good number of relevant and clever things, you are likely to forget some of them as the next great idea shoulders aside the previous one.

16.1.1 Modelling, design, revision

Form an idea of what the object type is to be used for, and perhaps more importantly, what it is *not* intended to do. Write the model down. Your first consideration should be, *what does it do?* Only later, when the design has matured, should you begin to consider *how do I make it do that?*

The new object type should have recognizable features and properties, comparable to other object types. You need to consider a number of items:

1. Will the object type instances be used in large numbers, or in small quantities? An object type from which many instances are created should waste no resources, as each extra expense will be multiplied in each instance.
2. Is this an object type that carries data between and in other objects? If so, should it be immutable, highly encapsulated, or generally accessible?
3. What properties of the object should be accessible to the outside, and what must be hidden within?

It is difficult to give general rules, but some old internet wisdom dictates that when you have come up with a first design, revise it. Remove as much as you can without breaking any requirement. When nothing more can be removed, you have reached a reasonable starting point.

Do not hesitate to try alternative designs. It is much more efficient to do so during the design stage, than after hours of tedious coding.

Explore the Standard Library classes carefully. Perhaps there is already an object type defined that can be used or extended. However, avoid twisting a library class into a purpose for which it was not designed. You will be driving a square peg into a round hole. Instead, borrow any ideas you find appropriate into your own design.

16.1.2 Choosing instance variables

Most object types have at least one instance variable. The instance variables should be chosen to carry the data the object needs, and to do so in an efficient way. It does not have to be optimal, just good enough. As an example, assume that we are designing the `Person` class for a Swedish citizen web service application, modelling a real person. Here is a first outline:

```
public class Person {
  private String firstName = "";
  private String lastName = "";
  private long personNumber;
  ...
}
```

This design seems to make sense, but on the first project group meeting several issues are raised:

- Last and first name is a cultural convention that no longer works that well. People can have both more and less than two names, and often do if all officially registered names are included.
- The digits in the person number will fit in a `long`, but then it will not be possible to represent temporary person numbers (that include letters),

nor will it be possible to accurately represent the + standard for people older than 100 years.

So, after some revision the Person type is redesigned to:

```
public class Person {
  private String fullName = "";
  private String displayName = "";
  private String personNumber;
  ...
}
```

This simplified model has better chances to accurately represent the names people are registered with, and what they like to call themselves. In addition to that, the person number can be stored correctly according to the standard of the Swedish Tax authorities.

In this example, the only change in the name variables concerned the naming of the variables. But these names also reflect a different view of how the variables are to be used, and that is the important change.

A similar example can be made for class Point, the representation of planar points. We know we need two variables in there to represent the x and y coordinates. But what datatype to choose for them? Should they be integer or floating point? The answer to this question is not trivial, because it depends on how we imagine the points to be used. If all they ever will describe, are pixel positions on a screen, then it might be possible that integer coordinates is what we need.

On the other hand, if we think that our points are components in some mathematical model, then we should employ the highest precision storage we can, in order to delay any accumulation of roundoff errors in the computations. We can always map from floating point to integer should we need to, but we cannot go the other way very easily. Thus we settle for:

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

Now, this will require 16 bytes per point instead of 8, but at least we do not sacrifice precision. Choosing the double datatype also has implications for the methods in the class, but that does not pose a problem.

16.1.3 Accessor and mutator methods

Continuing with class Point, we must now consider in what ways the coordinates can be accessed and modified (mutated). It is already in the requirements that a `Point` must be mutable, so that we can apply geometrical transformations like scale, translate, and rotate to any structure it happens to be a part of.

As a result of those requirements we need to be able to hand out our coordinates and to accept new coordinates. We initially come up with the following suggestion:

```
public double getX()         // (1)

public double getY()         // (2)

public double [] getXY()     // (3) higher cost

public void setX(double x)   // (4)

public void setY(double y)   // (5)

public void setXY(double []) // (6) higher cost
```

This appears like a great idea. However, methods 1, 2, 4, and 5 is a set of methods that is necessary and sufficient to fulfill the requirement. Methods 3 and 6 is another such set, but with the extra expense of creating, reading, and writing small arrays, because that is how those two methods communicate. Taken together, methods 1-6 form a set of methods in which there is redundancy, without there being any redundancy in the requirements. As a result, methods 3 and 6 are removed from the design, because they are the more expensive ones.

16.1.4 Helper classes

Sometimes it is convenient to design helper classes for the primary object type. For example, the nested `Address` class for class `Person`. Such a helper class need not to be nested, though. If it is likely that it can be generally useful to other classes it may be better placed as a separate class. The only compelling reason for creating an *inner* class, is if the helper object should have a reference back to the object that created it, so that it can access its members.

16.1.5 Defining constructors

An important consideration is the design of the constructors. The constructors support various modes of creation, and we need to consider the application as well as other situations.

In our `Point` example we come up with the following design:

```
public Point () // default constructor
```

Having a default constructor is viable, because there are mutator methods that allow the coordinates to be set separately, if that is required. Also, the state of a `Point` is uncomplicated and needs no internal consistency management.

```
public Point (double x, double y) // standard constructor
```

The standard constructor is expected to be the one most used. The coordinates are supplied and the `Point` is initialized.

```
public Point (Point p) // copying constructor
```

The copying constructor is highly desired, since it will help the classes that are using the points to maintain independent references.

```
public Point (String s) // text argument constructor
```

The ability to create a `Point` from a text string like

```
"-6.78664 22.34432"
```

is in the requirements, because the application designer has stated the need to be able to save `Point` structures in text files, and later restore them. However, as the project group debates this, there are several different points of view:

- It is not necessary to have a string constructor, because code in some other class could read the string, parse the coordinates, and then use the default or standard constructor.
- The creation of a `Point` from a coordinate string could be placed in a static factory method, as a class resource.
- It is better to have the string constructor, because then programmers will see it together with the other constructors.
- The code that writes the saved text should be close to the code that later reads it back in. Since no other object is likely to have to know exactly how it is formatted, it makes good sense to have the `Point` write the text and later parse it in a constructor. This increases encapsulation and symmetry between writing and reading.

As a result of the discussion, the last argument is accepted as the best option, and the string constructor is decided.

16.1.6 Designing services

Two services related to points are introduced. The first one returns the distance between the `Point` and the origin (0, 0) of the coordinate system:

```
public double distance ()
```

The second service returns the distance between two points:

```
public double distance (Point a)
```

16.1.7 Combiner methods

A combiner method is a method that combines two instances, and produces something which is dependent on both of them. As an example, we can create the method:

```
public Point midPoint(Point a)
```

which returns a `Point` located at the middle of a straight line between the current `Point` and `a`.

16.1.8 Comparators

Comparators compare two objects and return a scalar value that can be used to order them, perhaps for sorting purposes. For points this is a little contrived, but for the sake of the example we will use the distance from the origin as the scale on which points are ordered. Points closer to the origin comes before points farther from the origin. The method itself is borrowed from the `Comparable` interface in the standard library:

```
public int compareTo (Point a)
```

It returns a negative value if the current `Point` is lesser than `Point a`, zero if they are equal, and a positive value if the current `Point` is greater than `a`.

16.1.9 Predicates

Predicates are methods that test a condition and then return either `true` or `false`. An often used condition is that of equality, so we want that one in there, overridden from class `Object`:

```
public boolean equals (Point a)
```

If the two points have exactly equal coordinates, they are equal. We expect this to be true sometimes, but we require it to be true in the following case:

```
Point p0 = new Point (Math.random(), Math.random());
Point p1 = new Point (p0); // the copying constructor

p0.equals(p1); // This must be true
```

We expect a copied `Point` to be `equal` to the original.

A second observation is that with double precision variables, roundoff errors may creep in, and we may want to know if two points are *close enough.* We can do that by offering a proximity test against a supplied maximum distance:

```
public boolean inrange (double maxDistance, Point a)
```

16.1.10 Converters

Converters (or transformers) provide a different representation of the object, in whole or int part. We have two such methods, which are somewhat similar. First out is the ubiquitous `toString` method:

```
public String toString ()
```

The purpose of the `toString` method is to return a text representation of the `Point` that can be read and understood by humans. Therefore it requires a bit of formatting and grouping, and it is not necessary to give all possible decimals of precision in the coordinate either. This is an example of the output format:

```
(-34.67, 7.54)
```

The second method is used to save the coordinates of the `Point` to a file. We could just return the appropriate string and expect the caller to write it to a file, but that does not protect against it being manipulated or abused. So instead we accept a reference to a `PrintWriter`, because we are writing text, not binary data. Then we print our saved text on the `PrintWriter`:

```
public void saveOn (PrintWriter pw)
```

This output text must unambiguously preserve all the data and there is no need for any extra characters except for a separating space. The output is similar to:

```
-34.666678456323456 7.54323425652343
```

16.1.11 Static variables

Static variables are best used for constants. Non-final static variables present potential problems in that they only exist in one instance (for the program running), are not thread-safe by default, and may cause the state of created object instances to be dependent on the global program state and its history. See also section 15.1.1.

16.1.12 Static methods

Static methods are useful to provide services that are not dependent on a particular object instance. As an example of such a method, we choose `getRandomPoint`:

```
Point getRandomPoint (double width, double height)
```

Static methods can also be used to advantage to create a library of services and functions that apply to the usage of the class.

16.2 Implementing the definition class

When the fields and methods for the definition class have been designed, one can go on and implement them. Since the method signatures have been established, the bodies of the methods are filled in.

At this time one can no longer procrastinate about the choice of algorithms, data structures, efficiency, and performance. However, even though one should pay attention to these issues and make the best engineering decisions available, the following things are equally important to consider when implementation begins:

- The first implementation should prioritize clarity and correctness over maximum performance. It should be the bug free reference implementation.
- Unless the circumstances are very demanding, the reference implementation is usually good enough.
- If further optimization is found to be required, having a correct reference implementation available is a great validation tool.

To reiterate: the choices of algorithms and data structures used in the class and in the program, will to a large extent determine its efficiency and maintainability. Choosing well is basically what software engineering is about.

Making poor or good choices can be illuminated further by this example:

> The programmer finds that at one or more points in the program, a sequence of elements need to be sorted. When it comes to implementing this, the programmer is faced with four different alternatives:
>
> 1. Implementing once again that simple sorting routine[1] that was among the first ones learnt

[1]For example Bubble sort, Insertion sort, or Selection sort

2. Selecting and calling an existing sorting method in the Standard Library
3. Finding and implementing a sorting algorithm from literature
4. Inventing and implementing a new, more efficient sorting method

Which alternative do you think is the best choice for the first implementation?

The programmer should select alternative 2. It is the best choice, because the library routine is very likely to be a good general algorithm, efficiently implemented, and professionally debugged after years of exposure to thousands of projects.

However, what sometimes can be seen among less experienced programmers, is a tendency to fall back to alternative 1, and write their own code from memory. Doing so, does not only introduce the perils of relaying on memory, but the algorithm itself if often trivial and does not scale well when demands and sequence sizes increase. That algorithm was taught in school or in a book because it is simple enough to demonstrate the *concepts*, not the optimum.

Finally, alternative 3 should only be considered if the library routine is faulty or unavailable. Alternative 4 is likely to cost a lot with little returns, unless it is such a central and critical part of the whole project that it is considered essential to pursue.

16.2.1 Constructors

Having more than one constructor may be convenient, especially if there are appropriate defaults available. More than three different constructors is rarely practical, though.

It is often the case with multiple constructors that they perform code that is common to all constructors. Rather than repeating that code in all constructors, it is sometimes better to place it in a private method and have each constructor call that method. That way it is easier to maintain changes to the common code.

When a class is extending (inheriting from) another class, the invisible first step of the subclass' constructor is always to call the default constructor of the superclass. The call is inserted by the compiler as commanded by the Java language specification.

If the superclass does not *have* a default constructor, then the compiler requires that the programmer inserts, on the first line of the subclass' constructor, an explicit call to an explicit constructor in the superclass. This is quite natural, considering that the code inherited from the superclass should be initialized before the code in the subclass.

16.2.2 Instance methods

Instance methods are written as by the design. During this time it is not uncommon that certain realizations call for adjustements or partial redesigns. Such can often be accomodated without too great an effort. However, it is important to understand that if the change is too great, it will impact the whole class, and it may be better to go back to the design stage and review it for implications and changes.

Implementing instance methods often leads to the realization that supporting or auxilliary methods are required. These should of course be provided, as they help make the code and program structure more clear and easier to debug.

Known limits on parameters for methods should also be checked. If a critical parameter is out of limit, it is better to catch it early on and throw an exception, rather than relying on the JVM throwing an exception later on, possibly from some obscure corner deep in the method call stack.

Note here that by *critical* or *invalid* is meant such parameter data that will crasch the program or generate a result of too poor quality. For example, if one parameter specifies an index, is that index within the bounds of the kept elements? If another parameter specifies a value used to divide, is it zero or too close to zero? Is a length positive? Is a requiredd reference non-null?

Early rejection of invalid input makes the program more robust. Optimally we would like to check input and reject invalid data as soon as possible. However, reading data and processing it may be done in completely different parts of the program, by different classes. We do not want to put internal performance details of our definition class into the data entry routine. This does not refer to general abnormalities that may benefit the whole program, but to details; parameters that we would like to be encapsulated in our definition class so that we can update them should need be, without having to review code in several classes (with the risk of forgetting a spot).

Again, there is a tradeoff here, and the general guideline is that your first choice should be to go for a late check, i.e. postpone the parameter check until the methods in the definition class are actually called. If parameters are unacceptable to the class, then it will throw the corrsponding exception when someone attempts to use it.

On the other hand, in those rare cases where it is deemed worthwhile to actually do the check before the actual method call, it turns out that object-orientation provides us with several ways of maintaining encapsulation without sacrificing efficiency. The answer lies in the category of methods called *predicates*.

