

Java Just in Time: task questions

John Latham

March 22, 2011

Contents

1	Chapter 1 Introduction	6
1.6	Section / task 1.6 Our first Java program	6
1.7	Section / task 1.7 Our second Java program	6
2	Chapter 2 Sequential execution and program errors	6
2.2	Section / task 2.2 Hello world	6
2.3	Section / task 2.3 Hello world with a syntactic error	7
2.4	Section / task 2.4 Hello world with a semantic error	7
2.5	Section / task 2.5 Hello solar system	8
2.6	Section / task 2.6 Hello solar system with a run time error	8
2.7	Section / task 2.7 Hello anyone	9
2.8	Section / task 2.8 Hello anyone with a logical error	9
2.9	Section / task 2.9 Hello solar system, looking at the layout	10
3	Chapter 3 Types, variables and expressions	10
3.2	Section / task 3.2 Age next year	10
3.4	Section / task 3.4 Age next year with a command line argument	11
3.5	Section / task 3.5 Finding the volume of a fish tank	12
3.6	Section / task 3.6 Sum the first N numbers – incorrectly	12
3.7	Section / task 3.7 Disposable income	13
3.8	Section / task 3.8 Sum the first N numbers – correctly	15
3.9	Section / task 3.9 Temperature conversion	16

4	Chapter 4 Conditional execution	17
4.2	Section / task 4.2 Oldest spouse 1	17
4.3	Section / task 4.3 Oldest spouse 2	17
4.4	Section / task 4.4 Film certificate age checking	18
5	Chapter 5 Repeated execution	18
5.2	Section / task 5.2 Minimum tank size	18
5.3	Section / task 5.3 Minimum bit width	19
5.5	Section / task 5.5 Compound interest one	19
5.6	Section / task 5.6 Compound interest two	20
5.7	Section / task 5.7 Average of a list of numbers	21
5.8	Section / task 5.8 Single times table	22
5.9	Section / task 5.9 Age history	23
5.10	Section / task 5.10 Home cooked Pi	23
6	Chapter 6 Control statements nested in loops	24
6.2	Section / task 6.2 Film certificate age checking the whole queue	24
6.3	Section / task 6.3 Dividing a cake (GCD)	24
6.4	Section / task 6.4 Printing a rectangle	25
6.5	Section / task 6.5 Printing a triangle	25
6.6	Section / task 6.6 Multiple times table	26
6.7	Section / task 6.7 Luck is in the air: dice combinations	27
7	Chapter 7 Additional control statements	28
8	Chapter 8 Separate methods and logical operators	28
8.2	Section / task 8.2 Age history with two people	28
8.3	Section / task 8.3 Age history with a separate method	29
8.4	Section / task 8.4 Dividing a cake with a separate method for GCD	29
8.5	Section / task 8.5 Multiple times table with separate methods	30
8.6	Section / task 8.6 Age history with day and month	30
8.7	Section / task 8.7 Truth tables	31
8.8	Section / task 8.8 Producing a calendar	31

9 Chapter 9 Consolidation of concepts so far	32
10 Chapter 10 Separate classes	32
10.2 Section / task 10.2 Age history with Date class	32
10.3 Section / task 10.3 Improving the Date class: lessThan() and equals() methods .	33
10.4 Section / task 10.4 Improving the Date class: toString() method	34
10.5 Section / task 10.5 Improving the Date class: addYear() method	35
11 Chapter 11 Object oriented design	36
11.2 Section / task 11.2 Age history revisited	36
11.3 Section / task 11.3 Greedy children	40
12 Chapter 12 Software reuse and the standard Java API	45
12.2 Section / task 12.2 A reusable Date class, with doc comments	45
12.5 Section / task 12.5 Simple Encryption	46
13 Chapter 13 Graphical user interfaces	49
13.2 Section / task 13.2 Hello world with a GUI	49
13.3 Section / task 13.3 Hello solar system with a GUI	49
13.4 Section / task 13.4 Hello solar system with a GridLayout	49
13.5 Section / task 13.5 Adding JLabels in a loop	50
13.7 Section / task 13.7 Stop clock	51
13.8 Section / task 13.8 GCD with a GUI	52
13.9 Section / task 13.9 Enabling and disabling components	52
13.12Section / task 13.12 Single times table with a ScrollPane	52
14 Chapter 14 Arrays	53
14.2 Section / task 14.2 Salary analysis	53
14.3 Section / task 14.3 Sorted salary analysis	54
14.4 Section / task 14.4 Get a good job	55
14.5 Section / task 14.5 Sort out a job share?	56
14.6 Section / task 14.6 Diet monitoring	58
14.7 Section / task 14.7 A weekly diet	62

15 Chapter 15 Exceptions	68
15.2 Section / task 15.2 Age next year revisited	68
15.3 Section / task 15.3 Age next year with exception avoidance	69
15.4 Section / task 15.4 Age next year with exception catching	69
15.5 Section / task 15.5 Age next year with multiple exception catching	69
15.6 Section / task 15.6 Age next year throwing an exception	69
15.7 Section / task 15.7 Single times table with exception catching	70
15.8 Section / task 15.8 A reusable Date class with exceptions	70
16 Chapter 16 Inheritance	71
16.3 Section / task 16.3 The Person class	71
16.4 Section / task 16.4 The AudienceMember class	73
16.5 Section / task 16.5 The Punter class	73
16.6 Section / task 16.6 The Person abstract class	74
16.7 Section / task 16.7 The remaining simple subclasses of Person	74
16.8 Section / task 16.8 The MoodyPerson classes	75
16.11Section / task 16.11 The Game class	76
16.12Section / task 16.12 The Worker classes	79
16.13Section / task 16.13 The CleverPunter class	80
16.15Section / task 16.15 The Object class and constructor chaining	80
16.16Section / task 16.16 Overloaded methods versus override	81
17 Chapter 17 Making our own exceptions	81
17.3 Section / task 17.3 The Date class with its own exceptions	81
17.4 Section / task 17.4 The Notional Lottery with exceptions	81
18 Chapter 18 Files	83
18.2 Section / task 18.2 Counting bytes from standard input	83
18.3 Section / task 18.3 Counting characters from standard input	84
18.4 Section / task 18.4 Numbering lines from standard input	85
18.5 Section / task 18.5 Numbering lines from text file to text file	87
18.6 Section / task 18.6 Numbering lines from and to anywhere	87
18.7 Section / task 18.7 Text photographs	88

18.8 Section / task 18.8 Contour points	89
19 Chapter 19 Generic classes	90
19.2 Section / task 19.2 A pair of any objects	90
19.3 Section / task 19.3 A generic pair of specified types	91
19.4 Section / task 19.4 Autoboxing and auto-unboxing of primitive values	91
19.5 Section / task 19.5 A conversation of persons	92
20 Chapter 20 Interfaces, including generic interfaces	92
20.3 Section / task 20.3 Sorting a text file using an array	92
20.4 Section / task 20.4 Translating documents	93
20.5 Section / task 20.5 Sorting valuables	93
21 Chapter 21 Collections	94
21.2 Section / task 21.2 Reversing a text file	94
21.3 Section / task 21.3 Sorting a text file using an ArrayList	96
21.4 Section / task 21.4 Prime numbers	97
21.5 Section / task 21.5 Sorting a text file using a TreeSet	99
21.8 Section / task 21.8 Word frequency count sorted by frequency	99
21.9 Section / task 21.9 Collections of collections	100
22 Chapter 22 Recursion	102
22.3 Section / task 22.3 Lecture attendance	102
22.4 Section / task 22.4 Sum of ages of descendants	102
22.5 Section / task 22.5 Factorial	102
22.6 Section / task 22.6 Fibonacci	103
22.7 Section / task 22.7 Number puzzle	103
22.8 Section / task 22.8 Dice combinations	104
22.10Section / task 22.10 Tower of Hanoi	105
22.11Section / task 22.11 Friend book	105
23 Chapter 23 The end of the beginning	107

1 Chapter 1 Introduction

1.6 Section / task 1.6 Our first Java program

- Aim of example: To show the mechanics of processing a finished Java source program so that it can be **run**, through to actually running it.
- Coursework title: **Compile and run HelloWorld**
- Coursework summary: To **compile** and **run** the HelloWorld program.
- Question: Carefully type in the **source code** for the HelloWorld program, and save it in the appropriately named **file**. Check it to make sure you have not made any typing errors – otherwise you may get error messages that will alarm you! (Try to get the **indentation** right – e.g. line 3 has two spaces at the front, and line 5 has four.) Now **compile** and **run** it. Record your progress and any observations you make, in your logbook.