When input data is read (as an example of a remote relationship), the input code is written to verify data by calling the appropriate predicate method in our definition class. These predicates can be static methods, if we have not created any instances yet, or they can be instance methods, if we have internal state to take into consideration. The benefit, however, is that the input code can now be completely agnostic about what is a good or bad value; all it needs to know is which predicate to call, and how to interpret the return value.

It is important to note here, that the predicates do not throw exceptions. If the predicate tells the input data routine that a value is unacceptable, then the data input code may throw an exception, or it may just discard that particular input and move on to the next. The decision to throw an exception is made by the caller of the predicate.

Later on during processing, when methods are called in the definition class, and predicates have been implemented, then the methods should call the same predicates for checking, and throw an exception if the data is unacceptable. There are two reasons for this. The first is that the class must not assume that its public predicates have been used and properly heeded prior to doing a method call. There is simply no way of knowing this. The second reason is, that with predicates, there is a single point of verification and external callers and methods should obviously use the *same* verification. Otherwise the program will be inconsistent.

Now, it is important to understand that all this checking of parameters primarily applies to *external* inputs, because the program has no control over what the user types, or what is found in a file. The same reasoning applies to library routines that are expected to be shared and used by other, unknown programs.

A method that is called with arguments generated internally, by the program itself, can be expected to only receive valid parameter values if the program is self-consistent, i.e. free of bugs. This applies in particular to private helper methods. If the program is well written, helper methods are shielded from illegal parameters by their more exposed callers. For such internal methods parameter checking is mostly required in the debugging stage, and for that the `assert` clause (see 16.3.3) is highly recommended.

16.3 Testing

There is no such thing as bug-free software.

As soon as the implementation of the class is runnable, you will want to do some testing. If it is a complicated class, it may even be advantageous to implement it in phases, so that you can test each phase separately. If a later phase relies on the proper operation of a previous one, development in phases helps in establishing where to look for problems.

Testing basically has two purposes. The first is that you want to exercise the software so you can fix it when it crashes. This is a manual labour, for the most part. Second, when it no longer is crashing, you want to verify that the state, behaviour, and outputs match what is expected from the inputs. This part of testing can be automated in various ways, but fixing any problems is still hands-on work.

Testing can be done with various degrees of ambition. A few of these are outlined in the next section.

16.3.1 Creating and displaying objects

A very simple way of testing a class is to create instances of it, and then display it. If the `toString` method has been properly implemented the information presented by it may be quite sufficient.

The next level of ambition is to introduce *trace printouts*. In the simplest case, this is just a line of printout inserted into the code at interesting places. For examples:

```
...
distance = time * speed; // application code

System.out.println("Distance = " + distance); // Trace output
```

When the line of trace output is no longer needed, it is commented out. When it is *really* no longer needed, it is removed.

Simple trace output is appropriate for small classes, like `Point` or `Triangle`. More complicated classes with 200 lines of code or more, may require a more developed tracing. This is how it can be done:

```
public class Mesh {
  public static int debugLevel = 2; // Controls trace outputs
  ...
  // Prints trace output
  private static void dmsg(int level, String msg) {
    if (level <= debugLevel)
      System.out.println("DEBUG[" + level + "]:" + msg);
  }
  ...
  Point p = new Point(-1, -1); // Item to be inspected
  dmsg(1, "Point p=" + p.toString()); // Trace output
}
```

In the above code, class `Mesh` has been given some extra debugging features. The class variable `debugLevel` determines what the current debugging level for the class is. Because it is `static`, it applies to all instances.

The method `dmsg` is called to generate a potential trace printout. The `level` parameter determines if the message is printed or not. The provided `level` must be less or equal to the current `debugLevel` in order to generate a print. This means that if `debugLevel` is set to 2, all trace outputs at level 0, 1, and 2 are printed, but any higher (and hence even more detailed) level messages are not printed. In most cases more than three levels of debug messages are not needed. The higher levels should be used for trace printouts inside loops, as they can generate a lot of output.

The use of a class variable and method makes it possible to quickly change the level of debug output when required. The method can also be modified to send the trace printout somewhere else, like `System.err`. The standard header on each trace line also makes it easier to find the trace printouts when they are mixed with ordinary program output.

16.3.2 Source code debugger

A source code debugger is an invaluable tool when looking for those non-obvious bugs. The Eclipse IDE has one. When using the debugger, you first set *breakpoints* at source code lines of interest. Then the program is run in debug-mode.

When the program reaches a breakpoint execution is suspended, and the debugger shows the current line, the currently active variables, the call stack, and more information. From this point on one can *step* to the next line of source code, *step over* or *into* a method call, *finish* the current method, or *continue* running to the next breakpoint or the end of the program, whichever comes first.

Running the program with a source code debugger requires some practice and a high degree of concentration, but it can be time well spent, especially on nastier bugs that have no obvious origin.

16.3.3 Assert

Asserts are inline predicates that are tested at runtime. They are used to test assumptions and conditions that are to be `true` when the point of execution passes through the assertion. They look like this:

```
int t
...
assert 0 < t;        // simple
assert 0 < t: "t not positive"; // with message
...
```

When the assertion is evaluated, and the condition evaluates to `false`, the JVM throws an `AssertionError`. If the message expression is present, it is evaluated and the result is placed in the error.

Assertions can be disabled or enabled in the JVM. They are disabled by default, so one must explicitly turn them on in the JVM, for example with:

```
java -ea ...
```

In general, assert clauses should be left in the program. Their great benefit comes when the code is modified. Running it again with assertions enabled will quickly reveal if any of the held assumptions and conditions guarded by the assertion have been violated by the update.

Asserts should *not* be used to check parameter and input limits! The most obvious reason for this is that assertions are turned off by default, and can be turned off by anyone running the program.

Please see the official documentation for more information[2].

16.3.4 Test program

Writing test programs is also a good strategy. A test program should test both valid and invalid inputs to make sure the response is the expected one. A test program can also sometimes perform testing with random parameters and thus quickly subject the tested code to millions of different inputs.

A test program should be small and test a well defined portion of the software. For the next portion, a different test program is written. This fine-grained modularity makes it easier to manage the suite of test programs. Over time, the project accumulates several test programs that together test many different parts of the software. Such a collection is a great help in performing automated regression testing, which means that for each new version of the software, all the test programs are run and should pass. This ensures that the software does not *regress* to a bad state because of the update.

16.3.5 Junit

Junit is a Java test framework supported by Eclipse and some other integrated development environments. Junit perform the same service as test programs, except that they are organized as classes that do not require a `main` method. The junit engine loads all tests associated with a project, runs the tests, and reports the result.

[2]`http://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html`

16.4 Comments

Comments in the code are essential to explain what the code is doing and why. Comments are written from one programmer to another, and very often the recipient is the future version of oneself.

Code without comments is harder to understand, because despite what can be heard by some programmers, code is not self-explanatory except in the most trivial of cases. The program code talks of small things: numbers, indices, method calls, expressions. It instructs the computer what to do, but does not reveal *why* it should do it, nor why it is done in that particular way.

Comments should provide that extra knowledge which cannot be read from the program code itself. Just like finding good identifier names is an art, writing good comments requires some practice. The reason that is, is because space is limited and very often you want to get on with the programming. But the comments are invaluable when you *come back* to the code to maintain it, and you have to (re)acquire the mindset in which it was written.

Comments in Java come in three flavours: two kinds of code comments, and Javadoc comments that provide text for the automatically generated documentation.

16.4.1 Code comments

Code comments can be directed at a single line, a block of code, a method, a whole class. The writer of the code must make the authoring decision. There are two kinds of code comments available, the single-line comment `//` which extends to the end of the line, and the multi-line comment `/* ... */` which spans from the starting token to the end token:

```
// This is a single-line comment

i = i + 1; // This is a single-line comment

// Several single-line comments can be used
// for a multi-line text.

/* This is a comment
   that spans multiple
   lines. */

/* This multi-line comment only uses one line */
```

The single-line comments `//` ends at the end of the line, and this limits where and how they can be placed. The multi-line comments offer full control over where they begin and end, and can thus be inserted anywhere. For example:

```
// Note: not sure about the root part, trying without it
```

```
y = x * k + /* Math.sqrt(i*u) + */ q;
```

The compiler does not see what is in the comments.

16.4.2 Javadoc comments

The documentation of Java code is usually embedded within the source code itself, using a special comment syntax. This makes it possible for the documentation tool `javadoc` to extract both comments and information from the code itself. The documentation for the Java Standard Library is produced in this way.

The `javadoc` tool reads the source code similar to how the compiler does, to extract names and type information. In addition to that, `javadoc` also looks at multi-line comments that begin with `/**` and end with `*/`. Such comments are read, and their contents inserted into the documentation. To enhance the structure of a Javadoc comment, certain rules and tags are used. However, here is first an example of a method documented Javadoc style:

```
/**
 * Returns the half of its argument. Calling this method is
 * exactly the same as dividing the parameter with two.
 *
 * @param d The value to halve
 * @return The halved value of the given parameter
 */
public int half(double d) {
  return d / 2.0;
}
```

Some simple rules for Javadoc text:

- Each Javadoc comment comes immediately *before* that which it documents.
- Javadoc is most commonly used to document classes, class variables (aka *fields*), and methods.
- The generated documentation is in HTML. Therefore, some simple HTML markup is allowed in the comment.
- The at-sign tags helps Javadoc to associate text with the corresponding entity. The text continues to the next tag or the end of the Javadoc comment.
- The following markers are good to know:
  - `@author` Documents the class author, usually by name and email
  - `@version` The version of the class

- `@param` Documents a method or constructor parameter
- `@return` Documents the return value for a non-`void` method
- `@throws` Documents why an exception may be thrown

Running the `javadoc` tool on your source files produces a directory with HTML files, according to the classes found. The files can be navigated and read with a standard web browser. A commandline argument to `javadoc` instructs the tool if documentation should be generated for private and protected elements too, or if only public elements should be included. The choice depends on the intended recipient of the documentation, as is illustrated in the next section.

16.4.3 Documentation

Documentation of software is written with different target audiences in mind. These audiences are for the most part:

1. Programmers and system developers who maintain and evolve the source code. Source code comments and full Javadoc documentation is for this group.
2. System administrators and power users who install, configure, deploy, and uninstall the software. For these, one should prepare as simple instructions as possible, use conventional installer tools for their systems, and make it easy to identify what actions to take to set up or take down the software.
3. End users who meet the running software through its programmatic interfaces. Depending on what the software does, this target group may be programmers and system developers for *other* systems, to which the software is an asset once it is available. They need to know the runtime APIs, protocols, procedures, and limits. If the software is a Java programming library, Javadoc documentation of public fields and methods should be provided.
4. End users who run and interact with the program directly through a user interface. The interface can be a commandline, a console style dialogue, a graphical user interface, a hardware interface, or a combination of all these things. These users need to know what actions are available, what they do, and how they can be undone if necessary.

Chapter 17

Inheritance

This chapter continues and extends the concepts first presented in 11.3.1.

Inheritance is an important concept in object-oriented programming. It is the idea that a general class can be extended and specialized. The specialization can express itself in various ways: the subclass may provide different or new functionality, it may have a different behaviour, it may restrict usage and services in certain desirable ways, or it may simply be a component in the type system for certain specialized purposes.

Review, for example, the `Shape` class hierarchy in figure 17.1 (this class hierarchy adds to the one in figure 11.4). The class `Shape` is a model of an abstract, geometrical shape in two dimensions. Since it is abstract, the concept of a shape, it is not possible to draw a `Shape`. It has no physical form in this model. In order to obtain a `Shape` that *can* be drawn, we must select one of the specializations: `Triangle`, `Rectangle`, or `Oval`.

```
Object
  +--Shape
       +--Triangle
       +--Rectangle
       |    +--Square
       +--Oval
            +--Circle
```

Figure 17.1: Vertical view of the Shape class hierarchy

17.1 Subclasses

In Java, inheritance is indicated by the keyword `extends` in the class declaration:

```
public class Triangle extends Shape {
  ...
}
```

This instructs the compiler that all that is defined in class `Shape` is also to be present in class `Triangle`. Fields, inner and nested classes, constructors, and methods declared in `Shape` will be present in class `Triangle`.

- Class `Triangle` is *an* immediate subclass of class `Shape`
- Class `Shape` is *the* immediate superclass of `Triangle`
- A `Triangle` *is a* `Shape`.

However, having inherited the elements from `Shape`, it does not follow that they are all *accessible* from `Triangle`. That depends on their access protection. A `private` variable in class `Shape` is not accessible from class `Triangle`, but it will be *present* in class `Triangle`, and memory will be allocated for it whenever an instance of `Triangle` is created.

That elements of a class can be hidden from a subclass may seem strange at first, but it is actually a way of providing vertical encapsulation. Using class `Shape` as our example, it may be that `Shape` is designed in such a way that only certain aspects of it should be possible to modify through subclassing. It could be that `Shape` contains services and data structures that subclasses should not try to modify. A very common reason for such a restriction is that doing so runs the risk of breaking the contract with other classes that expect a `Shape` to behave and respond in a certain way.

17.1.1 Extending an existing type

It is sometimes convenient to extend an existing class from the Java Standard Library. A typical example is when a custom exception is wanted. The subclassed exception might looks like this:

```
public class PointException extends Exception {
  public PointException (String msg) {
    super(msg); // call the constructor in Exception
  }
}
```

Class `PointException` extends `Exception` so that it can be used (`throw` and `catch`) and be treated identically to all other exceptions. It does not change the behaviour of the exception in any way. The only thing to remember is that we must supply a constructor that can insert a message string in the part inherited from class `Exception`.

Another recurring case is when programming with package `javax.swing`, the library for graphical user interfaces. A popular choice for custom graphical components is to extend class `JPanel`, and then supply a constructor and the method for how it is to be painted. All the other things inherited from `JPanel` are left untouched, because they control how the `JPanel` is integrated with other components in the GUI. To the GUI system, the object is still a `JPanel`, because it *is* a `JPanel`.

The Standard Library can generally be depended upon, because it is not likely to change drastically. Backwards compatibility is important. In the rare event of a library change, there will be notices and documentation.

Sometimes there are other classes, like application specific classes, that can be subclassed from, but there is a risk in doing so. The subclass introduces both expectations and a dependency on the superclass. The superclass is expected to continue to exist and behave in a certain way. If the maintainer of the superclass is not aware of this, the superclass may suddenly become incompatible with the subclass, or worse, deviate from the assumptions made on its behaviour.

17.1.2 Constructors in subclasses

There are few things to keep in mind when it comes to constructors in subclasses. The main issue is that the compiler will enforce that the first instruction in any constructor is a call to the constructor in the superclass. The compiler reasons similar to this, when compiling a constructor:

- *Is the first instruction in the constructor a call to a constructor in the immediate superclass?* If the answer is *yes*, then the compiler is happy and continues to compile the rest of the constructor. If the answer is *no*, it moves to the next item:
- *Does the superclass have a default constructor?* If the answer is *yes*, then the compiler inserts a call to the default constructor in the superclass, and then continues with the rest of the constructor. If the answer is *no*, the compiler stops with a compilation error.

In order to fully understand why the compiler sometimes stop with an error message that demands a constructor call, it must be remembered that there is no requirement to insert code for the default constructor unless there are statements that must be executed there. The compiler automatically inserts code to initialize instance variables, so it is not strictly required to do so in a constructor (although it may be good for clarity). This means that if a class has *no constructor*, it still has a default constructor, because the compiler inserts one.

However, if any non-standard constructor is specified, i.e. a constructor that takes parameters, then the compiler will *not* insert a default constructor. It could well be that the class designer has decided that the only way to create an instance is to supply parameters, and the compiler is not allowed to bypass that. This means, in turn, that if the class designer wants to have a default constructor in addition to the parameterized one, then that default constructor must be explictly inserted, *even if it has an empty body.*

Here are some further examples:

```
public class A {

  // A default constructor is inserted by
  // the compiler, calling the default constructor
  // in class Object

}

public class B extends A {

  // A default constructor is inserted by
  // the compiler, calling the default constructor
  // in class A

}
```

If we could see the result of the compiler's code insertions, it would look like this (you do *not* need to put this in your code, it is just to illustrate what the compiler does automatically):

```
public class A {

  public A () {  // Inserted by the compiler
    super();     // call default constructor in Object
  }

}

public class B extends A {

  public B () {  // Inserted by the compiler
    super();     // call default constructor in A
  }
}
```

Now, we add a non-default constructor to class `A`:

```
public class A {

  public A (String s) {
    ... // our code
  }
}
```

When compiling class `A`, the compiler discovers that there is now a non-standard constructor present. As a result it does not insert a default constructor. Intead it compiles the given constructor and inserts a call to the default constructor in class `Object`:

```
public class A {

  public A (String s) {
    super(); // Inserted by compiler, call default
             // constructor in Object
    ... // our code
  }
}
```

However, when the compiler then comes to class `B` it finds that it cannot insert a call to the default constructor in `A`, because there is none to call. Compilation of class `B` ends with an error message:

```
javac B.java
B.java:2: cannot find symbol
symbol  : constructor A()
location: class A
public class B extends A {
       ^
1 error
```

The compiler cannot find the constructor `A()`, which it expects. There are two ways to fix this problem, and which one is chosen depends on how class `A` is meant to be used:

- Add a default constructor to class `A`
- or, add a non-default constructor to class `B`, that in turn calls the non-default constructor of class `A` as its first instruction.

The first solution looks like this (class `B` is not changed):

```
public class A {

  public A () {} // Add default constructor

  public A (String s) {
    ... // our code
  }
}

public class B extends A {

}
```

The second solution looks like this (class `A` is not changed):

```
public class A {

  public A (String s) {
    ... // our code
  }
}

public class B extends A {

  public B (String s) { // Non-default
    super(s);           // constructor calling A
  }

}
```

Now the compiler is happy and compilation succeeds.

17.1.3 Methods in subclasses

A subclass can call methods inherited from its superclasses (unless access restrictions prevent this). When the method executes it will have access only to any parameters it receives, and the context acessible to its definition class. Most importantly, *when the method executes it does not know it is actually part of a subclass of itself.* That said, the method in the superclass is of course still part of the same instance, but it has no way of knowing about its subclasses. It can only manage itself, and perhaps any superclasses it may have in turn.

An example follows. It begins by defining the `Oval` shape (an ellipse):

```
public class Oval extends Shape {
  double radius1;
  double radius2;

  public Oval (double r1, double r2) {
    radius1 = r1;
    radius2 = r2;
  }

  public double area () {
    return Math.PI * r1 * r2;
  }

}
```

We then continue to subclass for a special case of the `Oval`, the `Circle`. The special condition is that $r_1 = r_2$:

```
public class Circle extends Oval {
  public Circle (double r) {
    super(r, r); // call the constructor in Oval
                 // with the same value for both
                 // radii
  }

}
```

As can be seen from the listing, class `Circle` extends `Oval`, and by definition it has just one radius, the one accepted by the constructor. The circle simply calls the constructor inherited from `Oval` with that very same value for both radii to make the ellipse circular. That will initialize the variables inherited from `Oval` to the same value.

When used, we get:

```
Circle c = new Circle(5.0);

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

with the printout:

```
78.53981634
```

Class `Circle` has inherited method `area` from class `Oval`, but when `area` executes, it has no idea that it is part of class `Circle`. It just proceeds with the calculation as specified for an `Oval`, using the instance variables `radius1` and `radius2` that by inheritance also are present in the `Circle` instance.

17.1.4 Overriding methods in the superclass

The simplicity and beauty presented by the `Circle` and `Oval` example does not happen often. More common is that we want to completely replace a superclass method, so that our own version of it is called instead. We want to *override* a method in a superclass.

Overriding a method is actually quite easy, just implement a method with the same signature (name and parameter list) as the one in the superclass. The compiler will make sure that the most special (deepest) version is called.

We can exemplify this by continuing with the `area` method. All we have to do is to introduce some code for class `Shape`:

```
public class Shape {

  public double area () {
    return 0.0;
  }
}
```

We said initially that a Shape cannot be drawn, it is only the concept of a shape. This makes any attempt to compute its area impossible, but for the sake of an example, let us assume we define it as having zero area, just like a point. We could just as well have defined it to be -1.0, or infinity. It does not matter, as our design expects each drawable shape to override Shape.area with something meaningful.

We have already shown above how class Oval and Circle define area for themselves. There is no need to make changes there, they already override area properly. What we need is to provide computations for the Rectangle and Triangle.

Here is class Rectangle:

```
public class Rectangle extends Shape {
  Point origin;
  double width;
  double height;

  ... // constructor

  public double area () {
    return width * height;
  }
}
```

Class Triangle is a bit tougher:

```
public class Triangle extends Shape {
  Point p1, p2, p3;

  ... // constructor

  public double area () {
    double a = p1.distance(p2);
    double b = p2.distance(p3);
    double c = p3.distance(p1);

    double s = (a + b + c) / 2.0; // semiperimeter

    // Heron's formula
    return Math.sqrt(s * (s - a) * (s - b) * (s - c));
  }
}
```

Now all subclasses override Shape.area, returning appropriate results for each one.

17.1.5 Access protection in the superclass

Variables, methods and other members in a superclass can only be directly accessed from a subclass when the members in the superclass are declared:

- `public`
- `protected`
- (default) and the subclass is in the same package[1] as the superclass

The subclass can never access `private` members directly, but it may be possible to use the resources indirectly through methods that *are* accessible.

17.2 Super- and subclass references

Instances of objects can only be accessed through references. The type of the reference (and access protection) determines what members can be accessed. To illustrate this, we add some features to classes `Oval` and `Circle` (nothing is removed).

First class `Oval`:

```
public class Oval extends Shape {
  double radius1;
  double radius2;

  ... // constructor
  ... // area

  public double getRadius1() {
    return radius1;
  }

  public double getRadius2() {
    return radius2;
  }
}
```

Then class `Circle`:

```
public class Circle extends Oval {

  ... // constructor

  public double getRadius () {
    return getRadius1();
```

[1]Remember that no package declaration puts the class in the default package.

```
    }
  }
```

Let us now examine what happens with member access, depending on the type of reference. Examine this code carefully:

```
  Circle c = new Circle(5.0);

  Oval  o = c; // valid, because a Circle is an Oval
  Shape s = c; // valid, because a Circle is also a Shape

  double d = c.getRadius(); // valid   (1)

  d = o.getRadius();        // invalid (2)
  d = o.getRadius2();       // valid   (3)

  d = s.getRadius();        // invalid (4)
  d = s.getRadius1();       // invalid (5)
  d = s.getRadius2();       // invalid (6)
```

We create a `Circle` and assigns the reference to variable `c`. Variable `c` has the type (reference to) `Circle`.

Then we create a variable `o` of type (reference to) `Oval`, and assigns the reference in `c` to `o`. This is perfectly ok, because `Circle` inherits from `Oval`, and is therefore also an `Oval`.

We create a third variable, `s` of type (reference to) `Shape`, and assigns that too the reference in `c`. This is also a valid assignment, because `Circle` inherits from `Oval`, which in turn inherits from `Shape`. Thus, a `Circle` is also a `Shape`.

We now have three different variables, `c`, `o`, and `s`. All three variables refer to the same object. That object is an instance of `Circle`.

Then in (1) we call method `Circle.getRadius`, using the reference in variable `c`. This works, because the type of `c` is (reference to) `Circle`, and class `Circle` has a method `getRadius`.

In (2), however, we attempt to call `Circle.getRadius` using the reference in variable `o`. But this variable is of type (reference to) `Oval`. Class `Oval` does not have a method `getRadius`. The compiler will issue an error at this source code line, and say that it cannot find symbol `getRadius`. The reason for this is that the compiler is restricted (on purpose) to do type checking that can be performed at the time of compilation. According to the definition of type `Oval`, it has no method `getRadius`, and the compiler must reject the access.

In (3) we use reference `o` to call method `getRadius2`. This is allowed, because that method is defined in class `Oval`. The type of the reference matches the method call.

Finally, in (4), (5), and (6), we use variable `s` in an attempt to call the

```
                Object
                  |
                Shape
                  |
    +------------+-------------+
    |            |             |
   Oval      Rectangle      Triangle
    |            |
  Circle       Square
```

Figure 17.2: A horizontal view of the `Shape` hierarchy

methods defined in `Circle` and `Oval`. The compiler rejects all three, because the type of the reference is to a `Shape`, and class `Shape` does not have those methods.

By this it should be clear that the compiler *requires* that access to a member can be verified at compile-time. The member must exist in the type of reference used. It does not matter what the reference may actually be at runtime, because the compiler will only allow what it can guarantee from static type checking.

Using the same example, it is also possible to point out that similar type restrictions apply to the variable assignments. Our assignments of the variables `c`, `o`, and `s` were all valid, because the type of the variable was of equal or higher type than the assigned value. The relative level of two types refers to their relative position in the inheritance chain. Class `Circle` is lower than `Oval` because it is a subtype to `Oval`. Class `Shape` is higher than `Oval` because it is a supertype to `Oval`.

The type level relationship between two classes only apply if they are on the same inheritance chain. If we examine the classes `Triangle` and `Rectangle`, they are on different branches of the type hierarchy tree. This becomes obvious by looking at figure 17.2. A variable declared as (reference to) a `Triangle` cannot be assigned a `Rectangle`, nor an `Oval`, `Circle`, `Shape` or anything else. It can *only* be assigned a `Triangle`, or `null`. This leads to the following failures:

```
Circle c = new Circle(5.0); // Valid, same type

Oval  o = c; // Valid (1) sub- to supertype
Shape s = c; // Valid (2) sub- to supertype

c = o;       // Invalid (3) super- to subtype
o = s;       // Invalid (4) super- to subtype
c = s;       // Invalid (5) super- to subtype
```

Assignments (1) and (2) are valid, because the type of the assigned value is a subtype of the variable type. Assignments (3), (4), and (5), on the other hand, are not valid because the assigned value is a supertype of the variable. The type of the reference cannot be automatically specialized in the assignment, or more accurately *the narrowing type conversion* is not allowed.

If we look closer at (5), for example, we see that variable `s` is of type `Shape`. It can thus refer to instances of all types below it. But variable `c` can only refer to instances of type `Circle`. As the compiler can not guarantee that this will be the case, the assignment is not allowed.

It is important to remember that these assignment rules also apply when passing references as arguments to a constructor or method. The assignment is to the local parameter variable, and it is the type of the parameter variable that determine what can be assigned to it. For example:

```
public void inspect (Shape s) { ... }
```

We can give any `Shape` or subtype of `Shape` as argument to method `inspect`, because the assignment to the parameter variable `s` is valid in those cases. At the same time, inside method `inspect`, the instance in parameter `s` is a `Shape` and cannot directly be treated as a more specialized type. What we *can* do, in this particular example, is to call `Shape.area` because that is defined for `Shape`.

Similarly, assume a method like this:

```
public void scrutinize (Circle c) { ... }
```

This method can only be called with a `Circle` as argument. Anything else, and the assignment to the parameter variable will be invalid and rejected by the compiler.

17.2.1 Vertical type derivation

The restrictions imposed by the compiler's strict type checking mostly work to our advantage, in that it prevents bugs and problems. Sometimes, however, we would like to be a little more flexible. For example, consider the following code:

```
Circle c = new Circle(5.0);