1.7 Section / task 1.7 Our second Java program

- Aim of example: To reinforce the process of the **compile** and **run** cycle of a Java program.
- Coursework title: **Compile and run HelloSolarSystem**
- Coursework summary: To **compile** and **run** the HelloSolarSystem program.
- Question: Carefully type in the **source code** for the HelloSolarSystem program, and save it in the appropriately named **file**. Check it to make sure you have not made any typing errors – otherwise you may get error messages that will alarm you! (Try to get the **indentation** right – e.g. line 3 has two spaces at the front, and line 5 has four.) Now **compile** and **run** it. Record your progress and any observations in your logbook.

2 Chapter 2 Sequential execution and program errors

2.2 Section / task 2.2 Hello world

- Aim of example: To introduce some very basic Java concepts, including the **main method** and `System.out.println()`.
- Coursework title: **HelloWorld in French**
- Coursework summary: Write a program to greet the whole world, in French!
- Question: Write a new version of HelloWorld which gives its greeting in French (or any other language of your choice, apart from English). You will learn most if you try to do this without looking at the original version. Your program should still be called HelloWorld, as only the message **data** is in French. Type the program in and save it in the appropriately named **file**. Check it against the original to see you have not made any

mistakes – otherwise you may get error messages that will alarm you. Then **compile** and **run** it.

2.3 Section / task 2.3 Hello world with a syntactic error

- Aim of example: To introduce the principle of program errors, in particular **syntactic errors**. We also see that a **string literal** must be ended on the same line it starts on.
- Coursework title: **Fortune syntactic errors**
- Coursework summary: Take a given program that has **syntactic errors** in it, and get it working.
- Question: Carefully type in the following program *exactly*. It contains some deliberate mistakes – type them in anyway even if you spot them as you read the code.

```
001: public class Fortune
002: {
003:     public static void main(String () args]
004:     {
005:         System.out.println('Sometimes having a fortune is too expensive!');
006:     }
007: }
```

Now **compile** the program and examine the error messages. Record these in your log-book, each along with your best attempt to explain the meaning and the cause of it. Can you see any errors which appear not to have caused a message? If so, record these too.

Now fix the errors *one at a time*, and compile the program in between. Record any new error messages that appear, along with your explanation for them.

Optional extra: Make the smallest number of edits to your HelloWorld program so as to have javac produce the largest number of **syntactic errors**! For example, what happens if you simply delete a space between two words of the program?

2.4 Section / task 2.4 Hello world with a semantic error

- Aim of example: To introduce **semantic errors** and note that these and **syntactic errors** are **compile time errors**.
- Coursework title: **ManchesterWeather semantic errors**
- Coursework summary: Take a given program that has **semantic errors** in it, and get it working.
- Question: Carefully type in the following program *exactly*. It contains some deliberate mistakes – type them in anyway even if you spot them as you read the code.

```
001: public class ManchesterWeather
002: {
003:     public static void main(spring[] args)
```

```
004:  {
005:      Cistern.flush.println("Rainfalls keep dropping on my head!");
006:  }
007: }
```

Now **compile** the program and examine the error messages. Record these in your log-book, each along with your best attempt to explain the meaning and the cause of it. Can you see any errors which appear not to have caused a message? If so, record these too.

Now fix the errors *one at a time*, and compile the program in between. Record any new error messages that appear, along with your explanation for them.

2.5 Section / task 2.5 Hello solar system

- Aim of example: To introduce the principle of **sequential execution**.
- Coursework title: **HelloFamily**
- Coursework summary: Write a program to greet some of your family.
- Question: Preferably without looking at HelloSolarSystem, write a program called HelloFamily which greets your maternal grand parents and all their descendants. Don't forget to include yourself. If you have a lot of relatives, then you may limit your program to around 12 of them if you wish. The greeting must be done in alphabetical order by name – so you had better plan the output before you start typing.

(Hint: one approach would be to type the names into a **text file**, one name per line, and then use some program which **sorts** lines of text – for example the sort program. After this the resulting text could be edited to become the final program.)

2.6 Section / task 2.6 Hello solar system with a run time error

- Aim of example: To introduce the principle of **run time errors**.
- Coursework title: **Quote run time errors**
- Coursework summary: Take a given program that has **run time errors** in it, and get it working.
- Question: Carefully type in the following program *exactly*. It contains some deliberate mistakes – type them in anyway even if you spot them as you read the code.

```
001: public class Quote
002: {
003:     public void Main(String args)
004:     {
005:         System.out.println("Programming is about making the stupid seem clever.");
006:         System.out.println("                - ^ -                ");
007:         System.out.println("                0 / 0                ");
008:         System.out.println("                =                ");
009:         System.out.println("                (At least, to the dumb user!)                ");
```



```
010:  }  
011: }
```

Now **compile** the program, and if you have typed it correctly it should compile without errors. However, when you **run** the program you will get an error message. Record this in your logbook, along with your best attempt to explain the meaning and the cause of it.

Now fix the error and run the program again. Record any new error messages that appear, along with your explanation for them.

2.7 Section / task 2.7 Hello anyone

- Aim of example: To introduce the principle of making Java programs perform a variation of their task based on **command line arguments**, which can be accessed via an **index**. We also meet string **concatenation**.
- Coursework title: **FlatterMe**
- Coursework summary: Write a program to say how wonderful the user is.
- Question: Without looking at HelloAnyone, write a program called FlatterMe which flatters the person named as the first **command line argument**, three times. The first comment should start with the person's name, the second should end with it, and the third should have some text either side of the name. (Hint: you will need to use two **concatenation operators** for that.)

2.8 Section / task 2.8 Hello anyone with a logical error

- Aim of example: To introduce the principle of **logical errors**.
- Coursework title: **Birthday logical errors**
- Coursework summary: Take a given program that has **logical errors** in it, and get it working.
- Question: Carefully type in the following program *exactly*. It contains some deliberate mistakes – type them in anyway even if you spot them as you read the code.

```
001: public class Birthday  
002: {  
003:     public static void main(String[] args)  
004:     {  
005:         System.out.print("Name: + args[1] + , " + args[2] + ";");  
006:         System.out.println("Born: " + args[0]);  
007:     }  
008: }
```

Now **compile** the program, and if you have typed it correctly it should compile without errors. Next you should **run** the program with *three command line arguments* – the first should be your personal name, the second should be your surname or family name,

and the third should be your date of birth, e.g. 24/04/1959. It is *intended* to print a result like the following.

Console Input / Output
\$ java Birthday John Latham 24/04/1959
Name: Latham, John; Born: 24/04/1959
\$ _

Note the position of spaces and punctuation in the desired output. However you will see that the output you actually get is wrong! Record the errors in your logbook, along with your best attempt to explain the cause of them.

Now fix the errors and run the program again. Record any new errors that appear, along with your explanation for them.

During this process you should have learnt about a **method** which is similar to `System.out.println()`. Record what this method is called and what it does.

2.9 Section / task 2.9 Hello solar system, looking at the layout

- Aim of example: To begin to explore the decisions behind the way we lay out the **source code** for a program.
- Coursework title: **Limerick layout**
- Coursework summary: Take a given program and lay it out properly.
- Question: Carefully type in the following program *exactly*. It contains very poor layout – type it in anyway even if you can see how it should be laid out as you read the code.

```
001: public class Limerick{public static void main(String[] args){System.out.
002: println("There was a young user of Java");System.out.println(
003: "Whose coding was such a palava!");System.out.println(
004: "His layout was pooh!");System.out.println(
005: "So what did we do?");System.out.println(
006: "We told him to stick to making coffee!");}}
```

You should **compile** the program, and if you have typed it correctly it should compile and **run** without errors. Record all the instances of poor layout you can see, in your logbook.

Now fix the layout!

3 Chapter 3 Types, variables and expressions

3.2 Section / task 3.2 Age next year

- Aim of example: To introduce the concepts of **type**, **int**, **variable**, **expression** and **assignment statement**. We also find out how to convert a number to a string, and discover what it means for **data** to be **hard coded**.

- Coursework title: **Hard coded YearsBeforeRetirement**
- Coursework summary: Write a program to determine how many years *you* have before you retire!
- Question: Preferably without looking at AgeNextYear, write a program called YearsBeforeRetirement which has your age **hard coded** into it, along with the age you expect to retire at (probably 68 – although that may well change before you get there!). Your program will need two **variables** for these values. It should then compute the difference between them and store it in a third variable, for which you should choose an *appropriate* name. Finally, it should produce three lines of output, similar to the following.