Shape s = c;
c = ... // something else for a while
c = s;  // Invalid, but we know it is a Circle
```

In this code example, the first value of `c` was saved in variable `s`, and later we want it back in `c`. But since the compiler does not allow this, it cannot be done that way. We are, however, completely sure that the reference in `s` refers to a `Circle`, so we add a typecast to reassure the compiler that we know what we are doing:

```
c = (Circle) s; // Valid because of the explicit
                // cast to a subtype
```

The code now compiles and the program runs. At runtime, there *will be* a check of the assignment, and if the assigned value can not be viewed as a `Circle`, a `ClassCastException` is thrown.

In the above example, we knew that the reference in `s` was actually a `Circle`. This is a actually a rare case, and it is not good to rely on beliefs, especially as the program evolves and becomes more complicated. What we want to do, is to be able to check the type at runtime and take action accordingly.

In Java the built-in boolean operator `instanceof` allows us to ask if a reference is of a certain type. We can use this to make sure that a type conversion towards a more special type is safe to perform, rather than risking the `ClassCastException`. As an example, we assume that method `inspect(Shape s)` would like to be able to adapt to the argument in this way:

```
public void inspect (Shape s) {

  if (s instanceof Oval) { // is s an Oval?
    if (s instanceof Circle) { // yes, it is a Circle?
      Circle c = (Circle) s;   // yes, safe to typecast
      ... // do Circle things
    }
    else {
      Oval o = (Oval) s; // it must be an Oval then
      ... // do Oval things
  }
  else ... // continue checking for other
           // subtypes of Shape if required

}
```

It should be remembered that the `instanceof` operator is not the best solution to deal with the problem. The reasons are that it is a bit costly to apply as it must pull out the class information from the object instance at runtime and compare it. It is also more convoluted to maintain the selection code when more subclasses are introduced. If, for example, shape `Polygon` is introduced, all places in the code that performs runtime type checking may have to be modified. Such dependencies easily leads to bugs.

Two good alternatives exist to the `instanceof` operator. What we want to achieve is to select code to execute based on the type of an object.

The first alternative is to use `overloading`, i.e. different versions of the same method, where the *parameter type* is different:

```
public void inspect (Circle c) {...} // (1)

public void inspect (Oval o) {...} // (2)

// public void inspect (Square sq) {...} // (3)

public void inspect (Rectangle r) {...} // (4)
public void inspect (Triangle t) {...} // (5)

public void inspect (Shape sh) {...} // (6)
```

Using overloading clearly identifies and separates the different cases. The compiler selects the method to call at compile time, *based on the type of the argument.*

Read that last sentence again. The method that will be called is decided already when the program is compiled. This is known as *static binding.* The compiler examines the type of the argument expression and then binds the call to the most specific match among the overloaded variants. Thus a call like:

```
inspect(new Circle(...));
```

will make a call to version (1), because the `Circle` matches best. The following code:

```
Oval o = new Oval(...);
inspect(o);
```

will call method (2) because `Oval` is the best match.

Note that in (3), the overloaded variant for `Square` is commented out. There is no special case for `Square`. Thus, the code:

```
inspect(new Square(...));
```

compiles into a call to (4), because the `Rectangle` is the most specific match, based on type.

Similarly, on the general level:

```
Shape sh = new Circle(...);
inspect(sh);

sh = new Rectangle(...);
inspect(sh);
```

The above code compiles as calls to (6), the `Shape` variant, because that is the type of the argument (variable `sh` in this case). It does not matter that the object instances assigned to `sh` are of a different type. See A.4 for the full source code.

Overloading is a helpful tool, but it does not completely solve all cases. Different code for different object types can mean that we want to *treat* the objects differently, depending on their type. The thinking is that we approach the objects from outside of them, and we want to do something with them. In such cases, overloading is a good choice.

On the other hand, overloading requires us to write a separate method for each case. This is better than having a complicated weave of nested `if`-statements and `instanceof` operators. But if we do this all over the program, we still have numerous places to edit if we add, remove, or modify our class hierarchy.

In object-oriented programming, we should try to avoid thinking in terms of *what we do* with the objects. Instead, we want that different object types *behave* differently when responding to a certain action. We should be thinking *inside* the object, *what it is doing* for us in its own special, and characteristic way.

In the `Shape` class hierarchy there are two good examples of this. The `area` method and the `toString` method. They are also examples of the most powerful way of specifying type-dependent selection of code — *polymorphism*.

17.3 Polymorphism and dynamic binding

One translation of the word *polymorphism* could be *many forms*. In object-oriented programming it represents the idea that one and the same type can exhibit different forms and behaviours. A simple, but still completely accurate example of this is found in section 17.1.4, where we gave the various subclasses of `Shape` their own way of calculating their area.

Class `Shape` has the method `area`, and because it is overridden by the subclasses, we can call method `area` on any `Shape`. We do not need to know if it is actually a `Triangle` or a `Circle` or some other subclass of `Shape`. Since `Shape.area` has been overridden, the call will automatically be directed to the deepest version of the method. In the case of a `Circle`, for example, the code that executes is that of `Oval.area`.

Here is a further illustration of how polymorphism can make the code more general:

```
Shape [] shapes = new Shape[3];
Point pt = Point.getRandomPoint(100, 100);

shapes[0] = new Circle(pt, 5.0);
shapes[1] = new Oval(pt, 2.5, 7,0);
shapes[2] = new Rectangle(pt, 1, 2);
```

The element type of array `shapes` is `Shape`. Thus we are allowed to assign to each element a (reference to a) `Shape`, or one of its subclasses. This is then what we do. Each element is a different subclass in this example.

We can now print the area of all the shapes in a simple loop. Note that we do not need to worry about what type each object instance actually is. They are all treated as instances of `Shape`:

```
for (Shape sh : shapes)
  System.out.println(sh.area());
```

When the code `sh.area()` is compiled, the compiler cannot know exactly what type the reference in `sh` will have. It may be a `Square` or a `Circle`, or something else. Since it cannot know in advance, static binding of the call to a specific method is not possible. What the compiler must do instead, is to generate code that will determine the method to call at runtime. This is called *dynamic binding*, and is usually implemented through a lookup table.

The full ramifications and power of polymorphism may need some getting used to. The most important point, however, is that it offers an elegant way to design objects that can have individual behaviours at the subclass level, while still being treated uniformly at the superclass level.

The second most important point is a corollary of the first: we can have our type-dependent code selected automatically at runtime, and it is located in the definition (source code) of the type for which it is written.

The third important point now follows from the second: when we need to add a new subtype, we just write the new type, extending from the common superclass or one of its pre-existing subtypes. We put its special code inside it, overriding the appropriate methods. The need to revisit and update external routines implemented with `instanceof` predicates or overloaded methods, can be minimized.

Now, there is of course a cost associated with polymorphism as well. One realization is that class hierarchies are monodirectional towards the superclass, i.e. every class that we write has exactly one superclass. Any type-dependent code selection must fit into the class hierarchy we have selected, or polymorphism will not be applicable. If there are several different relations that need to be modelled and programmed for, it may be the case that our class hierarchy can only support one such relation through polymorphism.

Object-oriented modelling of real-world concepts is often complicated and requires careful thought. It is difficult to give general guidelines, but here are some things that could be considered:

- A class can implement any number of interfaces. Interfaces are also classes, and polymorphism works through interface implementations as well.

- If definition classes become large, see if they can be split up into smaller classes and class hierarchies that have little or no dependencies between each other. Aggregation and access can then be provided by a container class.

17.3.1 Requiring class extension

In some cases it is desirable to be able to specify that a class is expected to be subclassed. For example, class `Shape` represents an abstraction that we never intend to make an instance of. We can present this formally in our declaration of class `Shape` by making it `abstract`:

```
public abstract class Shape {

  public abstract double area();
}
```

The abstract property prevents instantiation. Any class that extends `Shape` must override the abstract method `area` with a real method, or the subclass will be inherit the abstract property as well.

An abstract class can contain a mix of concrete (fully implemented) and abstract methods. However, a single abstract method is a sufficient condition to make the whole class abstract.

The *purpose* of the abstract declaration is to signal to other programmers that the abstract class is not meant to be instantiated. It is a partial implementation, and the remaining parts must be filled in by subclasses. By having the restriction built into the programming language, the compiler helps enforcing it.

17.3.2 Preventing class extension

A class that is declared `final` cannot be subclassed. The compiler will not accept an attempt to extend a final class. For example, in the design of our `Shape` hierarchy we can specify that for the classes `Circle` and `Square` we have decided that there are no meaningful specializations:

```
public final class Circle extends Oval {
  ...
}

public final class Square extends Rectangle {
  ...
}
```

Giving a class the `final` property is an indication to other programmers that further subclassing is prohibited, and that they should seek other solutions to what it is they are trying to do.

17.4 Source code for the `Shape` hierarchy

Source codes for the `Shape` hierarchy, test, and demo programs can be found in appendix A. The code fragment examples used in this chapter may differ slightly from the full working code in the appendix.

Chapter 18

Class hierarchies

This chapter revisits several concepts that were illustrated in previous chapters. It is an attempt to give a coherent picture of the tools available to extract the most from an object-oriented design.

18.1 Class hierarchy

The `Shape` class hierarchy is shown in figure 17.2. It has a width of three and maximum depth of three. The size is managable, and as illustrated by class `Triangle` (see A.2.6) not trivial despite its apparent simplicity.

The Standard Library is rich with similar class hierarchies, both small and large. For examples, look at classes `java.lang.Number` and `java.lang.Throw-able` and their descendants.

The obvious advantage with having code organized in a class hierarchy, is that common behaviours and data can be placed in superclasses, while unique behaviours and data are pushed down into subclasses. However, this is not a dogma to be pursued for its own sake; there should be some actual code sharing, or other tangible benefit won from the design. If no such advantage can be found, or the design becomes too convoluted just to fit inside a single common framework, then it is probably better to rethink things.

18.1.1 Simple inheritance

A class always has exactly one superclass. If no `extend` keyword is used in the class declaration to explicitly point out a superclass, the class will inherit from the special class `Object`.

```
       Object
          |
      +---+---+
      |       |
      A       X
      |
   +--+--+
   |     |
   B     E
   |
+--+--+
|     |
C     D
```

Figure 18.1: A deep class hierarchy

The rules for inheritance lead to *simple inheritance*. Each class has a single superclass, and zero or more subclasses.

18.1.2 A common superclass

Since each class has a single explicit superclass, or defaults to `Object`, all classes share a common superclass somewhere above them in the class hierarchy. For many classes, the shared superclass is class `Object`.

It is good to be aware how the location of a common superclass controls what reference types can be used in relation to objects of interest.

Examine figure 18.1 and the classes `C` and `D`. Their closest common superclass is `B`, so a reference to `B` can be used to refer to instances of `B`, `C`, and `D`.

The lowest common superclass of, say, `D` and `E` is class `A`. With a reference of type `A` we can refer to instances of both `D` and `E`, but at the same time also to `B` and `C`, as these also are subclasses of `A`. With the given class hierarchy, there is no common type that allows us to refer only to `D` or `E`, all they have in common is what `A` defines for all of its subclasses.

Class `X` has nothing in common with class `A` and its subclasses. The only common superclass is `Object`.

18.1.3 The inheritance chain

Class `C` and `D` are deepest in figure 18.1. A reference to `C` can be used to access members defined in `C`, `B`, and `A`, because `C` extends `B`, which in turn extends `A`.

```
       Object
          |
      +---+---+
      |       |
      A       X
      |
   +--+--+
   |     |
   B     |
   |     |
+--+--+  |
|     |  |
C     D  E
```

Figure 18.2: A deep class hierarchy

Similarly, a reference to `D` can access members in `D`, `B`, and `A`.

A reference to `E` can access members in `E` and `A`.

The way we make images of a class hierarchy is not unimportant. Of course, they should convey the class structure accurately, but unless drawn with care they may impart false expectations in our minds. Look at figure 18.1 again. From the graph it would appear that classes `B` and `E` are somehow equal, or on the same level of abstraction, because they appear on the same height in the graph.

But what if it is the case, that class `C`, `D`, and `E` actually are on the same level of abstraction? What if class `B` is just there to capture something that `C` and `D` happen to have in common? Would it not be better to draw the graph as illustrated by figure 18.2, with similar abstractions on the same vertical level?

18.1.4 Java and class Object

Class `Object` is special in the Java programming language. All classes ever defined, in any Java program, are subclasses of `Object`. Class `Object` provides a root for the global class hierarchy, and gives the reference type for the superclass that is common to all classes. A reference of type `Object` can refer to any instance.

Class `Object` also performs an important service by giving every class a few common methods.

The `equals` method guarantees that two objects always can be compared, even if the default definition of equality is quite weak: an `Object` is only equal to another `Object` if they are in fact *the same* object instance. Classes that want a different definition of equality, like for example class `String`, must override method `equals`.

The `toString` method provided by `Object` does not return a pretty string description (which is why it should always be overridden), but it guarantees that all objects can be printed.

As demonstrated by the classes in the `Shape` hierarchy (see appendix A) it is sometimes convenient to be able to print the name of a class without having to rely on a string constant. Such a string constant can easily be misspelled, or forgotten if the class is renamed. Better then to extract the name information from the object directly, at runtime. Class `Object` provides a way to do this, with the method `getClass`.

Method `getClass` returns (a reference to) an object instance of type `Class`. The `Class` instance contains information about the class *in which the method was called.* The information can be retrieved by calling methods in the `Class` instance. One of those methods is `getName`, and it returns a string with the name of the class.

Thus, with reference to figure 18.2, if an instance of `D` executes:

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

then the printout will be:

```
D
```

It does not matter if the call is made by code located in `D`, or if the call is made by code inherited from superclasses. For example, if we place the following method in class `A`:

```
public String myClassName() {
  return getClass().getName();
}
```

and we then code in `D`:

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

it will still print:

```
D
```

If class `C` executes the same code the printout is:

```
C
```

This happens because the name belongs to the executing class (C or D, in this example), not the location of the call to method `getClass`.

For the remainder of the `Object` methods, please consult the official Java documentation.

18.1.5 Inheriting from the Standard Library

When creating subclasses from classes in the Standard Library, it is usually a good idea to study the class hierarchy in which the intended superclass is located. This can shed additional light on how it is supposed to be used, and if subclassing from it is a viable approach.

In general, subclassing from the Standard Library happens in these situations:

- The library class is designed for subclassing to simplify programming
- The return value of a method is an abstract type, because the library does not specify any particular implementation. This is common with more complicated objects like parsers or advanced data structures.

Classes in the Standard Library that are intended for subclassing are usually documented as such. One example is found in the package `java.awt.event` and class `MouseAdapter`. This is an abstract class that implements the interfaces `MouseListener`, `MouseMotionListener`, `MouseWheelListener`, and `EventListener`. These interfaces together specify quite many methods but the `MouseAdapter` class has implementations for each. This means for an application, that rather than having to implement each method in each relevant interface (most of which will be empty methods to satisfy the compiler), it can subclass `MouseAdapter` and just override the methods it is interested in.

Abstract classes are meant to be subclassed, but not all abstract classes are meant to be subclassed by application programs. Many abstract classes are abstract because it fits with the design of the Standard Library itself.

Classes that are not intended for subclassing are marked `final`, and this can also be learnt from the documentation.

Before subclassing from the Standard Library, one should consider if the desired effect can not be achieved by simply creating an instance of the library class. For example, the designer may consider this:

```
public class Foo extends java.util.Timer {
  ...
}
```

One should always first consider why doing it like this does not work:

```
public class Foo {
  java.util.Timer myTimer = new Timer();
  ...
}
```

A good argument *for* subclassing is that the new subclass will be able to participate in the program as a customized instance of its superclass. This is great for graphical components or other spots in the Standard Library designed with this in mind. If that is all the new class does, then fine. If it is supposed to do more, and in particular do things that are not directly related to the role of the superclass, then it gets trickier.

An argument *against* subclassing the Standard Library as a solution is that the superclass occupies the single superclass position. The subclass can not be part of another class hierarchy.

Every class designed for an application is of course important. It has tasks to perform, data to protect, and so on. It has a *role* to fill in the overall design. If we try to put multiple roles into a class, we may find that it is not capable to support all of them equally well. This may be because of the way we would like to use polymorphism, or because how we want the class to interact with other classes.

When such conflicts appear, it may be possible to divide the class into different classes, so that each can be devoted to a single role or task. However, this may now lead to new conflicts, if the two classes need to share data. One role may be to ensure that data is represented and manipulated with accuracy and consistency, while another role may be to present or display the same data through a GUI. Both roles require access to the data, and we want the benefits of encapsulation. There are at least three solutions to this:

- A single class performs both roles through the use of inheritance, public methods, and perhaps also implementations of interface classes.
- The class is split into two or more classes, one for each behaviour, or role. An additional class is created to govern shared data, and the behavioural classes are given references to that object. This has the benefit of clarity and a clear division of labour, but also a performance penalty which can be significant, in that all access to shared data must now be done through method calls.
- A single class performs the governing of data and provides the public interface. Nested classes in it are created to support the behavioural roles. Nested classes has the advantage that we are back to a single class which can enforce encapsulation, but at the same time the inner classes are able to *extend from any class*, and thereby be compatible components of such environments. There remains a very small performance penalty when the nested class is accessing instance variables in the surrounding class, but it is neglegible when compared to the overhead of a method call.

```
+--class Sensor---------------------------------------------+
|                                                           |
| double curCel, curFar                                     |
| byte [] history                                           |
|                                                           |
| +--class Sampler--+ +--class Log----+ +--class Display--+ |
| |                 | |               | |   extends JPanel| |
| | read hardware   | | read vars     | |                 | |
| | filter          | | write to log- | | read vars       | |
| | store in vars   | | file          | | paintComponent  | |
| |                 | |               | |                 | |
| +-----------------+ +---------------+ +-----------------+ |
|                                                           |
+-----------------------------------------------------------+
```

Figure 18.3: The structure of class `Sensor`

An example helps with the illustration of these design decisions. Assume we have a class `Sensor` which represents a hardware sensor of some kind. Let us assume it is something with a single data channel, like a thermometer.

One role, or behaviour, for class `Sensor` is to communicate with the hardware, and every second read the raw sensor value, then filter and transform it, and finally store it in variables and data structures that represent the current value and recent history.

A second role for class `Sensor` is to send formatted lines of text to a logfile every other minute.

A third role for class `Sensor` is to provide a graphical presentation of the current state of the sensor.

We can design a single class `Sensor` which is able to manage all three roles. An outline is shown in figure 18.3.

Class `Sensor` has three nested classes. Class `Sampler` contains the code to read raw bytes from the hardware, low-pass filter it, and store it in the byte array `history`. It also performs a transformation from byte value to double-typed Celcius and Fahrenheit and stores the most recent update in the corresponding variables, located in the outer class.

Class `Log` reads data from the `history` array, formats a string containing a timestamp, and then writes that string to a logfile.

Class `Display` subclasses `javax.swing.JPanel`, and is thus a GUI component. When the GUI system calls its version of method `paintComponent`, the code there reads the variables in the outer `Sensor` class and draws a graphical representation based on their values.

Thus, by using nested classes we are able to support encapsulation of both data and behaviours, while at the same time provide separation and modularization of different behaviours. This, is a good thing.

So, in summary:

- If the custom subclass only supports the Standard Library superclass, albeit with an application-specific tweak, then subclassing is probably viable.
- If the custom subclass is also supposed to do things that are not related to the role of the Standard Library superclass, then it is probably better to separate things into different classes.
- Separation of duties and encapsulation of data can be maintained in a single class by using nested classes. A nested class can extend any extendable class.

18.2 Managing a class hierarchy

18.2.1 Abstract superclasses

Abstract classes are designed to be extended. It is not possible to create an instance of an abstract class, because it contains one or more methods that are abstract and not fully implemented.

Inheriting from an abstract class

When an abstract class is extended, the software designer must decide if the subclass is to be abstract too. The subclass can be designed to override none, some, or all of the abstract methods in superclass. Only when all abstract methods have been overridden, can a subclass become concrete, and instances created from it. If a single abstract method is not overridden, the subclass will be abstract too.

The Java programming language even allows the the abstract subclass to introduce additional abstract methods. For example:

```
public abstract class C {

  public abstract void foo();

}

public abstract class D extends C {
  public abstract void bar();

}

public class E extends D {

  // class E must now override both D.bar
  // and C.foo in order to be concrete.

}
```

18.2.2 Managing objects with the same superclass

Objects that share a superclass (all do), can be treated uniformly by referring to the common superclass. However, only members present in the common superclass can be accessed.

Array

Making an array (or collection) where the element type is that of the common superclass, makes it possible to treat a group of objects uniformly, even though they may have distinguished behaviours in their specializations. For an example, see the `Shape` class hierarchy in appendix A, and examine class `TestShape.java`.

The `instanceof` operator

The `instanceof` operator is a predicate that returns `true` if (a reference to) an object is of the proposed type:

```
  Shape sh = ...

  if (sh instanceof Rectangle)
```

The condition in the `if` statement is true if the reference in variable `sh` refers to a `Rectangle` or a `Square`.

18.3 More about class Object

18.3.1 The class Class

The method `Object.getClass` returns an object of type `java.lang.Class`. This object describes the object which is calling the method.

Methods in class `Class` allows the program to perform something which is called *reflection*. An object can look itself in the mirror, or ask for the description of another object. Available fields and methods can be located and called.

The ability to perform reflection might appear redundant, but remember that the Java language allows a program to receive objects from a file or a network stream. Such an object may not be known to the program in advance. Class `Class` and reflection offers a way for the running program to examine an unknown object at runtime, and perhaps utilize it.

18.3.2 References to `Object`

A reference of type `Object` is universal, because every class in Java is an `Object`. A variable of type `Object` can be assigned any reference.

Such references are rarely useful, though, except in special cases such when receiving a truly unknown object, and even then the programmer usually has some idea of what to look for. Before Java 1.5 references to `Object` were used frequently, but the introduction of *generic types* (see 19) provided a clever improvement.

18.4 Type independent programming

Type independent programming is what polymorphism enables. It is the ability to work at a higher level of abstraction, using the operations provided by the common type (the common superclass). It does not matter that the common type may be an abstract class, or an interface class. As was demonstrated with class `Shape` (see chapter 17 and appendix A), the actual instances perform and implement the operations in a type-dependent way.

The type indepedent code can be written at the common level of abstraction, and is uncluttered by special cases and multiple of considerations. Adding new subclasses to the class hierarchy *does not require any change to the main code*, if done within the envelope of the common model. Similarly, any maintenance to existing code that remains shielded by encapsulation, can be done without having to change anything else. Internal algorithms and data structures can be

```
                    Number
                      |
  +-------+-------+---+----+--------+------+
  |       |       |   |    |        |      |
Byte   Double   Float | Integer   Long   Short
                      |
        +-------------+---------+----------+
        |             |         |          |
   AtomicInteger AtomicLong BigDecimal BigInteger
```

Figure 18.4: The `java.lang.Number` class hierarchy

Modifier and Type	Method and Description
`byte`	`byteValue()` Returns the value of the number as a byte
`abstract double`	`doubleValue()` Returns the value of the number as a double
`abstract float`	`floatValue()` Returns the value of the number as a float
`abstract int`	`intValue()` Returns the value of the number as an int
`abstract long`	`longValue()` Returns the value of the number as a long
`short`	`shortValue()` Returns the value of the number as a short

Table 18.1: Methods in class `Number`

tweaked, or even replaced, as long the class continues to exhibit a consistent behaviour.

18.4.1 The `Number` class hierarchy

Figure 18.4 shows the class hierarchy below class `java.lang.Number`, an abstract class. Please note that the hierarchy only has two levels; the figure is drawn to accomodate that the subclass level has ten members.

Among the subclasses to `Number` we find the wrapper classes to the primitive number datatypes, `Byte`, `Double`, `Float`, `Integer`, `Long`, and `Short`. Class `Character` is not included, because it represents characters, not numbers, even though characters happen to be encoded by positive numbers. Class `Boolean` represents logical values, and is not a `Number` either.

The numbers `AtomicInteger` and `AtomicLong` are two classes out of a larger set that support concurrent programming. `BigDecimal` and `BigInteger` are classes that support numbers with an arbitrary number of digits in them.

Table 18.1 shows the methods available in class `Number`. The value of the `Number` can be retrieved as one of six primitive datatypes, with the reservation that it may have been rounded, or truncated. For example, it seems obvious that taking the `byteValue()` from a `Long` may result in misrepresentation as we go from 64 to 8 bits of storage.

The point here is not that the programming language does not protect us from making a mess of our data if we want to. The point is that the many ways of representing a number does not prevent us from treating them *uniformly*.

Chapter 19

Generic types and interfaces

19.1 Type independent data structures

It is quite common that one wants to collect references to objects in a data structure of some kind. It can be a list, a set, a tree, a hashtable or something else. Such data structures are provided by the Standard Library in the package `java.util`. Together they are called *collections*, because they are used to collect (references to) object instances.

The collection classes has to be general. They are provided to all programmers through the Standard Library, so preferably they should be able to hold any existing and future object reference. Prior to Java 1.5 this was solved by letting the collected reference be of type `Object`. This works because every Java class, both existing and future, is a subclass of `Object`. If you used a `LinkedList`, you had a list of `Object`. If you used a `Set`, it was a set of `Object`.

Having `Object` as the common superclass was quite alright when adding object instances *to* the data structure, because adding means assignment, and any reference can be assigned to a variable of type `Object`. However, when getting them out again, `from` the data structure, the reference that came out was of course a reference to `Object`. So, if you had put, for example, strings in a list, you had to explicitly cast them back to `String` each time you retrieved them. It looked like this:

```
LinkedList lst = new LinkedList();
...
lst.add("Foo");
lst.add("Bar");
lst.add("Bletch");
...
```

```
String s0 = (String) lst.get(0);
String s1 = (String) lst.get(1);
String s2 = (String) lst.get(2);
```

All that explicit typecasting was necessary in order to satisfy the compiler, but if was often felt redundant because the programmer *knew* it was nothing but strings in that list. It was also somewhat unsafe, because the compiler could not help with typechecking below the level of `Object`, and since `Object` is the top level, that meant no type checking help at all.

The issue was further complicated if the programmer, by design or mistake, loaded the data structure with objects of different types and did not cast to the most common superclass when retrieving them. A situation where the programmer has to remember what type went where, is ripe for bugs. The only mechanism to prevent against disaster was heavy use of the `instanceof` operator, which further complicated the code and obscured the logic of the main purpose of the program.

The developers of the Java language were faced with the problem of doing something about the collections that would create cleaner and safer source code, *without breaking backwards compatibility.* It was imperative that old bytecode would continue to run on the JVMs.

So they decided to work on the compiler, and their solution was the introduction of *generic types.*

19.1.1 Generic types

A generic type can, and in most instances should, be thought of as a *type parameter.* When it is used, the programmer tells the compiler: *"this class and this variable, is going to have a reference type, but I don't know yet which one it will be."* In Java code, it can look like this:

```
public class Foo<T> { // type parameter declaration

  public T bar; // variable declaration
}
```

In this example we are telling the compiler that class `Foo` will be using some type `T`, but we do not know which type that will be. We do this by giving the type parameter in angle brackets after the class name:

```
class Foo<T>
```

The identifier `T` is the name we choose for the *type variable*, so the compiler can see where in the class it is being used. By convention type variables are single, uppercase letters, but we are free to choose any valid variable name. Since we have not decided upon which type it will actually be, the type is *generic.*

Inside class Foo we then use our type parameter, by declaring that variable bar will be a reference to type T, whatever it happens to be.

"Ok then," says the compiler and compiles class Foo, generating code that is conceptually equivalent to:

```
public class Foo { // Foo is declared <T>

  public Object bar; // bar is declared T
}
```

Later on, the compiler encounters some other code that wants to use class Foo together with an actual type:

```
Foo<String> fs; // Variable fs is declared
```

Here, the code declares that variable fs is a reference to class Foo, but specifically and only when the type parameter in Foo set to String. So the compiler sees this and makes a note that variable fs can only be assigned references to instances of Foo where the generic type T has been qualified to String.

The compiler keeps compiling, and now it comes to the statement:

```
fs = new Foo<String>();
```

Here a new instance of class Foo is created, and this instance has the generic type T qualified to String. Since this is exactly the type with which variable fs was declared, the compiler is happy and allows the assignment. Nothing happens to class Foo itself, it is already compiled and in bytecode.

The next line of code then does:

```
fs.bar = "hello world";
```

Now the compiler looks at variable fs and sees it is declared to hold references to instances of class Foo with T=String. So far so good. For the access to bar the compiler has to dig into its notes about member Foo.bar. It sees that it is declared public and is of type T. Since variable fs has T=String, and the reference being assigned is of type String, the compiler allows the assignment.

Then the compiler comes to the statement:

```
String s = fs.bar;
```

Once again the compiler examines variable fs, finds that fs refers to class Foo, with T=String, and finds that Foo.bar is of type T.

"This means," the compiler thinks, *"that whatever* bar *is referring to, it* must *be a string, or I would never have allowed the assignment in the first place."* As a result, the compiler is satisfied and can generate code similar to:

```
String s = (String) fs.bar;
```

The compiler provides the typecast from `Object` to `String`, because it has annotated the reference variable `fs` with `T=String`, and it will only allow assignments that are compatible with that. If every assignment is compatible with the parameterized type, then it must be safe to cast an `Object` reference qualified by that parameterized type, back to the parameter type.

19.1.2 Using collections

Many of the classes in the `java.util` package uses generic types. For example, we can look att the documentation of class `LinkedList`:

```
public class LinkedList<E>
```

It has a type parameter called `E`. The use of the letter `E` is a reminder to the human reader that the generic type refers to the type of the list elements.

When we want to use a `LinkedList` we must decide on a type for the type parameter. This of course depends on what we want to store in the list. As an example, let us create a list that stores strings:

```
LinkedList<String> lst = new LinkedList<String>();
```

Here we say that variable `lst` is a reference to class `LinkedList`, with the type parameter set to `String`. Variable `lst` is then initialized to a new `LinkedList`, with the same qualification of the generic type. We can now add and extract elements from the list without the need for explicit typecasting:

```
lst.add("foo");
lst.add("bar");
lst.add("bletch");

String s0 = lst.get(0);
String s1 = lst.get(1);
String s2 = lst.get(2);
```

In addition to the cleaner code, the compiler will refuse assignments that are not compatible with the type selected for variable `lst`:

```
lst.add(new Double(5.0)); // error, Double is not String
...
Boolean b = lst.get(2); // error, b is not String
```

It is possible to have more than one type parameter in a class declaration. Take for example class `java.util.HashMap`. A `HashMap` is a key-value storage. An object reference is used as a key to store another object reference which is the value. Later, the value can be retrieved by presenting the key to the `HashMap`.

Class `HashMap` is declared like this:

```
public class HashMap<K,V>
```

This is no different from what has been shown above, except that instead of a single type parameter, there are now two of them. The first parameter is named `K` to remind us it is the type of the key, and the second parameter is named `V` to remind us that it is the type of the value.

Let us assume that we want a lookup table that maps room names to temperatures. We create a `HashMap` that stores double precision numbers keyed by strings. It looks like this:

```
HashMap<String,Double> roomTemps =
                       new HashMap<String,Double>();
...
roomTemps.put("bedroom", 17.8);
roomTemps.put("kitchen", 19.2);
roomTemps.put("foyer",   16.9);
```

The declaration of method `HashTab.put` is:

```
public V put (K key, V value)
```

So method `put` must be called with the first argument of the type we have decided for the key, and the second argument of the type we have chosen for the value. This will install the entry in the table. If there was a value already stored under the given key, that value is returned. Each key can only store a single value.

In our case, we have chosen `Double` as the type for the value. When the compiler sees our `double` type temperatures in the source code, it will autobox them by creating instance of `Double` for us.

Then later we want to extract the temperature again. This is done with method `HashMap.get`:

```
public V get(Object key)
```

Here the parameter type of the key is `Object`. Surely it should be `K`?

It turns out that since the `HashMap` allows *any* object to be used as a key, it must use some property of the key that can be found in *all* objects. The property in question is the value returned by method `int Object.hashCode`. This method returns a number that is used internally in the `HashMap` to locate the place where the key's value is stored. For this reason, method `HashMap.get` does not need to typecheck the key argument. All objects can acts as keys. The checking was already done when the key was `put` in the table.

So, we can call `get` with our key and retrieve the value:

```
double t = roomTemps.get("kitchen");
```

Since the return type of `get` is `V`, and `roomTemps` was declared with `V=Double`, the returned data type will be a `Double`. The compiler knows how to unbox a `Double` into a `double`, and assign that to variable `t`[1].

19.1.3 Type erasure

All of the generic type processing is done during compilation. A little bit of information is stored with the compiled classes so that future compilation passes can see the generic type declarations. At runtime, however, the types have been *erased*, and assignments and typecasts are executed as approved and supported by the compiler.

Generic types make the language stronger because it is somewhat more restricted. The programmer sometimes has two write longer declarations, specifying the data types to use, but is relieved of tedious and potentially dangerous typecasts.

19.1.4 Bounded type parameters

The full story about generic types has more to it. In particular, there are expressions that can be used to limit a type parameter variable, so that only certain types are permitted. This can be done by specifying bounds on the parameter types. For example, assume that we would like to have a class `Foo` (see 19.1.1) parameterized on a `Shape` or one of its subclasses. It can be done like this:

```
public class Foo<T extends Shape> {

  public T bar;
}
```

The *type expression* `<T extends Shape>` states that parameter variable `T` is only allowed to be used with type `Shape`, or a type that extends `Shape`.

Wildcards

In some contexts you may want to relax the type specification of a variable. Assume you have the following variable declaration:

```
LinkedList<Shape> shapeList;
```

The type of variable `shapeList` is `LinkedList<Shape>`. It can only be assigned values of that type. If you happen to have another list, like:

[1]If no value is stored under the key “kitchen”, a `NullPointerException` will be thrown, because there is nothing to unbox from.

```
      +----------OK----------+
      |                      |
 LinkedList<Shape>      LinkedList<Oval>
              |                      |
              +--------- X ----------+

                        ERROR
```

Figure 19.1: Type parameter mismatch

```
      +----------------OK--------------+
      |                                |
 LinkedList<? extends Shape>      LinkedList<Oval>
            |                                  |
            +----------------OK---------------+
```

Figure 19.2: Type parameter match

```
LinkedList<Oval> ovalList;
```

then the attempted assignment

```
shapeList = ovalList;
```

will be rejected by the compiler, because the variables are of different types. It does not matter that `Oval` is a subtype `Shape`. The types of the lists are different, because the element types are *fixed.* Figure 19.1 illustrates how the compiler fails to match up the element types of the lists.

The solution to the problem is to relax the type, using a `type wildcard`:

```
LinkedList<? extends Shape> slist;
```

The type of variable `slist` is now declared to be `LinkedList` with the element type to be `Shape` or a type that extends `Shape`. Since there is no need for a type variable, like `T` or `E`, the question mark `?` represents the bounded type wildcard in the type expression. Figure 19.2 shows how class `Oval` matches the wildcard in the type expression.

Now the assignments are allowed, for example:

```
slist = ovalList;
...
slist = shapeList;
```

The type expression `<?  extends Shape>` puts an *upper bound* on the type. Only types that are `Shape` or extend `Shape` can match. Similarly, it is possible to specify a *lower bound* on the type. For example, assume that we want to allow lists with element types `Circle`, `Oval`, `Shape` or `Object`[2]. This be done like so:

```
LinkedList<? super Circle> roundList;
```

The following assignments to variable `roundList` are then valid:

```
roundList = new LinkedList<Circle>(); // OK
roundList = new LinkedList<Oval>();   // OK
roundList = new LinkedList<Shape>();  // OK
roundList = new LinkedList<Object>(); // OK
```

These other possible lists do not match the type expression:

```
roundList = new LinkedList<Square>();    // error
roundList = new LinkedList<Rectangle>(); // error
roundList = new LinkedList<Triangle>();  // error
```

Generic methods

A class definition can be written with one or more generic types and type variables that can be used all over the class. In a similar way it is possible to define generic types and type variables that only apply to a method. The type parameters are the same as for a class, but they must be inserted before the method's return type. For example:

```
public class Tools {
  public static <T> void process (Foo<T> fobj)
  {
    T temp = fobj.bar;
    ...
  }
}
```

Calling a generic method using the full syntax then becomes:

```
Foo<Oval> f = new Foo<Oval>();

Tools.<Oval>process(f);
```

However, the compiler is usually clever enough to apply *type inference* and figure out what it needs to know from the calling context. In most cases it is therefore possible to call the method in the usual way:

```
Foo<Oval> f = new Foo<Oval>();

Tools.process(f);
```

[2]You can specify an upper bound, or a lower bound, but not both.

19.2 Interface classes

Interface classes are a special kind of classes that contain *method signatures.* There is no implementation code, just the signatures. A particular interface class may have zero, one, or any number of such method signatures. Usually the number of methods in the interface is small.

A normal Java class can declare that it `implements` an interface. The class fulfills this declaration by providing implementations for all methods specified in the interface.

A Java interface is also a data type, an *interface type.* When a class implements an interface, instances of that class are instances of the interface type. They can be assigned to variables of the interface type, or placed in arrays, or collections that hold references to instances of the interface type.

When an object instance is accessed through the interface type, *only the methods specified in the interface can be accessed.*

19.2.1 Interface `Comparable`

One example of an interface out of the Standard Library is that of interface `java.lang.Comparable`:

```
public interface Comparable<T> {

  public int compareTo(T o);
}
```

Interface `Comparable` makes it possible for a class to define how its instances compare to each other (or to other classes). This *natural ordering* is useful if the class is to be sorted or searched by the methods in `java.util.Collections`.

Note how the declaration of interface `Comparable` has a generic type parameter. The type parameter is used to *restrict* the implementation of the interface, so that it only applies to types given in the type parameter. The type parameter is used by the class that implements the interface.

Implementing an interface

A class that implements an interface must provide implementations for all methods listed in the interface. Sometimes, some of these methods are empty, and just present to satisfy the compiler. For this reason, it is wise to have as few methods as possible in an interface. The required minimum, and no more, should be the rule.

Here is shown an example of how to implement an interface, by implementing interface `Comparable` for class `Point` (see appendix A for full source code). The first step is to declare that class `Point` implements `Comparable`:

```
public class Point implements Comparable<Point>
```

The keyword `implements` is placed after the class name, followed by a comma-separated list of the interfaces we implement. In this case the list only contains one interface, so there is no need for a comma.

The type parameter `T` in the interface is replaced by class `Point`, because we are only ready to compare a `Point` against another `Point`. Then we must implement the method `compareTo` in the class:

```
  public int compareTo(Point p) {
    return (int) Math.signum(distance() - p.distance());
  }
```

The documentation for `java.lang.Comparable` states that method `compareTo` should return a negative integer if `this` object is smaller than the other object, zero if they are equal, and a positive integer if `this` object is greater than the other object.

For planar points there is no directly obvious concept of *smaller* or *greater*, so we use the distance from the origin (0,0) as our measure. If point a is closer to the origin than point b, then $a < b$.

In order to find out the distance to the origin, we use the `Point` method `distance`:

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

and simply subtract. A negative result means that `this` point is the smaller one.

However, we cannot simply round or truncate the result into an integer. If both points have very similar distances, the difference may be very small, for example:

```
  -0.00034
```

If we truncate away the fraction, we are left with -0, but the `int` data type cannot represent that, so the result will be just, 0. That is incorrect. We therefore call `Math.signum` to extract the sign of the result, and *that* value can be safely typecast to an `int`, and returned as the value of the comparison.

19.2.2 Empty interfaces

An interface is allowed to contain no method signatures. The result is an empty interface, one which only has a name. These are, however, not useless. They are used as a kind of markers, or indicators.

One such marker interface is `java.io.Serializable`. Implementing this interface is required for a class that is to be serizalied (written to a stream) and deserialized (read from a stream). There are no methods in `Serializable`, it is a marker interface. Its presence means that the programmer has made an active decision about serialization, and taken steps to support it.

19.2.3 References to interface types

A variable can be declared to be of an interface type. It can then refer to object instances that implement the interface. If all we are interested in is the presence and capabilities of the interface, then we do not care about the rest of the object. We can refer to it, and access it through the interface type.

References to interface types enables additional dimensions of polymorphism, in that different classes can implement the methods in the interface differently. It would be perfectly viable to rewrite the `Shape` hierarchy (appendix A) using an interface like this:

```
import java.awt.Graphics;

public interface DrawableShape {

  public void draw (Graphics g);

}
```

In the program `DemoShape`, we could then simply refer to the shapes as:

```
DrawableShape [] shapes = ...
```

and the only method that can be accessed through array `shapes` would be `DrawableShape.draw`.

Chapter 20

Interfaces

For an introduction to interfaces, see 19.2.

20.1 Type independent programming with interfaces

Interfaces allow the use of interface types, and access to object instances through those types. Since any class can implement any number of interfaces, this enables an additional way of doing type independent programming, in addition to the one offered by class hierarchy.

20.1.1 Hierarchies of interfaces

An interface class can be extended. The extending interface can thus add methods to the interface. For example:

```
public interface A {

  public void a();

}

public interface B extends A {

  public void b();

}
```

Interface B extends interface A. This means that interface B inherits all methods in A. Any class that wants to implement interface B, must therefore implement both method a and method b:

```
public class C implements B {

  public void a() {  // required by interface B
    ...              // because B extends A
  }

  public void b() {  // specified by interface B
    ...
  }

  public void c() {  // local to class C
    ...
  }

}
```

It should come as no surprise that it is possible to refer to an instance of class C in three different ways:

```
C cref = new C(); // create an instance of C
B bref = cref;    // reference to interface type B
A aref = cref;    // reference to interface type A
```

Since class C implements interface B, a reference type of B can be used to refer to C. Similarly, because interface B extends interface A, a reference of type A is also a valid reference to C. However, depending on the reference used, there are restrictions on what we can access in the object instance.

The reference in variable cref is of type C, so we have full access to all public members in C:

```
cref.c(); // ok
cref.b(); // ok
cref.a(); // ok
```

The reference in variable bref is of type B, so we can only access members that are in interface B:

```
bref.c(); // illegal, no member c in B
bref.b(); // ok
bref.a(); // ok
```

The reference in variable aref is of type A. It can only be used to access the members defined in interface A:

```
aref.c(); // illegal, no member c in A
aref.b(); // illegal, no member b in A
aref.a(); // ok
```

Interfaces as parameters and return type

Interface types can also be used (and often to advantage) to specify the types of method parameters or return types. Using the example above, one can imagine a method that operates on instances of interface `B`:

```
public void doSomeBstuff (B obj) {
  ...
}
```

Of course, the implementation of method `doSomeBstuff` can only call the methods specified in interface `B`. It cannot, and should not, need to know about anything else.

Similarly, it is equally possible to use an interface type as a method return type:

```
public B getNext() {
  ...
}
```

Method `getNext` returns a reference to an object that implements interface `B`.

20.1.2 `java.util.Iterator<E>`

A good example of a method that returns an interface type, is the method `iterator` found in many of the `java.util` collection classes. The definition of the method is:

```
Iterator<E> iterator()
```

This means, that when you call method `iterator`, the return value is a reference to an object that implements interface `Iterator<E>` (note the capital 'I'), which is an iterator defined for the element type `E` of that collection reference.

The purpose of an iterator is to allow the caller to inspect all elements in a collection in a type-indedepent way. Interface `Iterator` itself is short:

```
boolean hasNext() // Returns true if the iteration
                  // has more elements.
E       next()    // Returns the next element
                  // in the iteration.
void    remove()  // Removes the most recent
                  // element from the collection
                  // (optional side-effect)
```

If we want to implement interface `Iterator<E>` in our own class, the following things should be remembered:

- The object that actually implements interface `Iterator` can be the same object instance that has the data elements, or an instance of a nested class, or a completely different object that we load up and configure when creating the iterator.
- Method `next` has return type `E`, so we need to qualify the return type of this method with the element types that we support.
- If we are careful about giving away references to encapsulated and mutable data, we may want to make sure that the iterator presents references to copies of the data elements.
- To implement the interface we must implement all three methods: `hasNext`, `next`, and `remove`. However, `remove` is optional so while it must exist it does not have to do anything (our Javadoc documentation should state if removal is possible or not).

Using an iterator to inspect all elements is easy. Assume that we have created a `LinkedList` of `Shape`:

```
LinkedList<Shape> lst = ...;

for(Iterator itr = lst.iterator(); its.hasNext();)
{
  Shape sh = itr.next();
  ...
}
```

20.1.3 java.lang.Iterable<T>

In package `java.lang`, there is the interface `Iterable<T>`. This interface has a single method:

```
Iterator<T> iterator()
```

Put simply, to implement interface `Iterable`, there must be a method called `iterator` that returns an `Iterator` over some type `T`.

The `Iterable` interface is in the `java.lang` package, which is the package for the Java language itself. It is put there, because any object that implements this interface can be used in a `for` each loop. The collection classes in `java.util` implement this interface, and for that reason the above example can be simplified:

```
LinkedList<Shape> lst = ...;

for (Shape sh : lst) {
  ...
}
```

20.2 Interfaces vs multiple inheritance

Java does not have multiple inheritance; each class has a single superclass. The use of interfaces, however, allows a class to *be* several other types as well, apart from the main inheritance chain. As interface classes may also form inheritance chains, a limited sort of multiple inheritance is therefore possible, at least in the datatype sense.

Of course, on the standard class inheritance chain, a class inherits variables, methods, and other properties of its superclasses. When implementing an interface, nothing is really inherited. The class is given is the opportunity to be an instance of the interface type. In return, the class must implement all methods of the interface.

While classes on the inheritance chain are related to each other by greater complexity downwards, towards the subclasses, and more simplicity and abstraction upwards, towards the superclasses, the classes that implement an interface are by definition exactly the same. They implement the interface, period. Their implementations of the same interface may be different from each other, but unless they fulfill the contract established by the interface, the program will break.

20.3 Additional members in interfaces

The Java language allows interfaces to contain additional members in addition to method signatures. Such members are automatically made `public` and `static`, although it is allowed to declare them as such as a reminder.

20.3.1 Constants in interfaces

A variable in an interface will automatically be `public`, `static`, and `final`, i.e. a constant. The variable *must* be a constant, because it is not possible to create an instance of an interface. Without an instance there is no memory allocated for a variable.

The proper use of interface constants is, however, debated. The main objection against interface constants is that interfaces should concern themselves with the *services* defined by the interface. Constants are part of the implementation. Common constants are therefore better placed in a class.

20.4 Interfaces and inner classes

It is allowed to declare classes inside an interface. The classes will automatically be `public` and `static`.

Placing a class inside an interface may be a good idea if:

- the methods declared in the interface use the class as parameter variable types, return value types, exceptions, or enums, and
- the type declared by the interface class has no obvious use without the interface

Here is one example, modelled after an API to a sensor:

```
public interface SensorListener {

  public static class SensorEvent {
    public static final byte SENS_ERROR  = -1;
    public static final byte SENS_RESET  =  0;
    public static final byte SENS_UPDATE =  1;

    public byte state;
    public byte data;
  }

  public void sensorUpdate (SensorEvent event);
}
```

A normal class that wants to be fed updated sensor values then implements the `SensorListener` interface. The class `SensorEvent` in the interface defines how the update is delivered, and how it should be interpreted. An implementation could look like this:

```
public class TempWatcher implements SensorListener {

  private double lastDataKelvin = 273.0;

  // In interface SensorListener:
  public void sensorUpdate (SensorEvent event) {

    if (event.state == SensorEvent.SENS_UPDATE) {

        lastDataKelvin = 273.0 + event.data;

    }
    else if (event.state == SensorEvent.SENS_RESET)
      // manage a sensor reset
    else if (event.state == SensorEvent.SENS_ERROR)
      // manage the error state
  }
}
```

Here is another example. An interface defines an exception to be used with the interface:

```
public interface Actuator {

  public static class StuckException extends Exception {}

  public void setPosition (int position)
    throws StuckException;
}
```

The class that implements the `Actuator` interface must implement method `setPosition`. If the actuator cannot move from its current position, the method must throw a `StuckException`.

20.5 Anonymous inner classes

Type-independent programming using interfaces is a powerful way to produce general and reusable code. For example, a sorting routine can be written that can sort any sequence of objects, as long the objects implement the interface `Comparable`. The sorting routine does not need to know what the objects are, or how they are constructed internally. All it needs to know, is how to ask a pair of objects how they compare relative to each other. By implementing `Comparable` the objects themselves decide which one of them is lesser and greater.

Classes that implement `java.lang.Comparable<T>` implement the single method:

```
public int compareTo(T o)
```

This method takes a single argument, so the conceptual use of the interface in action, is to pick one object, and then ask it how it compares to some other object. The object responds, according to its implementation of `Comparable`.

Interface `Comparable` provides the concept of a *natural ordering*, or default ordering, for a class. This is very useful, but in some cases we may want to sort the objects according to a different criterion. We want to deviate from the natural ordering.

We can provide an alternative ordering by finding an implementing of the interface `java.util.Comparator<T>`. This interface has two methods:

```
// Compare two objects
public int compare(T o1, T o2)
```

```
// Compare this Comparator against some other object
boolean equals(Object obj)
```

We can safely ignore method `equals` in this example. Method `compare`, on the other hand, takes two references, and then determines which one is the lesser one. If `o1` is smaller than `o2`, return a negative integer; if `o1` is greater than `o2`, return a positive integer; if `o1` and `o2` are equal (in terms of sort ordering), return zero.

Assume now that we want to sort an array of `Shape` (see appendix A). Class `Shape` does not have a natural ordering defined, and we see no reason to put one in. We just want to be able to sort some shapes according to their area.

The class `java.util.Arrays` has static utility methods for arrays. Among other things, we can have an array sorted, but we must provide an instance of `Comparator`. The method we want to call for the sorting looks like this:

```
public static <T> void sort(T[] a, Comparator<? super T> c)
```

This means:

- The method is `public` and `static`
- The declaration uses the generic type parameter `T`
- The return type is `void`
- The method is named `sort`
- The method takes two arguments
- The first argument is an array with element type `T`
- The second argument is a `Comparator` defined for type `T`, or a superclass of `T`.

Let us dwell a moment on the expression `Comparator<? super T>`. This is a wildcard expression, with a lower type bound. The lower bound says that the array must be of type `T`, but the `Comparator` is allowed to be defined for a superclass of `T`.

If we apply that to our `Shape` class hierarchy, this means that if we define a `Comparator<Shape>`, then we can use the very same comparator to sort:

```
Shape []   // array of Shape
Oval []    // array of Oval
Circle [] // array of Circle
Rectangle [] // array of Rectangle
Square [] // array of Square
Triangle [] // array of Triangle
```

This is very good!

If the second parameter had been defined like this:

```
Comparator<T> c
```

then the compiler would have required that the element type of the array and the definition type of the `Comparator` were the same, and we would have had to write separate comparators for each possible array. But, since we want to compare the shapes based on their area, and we know class `Shape` has he method `area`, a single `Comparator<Shape>` will suffice. We only need to write it once.

So then, we need to write a class that implements `Comparable<Shape>`. Do we *really* need to create a separate source file for this? Well no, we can use an inner class:

```
import java.util.Arrays;

public class ShapeHack {
  ...

  private static
    class AreaComp implements Comparator<Shape>
  {
    public int compare(Shape o1, Shape o2) {
      return (int) Math.signum(o1.area() - o2.area());
    }
  }

  ...

  {
    Shape [] shapes = ... // some shapes
    AreaComp comp = new AreaComp();

    // Sort the shapes array using an instance of
    // our local Comparable

    Arrays.sort (shapes, comp);
  }
}
```

The above version of class `ShapeHack` is ok, but its still a bit spread out. There is the inner class definition, there is a variable for it, and finally the call. If the program becomes large, these parts will be spread out in the source code. If we only do the special sorting once, would it not be great if the parts that belong together, could actually be written together?

It turns out, that this is actually possible. The solution is something called *anonymous subclassing*, or in the case of interfaces *anonymous implementation*. If we only need *one* instance of a class used only *once*, there is no need to give it a place and a name and so on. We can define it as an anonymous class in the spot where we need it. Then it looks like this:

```
import java.util.Arrays;

public class ShapeHack {
  ...
  {
    Shape [] shapes = ... // some shapes

    Arrays.sort (shapes,
                 new Comparator<Shape> ()
                 {
                   public int compare(Shape o1, Shape o2)
                   {
                     return (int)
                       Math.signum(o1.area() - o2.area());
                   }
                 } // end of anonymous class
                ); // end of method call
  }
}
```

Examine carefully the second argument to `Arrays.sort`. It is the `new` operator, creating a new instance of `Comparator<Shape>`. The empty parentheses are for the default constructor, because anonymous subclassing only works with that. What are we subclassing from? Since we are giving an interface to the `new` operator, we are subclassing from `Object`.

Then, after the empty parentheses follows the body of the anonymous class. Since we are implementing an interface, we must provide the methods in the interface. As mentioned above, method `equals` is already inherited from `Object` so there is no need to mention it again. All we have to do, is to give our version of method `compare`.

Appendix A

Sources

A.1 Class Point

A.1.1 File Point.java

```
// Class Point represents a 2D planar point.
// A Point is immutable and stored with double precision.
public class Point implements Comparable<Point> {
  private double x;
  private double y;

  // Return a new point randomly located inside a region centered
  // on (0,0) and with the given width and height.
  public static Point getRandomPoint (double width, double height) {
    double x = width  * (-0.5 + Math.random());
    double y = height * (-0.5 + Math.random());
    return new Point(x, y);
  }

  // Creates a new Point at the given coordinates
  public Point (double x, double y) {
    this.x = x;
    this.y = y;
  }

  // Creates a new Point that is a copy of the
  // given Point
  public Point (Point p) {
    x = p.x;
    y = p.y;
  }

  // Returns a new Point located halfway
  // between this point and the given Point.
  public Point midPoint (Point p) {
    return new Point ((x + p.x) / 2d, (y + p.y) / 2d);
  }

  // Returns the x coordinate of this Point.
  public double getX() {
    return x;
  }

  // Returns the y coordinate of this Point.
  public double getY() {
    return y;
  }

  // In interface Comparable:
  public int compareTo(Point p) {
    return (int) Math.signum(distance() - p.distance());
  }

  // Returns the length of the distance between
  // the origin (0,0) and this point.
  public double distance () {
    return Math.sqrt(x * x + y * y);
  }

  // Returns the Euclidian distance between this
  // Point and the given Point.
  public double distance (Point p) {
    double xr = x - p.x;
    double yr = y - p.y;
    return Math.sqrt(xr * xr + yr * yr);
  }

  // Returns a string representation of this Point.
  public String toString () {
    return String.format("(%.2f,%.2f)", x, y);
  }

}
```

A.2 The Shape hierarchy — classes

A.2.1 File Shape.java

```
import java.awt.Graphics;

// This is the superclass of all the various shapes.
// Each shape has an origin that is managed by the
// Shape class. The origin marks the center or centroid
// of the shape. The class is abstract because it does
// not provide implementations for the methods area
// and draw. These must be provided by subclasses.
```

```
public abstract class Shape {
  // The origin of this shape
  protected Point origin;

  // Create a new Shape with the given origin
  public Shape (Point origin) {
    this.origin = new Point(origin);
  }

  // Return the origin of this Shape
  public Point getOrigin () {
    return new Point(origin);
  }

  // Compute and return the area of this Shape
  public abstract double area ();

  // Draw this Shape on the given graphics context
  public abstract void draw (Graphics g);

  // This method returns the toString() description for
  // class Shape. It is put here for the benefit of
  // subclasses for which the toString() result of their
  // immediate superclass is not useful.
  protected String shapeDescription () {
    return String.format("%s origin=%s",
                         getClass().getName(),
                         origin.toString());
  }

  // Returns a string describing this Shape
  public String toString () {
    return shapeDescription();
  }
}
```

A.2.2 File Oval.java

```
import java.awt.Graphics;

// The class Oval represents an oval shape (ellipse).
public class Oval extends Shape {

  // Two orthogonal radii define the Oval
  double radius1;
  double radius2;

  // Creates a new Oval with the given parameters.
  public Oval (Point origin, double radius1, double radius2) {
    super(origin); // install the origin
    this.radius1 = radius1;
    this.radius2 = radius2;
  }
  // Returns the first radius
  public double getRadius1 () {
    return radius1;
  }

  // Returns the second radius
  public double getRadius2 () {
    return radius2;
  }

  // Returns the area of this Oval
  public double area () {
    return Math.PI * radius1 * radius2;
  }

  // Draws this oval centered on the origin.
  public void draw (Graphics g) {
    int x = (int) (origin.getX() - radius1);
    int y = (int) (origin.getY() - radius2);
    int width  = (int) (2d * radius1);
    int height = (int) (2d * radius2);

    g.drawOval(x, y, width, height);
  }

  // Returns a string representation of this Oval
  // Note that in (1) we call toString in our superclass
  // (Shape) to get our class name and origin.
  public String toString () {
    return String.format("%s radius1=%.2f radius2=%.2f",
                         super.toString(), // (1)
                         radius1,
                         radius2);
  }
}
```

A.2.3 File Circle.java

```
// Class Circle represents a circle. This is done
// by creating a special case Oval where both radii
// have equal length.
public final class Circle extends Oval {

  // Create a new Circle with the given parameters.
  public Circle (Point origin, double radius) {
    // Call the constructor in Oval, giving our
    // radius twice
    super(origin, radius, radius);
  }

  // Returns our radius. This is a convenience method
  // to emphasize that the circle has a single radius.
  public double getRadius () {
    return getRadius1();
  }

  // Returns a string representation of this Circle.
  // We want our class name and origin, but if we were
  // to call Oval.toString we would get the two radii
  // as well. For this reason we (1) bypass Oval and fetch
  // the name and origin directly from the service method
  // Oval.shapeDescription.
  public String toString () {
    return String.format("%s radius=%.2f",
                         shapeDescription(), // (1)
                         getRadius());
  }
}
```

A.2.4 File Rectangle.java

```
import java.awt.Graphics;

// This class represents a rectangle, the sides of which
// can have different lengths. For convenience the sides
// are modelled as the width and height of the rectangle.
public class Rectangle extends Shape {

  double width;
  double height;

  // Creates a new Rectangle with the given parameters.
  public Rectangle (Point origin,
                    double width,
                    double height)
  {
    super(origin); // install origin
    this.width = width;
    this.height = height;
  }

  // Returns the width of this Rectangle.
  public double getWidth () {
    return width;
  }

  // Returns the height of this Rectangle.
  public double getHeight () {
    return height;
  }

  // Returns the area of this Rectangle
  public double area () {
    return width * height;
  }
  // Draws this Rectangle centered on the origin.
  public void draw (Graphics g) {
    int x = (int) (origin.getX() - width / 2d);
    int y = (int) (origin.getY() - height / 2d);
    g.drawRect(x, y, (int) width, (int) height);
  }

  // Returns a string representation of this Rectangle.
  public String toString () {
    return String.format("%s width=%.2f height=%.2f",
                         super.toString(),
                         width,
                         height);
  }
}
```

A.2.5 File Square.java

```
// The class Square represents a square by making
// a special case Rectangle in which both sides
// have the same length.
public final class Square extends Rectangle {

  // Create a new Square with the given parameters
  public Square (Point origin, double side) {
    // Call the constructor in Rectangle
    super(origin, side, side);
  }

  // A convenience method to emphasize that the
  // square has only one side length.
  public double getSide () {
    return getWidth();
  }

  // Returns a string representation of this Square.
  // We do not use Rectangle.toString because it contains
  // width and height properties. Instead we go directly (1)
  // to Shape.shapeDescription and retrieve our class
  // name and origin from there.
  public String toString () {
    return String.format("%s side=%.2f",
                         shapeDescription(), // (1)
                         getSide());
  }
}
```

A.2.6 File Triangle.java

```
import java.awt.Graphics;
```

```
// Class Triangle represents a triangle defined by
// three corner points.
public class Triangle extends Shape {

  private Point p1, p2, p3;

  // Returns a new Point located at the centroid
  // of the triangle defined by the three given
  // corner points.
  public static Point
    centroid(Point p1, Point p2, Point p3)
  {
    return
      new Point((p1.getX() + p2.getX() + p3.getX()) / 3d,
                (p1.getY() + p2.getY() + p3.getY()) / 3d);
  }

  // Returns a new Triangle defined by the given
  // corner points.
  public static Triangle
    triangleFactory (Point p1, Point p2, Point p3)
  {
    return new
      Triangle (centroid(p1, p2, p3), p1, p2, p3);
  }

  // Creates a new Triangle with the given origin
  // and corner points.
  //
  // Note that the constructor is private. Triangles
  // are created by calling the static factory method
  // Triangle.triangleFactory.
  //
  // The reason for this is that our constructor must
  // call the superclass constructor with the origin
  // as the first thing it does. Since we do not trust
  // the general caller to provide a correct origin, we
  // instead refer to the factory method which computes
  // the origin before calling the constructor.
  private Triangle (Point origin, Point p1, Point p2, Point p3) {
    super(origin);
    this.p1 = new Point(p1);
    this.p2 = new Point(p2);
    this.p3 = new Point(p3);
  }

  // Returns the area of this Triangle.
  public double area () {
    double a = p1.distance(p2);
    double b = p2.distance(p3);
    double c = p3.distance(p1);

    double s = (a + b + c) / 2.0; // semiperimeter
    // Heron's formula
    return Math.sqrt(s * (s - a) * (s - b) * (s - c));
  }

  // Draws this Triangle.
  public void draw (Graphics g) {
    int x1 = (int) p1.getX();
    int y1 = (int) p1.getY();
    int x2 = (int) p2.getX();
    int y2 = (int) p2.getY();
    int x3 = (int) p3.getX();
    int y3 = (int) p3.getY();

    g.drawLine(x1, y1, x2, y2);
    g.drawLine(x2, y2, x3, y3);
    g.drawLine(x3, y3, x1, y1);
  }

  // Returns a string representation of this Triangle.
  public String toString() {
    return String.format("%s p1=%s p2=%s p3=%s",
                         super.toString(),
                         p1.toString(),
                         p2.toString(),
                         p3.toString());
  }
}
```

A.3 The Shape hierarchy — test and demo

A.3.1 File TestShape.java

```
// The TestShape class generates random shapes and then
// prints them, using their toString methods.
public class TestShape {
  // These constants limit the area and dimensions
  // of the randomly generated shapes.
  public static final double AREA_WIDTH_LIMIT  = 200;
  public static final double AREA_HEIGHT_LIMIT = 200;
  public static final double HORIZ_LENGTH_LIMIT = 50;
  public static final double VERTI_LENGTH_LIMIT = 50;

  // Returns a random shape
  public static Shape randomShape () {
    // Randomize parameters
    Point p = Point.getRandomPoint(AREA_WIDTH_LIMIT,
                                   AREA_HEIGHT_LIMIT);
    double a = HORIZ_LENGTH_LIMIT * Math.random();
    double b = VERTI_LENGTH_LIMIT * Math.random();

    // Pick a random number between 0 and 4
    int select = (int) (5d * Math.random());
    switch (select) {
    case 0:
      return new Oval(p, a, b);
    case 1:
      return new Circle(p, a);
    case 2:
      return new Rectangle(p, a, b);
    case 3:
      return new Square(p, a);
    case 4:
      return
        Triangle.triangleFactory(p,
                                 Point.getRandomPoint(200,200),
                                 Point.getRandomPoint(200,200));
    }

    return null;
  }

  // Return an array of n random shapes.
  public static Shape [] randomShapes (int n) {
    Shape [] rtn = new Shape [n];

    for (int i = 0; i < n; i++) {
      rtn[i] = randomShape();
    }

    return rtn;
  }

  // Generate n random shapes and print them.
  public static void sampleShapes (int n) {
    Shape [] shapes = randomShapes(n);

    for (Shape sh : shapes) {
      System.out.println(sh);
      System.out.printf("area : %.2f%n", sh.area());
      System.out.println();
    }
  }

  public static void main (String [] args) {

    sampleShapes(10);
  }
}
```

A.3.2 File DemoShape.java

```
import java.awt.Graphics;
import java.awt.Graphics2D;
```

```java
import java.awt.Dimension;
import java.awt.Color;
import javax.swing.JPanel;
import javax.swing.JFrame;

// This class extends JPanel so that we get a graphics
// component we can paint coloured lines and shapes on.
public class DemoShape extends JPanel {

  // The shapes array holds the shapes we paint
  Shape [] shapes;

  // Create a new DemoShape with the given array of shapes.
  public DemoShape (Shape [] shapes) {

    // How big we wish this component to be
    setPreferredSize(new Dimension(400,400));
    setBackground(Color.black);
    this.shapes = shapes;
  }

  // This method is called by the GUI system when this
  // JPanel needs to be painted. This can be after a
  // move, a resize, a lifted occlusion, or a window
  // restore. We do not know and we do not care. We just
  // paint it as it is supposed to look.
  public void paintComponent (Graphics g) {

    // Clear to background color by calling the super-
    // class version of this method.
    super.paintComponent(g);

    // Typecast downwards as documented in std lib
    Graphics2D g2 = (Graphics2D) g;

    // Find out size of this JPanel
    Dimension size = getSize();

    // Translate graphics to center of JPanel
    g2.translate(size.width / 2, size.height / 2);

    g2.setColor(Color.white);

    // Draw all shapes using polymorphism
    for (Shape sh : shapes)
      sh.draw(g);
  }

  // The main program does not have to be in this
  // class, but it is a good spot.
  public static void main (String [] args) {

    // The JFrame represents the outermost window, the
    // window that comes from the operating system.
    JFrame frame = new JFrame("DemoShape");

    // Exit when the close button (X) is clicked
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Get the content pane of the JFrame and stuff it
    // with an instance of DemoShape. We get the shapes
    // from our TestShape class.
    frame.getContentPane().add(new DemoShape (TestShape.randomShapes(10)));

    // Put it all together
    frame.pack();

    // And show it
    frame.setVisible(true);
  }
}
```

A.4 The OverShape demo

```
// The OverShape program demonstrates the effect of
// overloading. The compiler selects the method to
// call at compile-time, by matching the type of
// the calling expression to the appropriate method.

public class OverShape {

  // Six different versions of the inspect method.
  // There is one method for each type in the
  // Shape type hierarchy.

  public void inspect (Circle c) {
    System.out.printf("Circle : %s%n", c);
  }

  public void inspect (Oval o) {
    System.out.printf("Oval : %s%n", o);
  }

  public void inspect (Triangle t) {
    System.out.printf("Triangle : %s%n", t);
  }

  public void inspect (Rectangle r) {
    System.out.printf("Rectangle : %s%n", r);
  }

  // The Square version is commented out on purpose.
  //
  // public void inspect (Square sq) {
  //   System.out.printf("Square : %s%n", sq);
  // }
  public void inspect (Shape sh) {
    System.out.printf("Shape : %s%n", sh);
  }

  public void test () {
    double w = 50;
    double h = 50;
    Point pt = Point.getRandomPoint(w, h);

    Circle c  = new Circle(pt, 5.0);
    Oval o    = new Oval(pt, 3.0, 3.0);
    Square sq = new Square(pt, 4.0);

    inspect(c);  // Selects the Circle version
    inspect(o);  // Selects the Oval version
    inspect(sq); // Selects the Rectangle version

    Shape [] shapes = {
      new Circle(pt, 3), new Oval(pt, 4,5),
      new Square(pt, 6), new Rectangle(pt, 7, 8),
      Triangle.triangleFactory
      (Point.getRandomPoint(w, h),
       Point.getRandomPoint(w, h),
       Point.getRandomPoint(w, h))
    };

    for (Shape sh : shapes)
      inspect (sh); // Selects the Shape version
  }

  public static void main (String [] args) {
    new OverShape().test();
  }
}
```