Console Input / Output

```
$ java YearsBeforeRetirement
My age now is 51
I will retire at the age of 68
Years left working is 17
$ _
```

3.4 Section / task 3.4 Age next year with a command line argument

- Aim of example: To introduce the idea of converting a **command line argument** into an **int** and using the value in a program.
- Coursework title: **Command line YearsBeforeRetirement**
- Coursework summary: Write a program to determine how many years the user has before he or she retires.
- Question: Here you will write another version of YearsBeforeRetirement, which is similar to the fully **hard coded** version of the program, but takes the user's age as its first **command line argument**. The retirement age is still to be hard coded as 68.

Before implementing the program, you should design **test data** for various tests which do the following.

- Make the program behave sensibly.
- Make the program behave inappropriately (i.e. a silly input resulting in a silly output).
- Make the program crash.

You should record this data in your logbook along with what you expect from each test.

Then copy your previous version of the program and alter it to suit the new requirement. You will learn most if you try not to look at the latest version of AgeNextYear while you do this. Finally, **run** it with your pre-planned tests and record whether the outcome was as you expected.

3.5 Section / task 3.5 Finding the volume of a fish tank

- Aim of example: To reinforce the use of **command line arguments** and **expressions**, and introduce the idea of splitting up lines of code which are too long, whilst maintaining their readability. We also see that a **variable** can be given a value when it is declared.
- Coursework title: **FieldPerimeter**
- Coursework summary: Write a program to determine how much fence is needed to surround a rectangular field.
- Question: Here you will write a program called `FieldPerimeter` which takes the length and width of a field as its two **command line arguments**, and computes the length of fence needed to enclose the field. (The simplest way to compute this is $length + length + width + width$).

Before implementing the program, you should design **test data** for various tests which do the following.

- Make the program behave sensibly.
- Make the program behave inappropriately (i.e. a silly input resulting in a silly output).
- Make the program crash.

You should record this data in your logbook along with what you expect from each test.

You will learn most if you try not to look at `FishTankVolume` while writing your program. Afterwards, **run** it with your pre-planned tests and record whether the outcome was as you expected.

3.6 Section / task 3.6 Sum the first N numbers – incorrectly

- Aim of example: To introduce the principle of **operator precedence**, and have a program containing a **bug**.
- Coursework title: **FishTankMaterials**
- Coursework summary: Take a program with **bugs** in it, and fix them.
- Question: Suppose you want to build fish tanks. Each tank has five pieces of glass – two sides, a front, a back and a bottom. It also has twelve pieces of metal angle-strip to form the edges. Below is a program which computes the surface area and the length of the edges for a tank with dimensions given by three **command line arguments**.

Design some **test data** for the program, predicting what the surface area and edge length should be if the program worked correctly. For simplicity, you do not need to worry about missing or meaningless arguments for this program. Record your planned test data in your logbook.

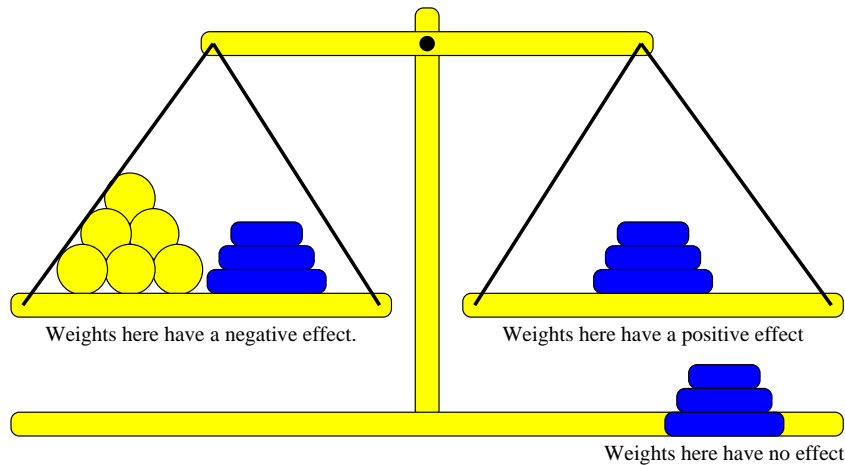
Then carefully type in the program, as is, and **run** it with your tests. Record the actual results, and attempt to explain the **bugs**, in your logbook.

```
001: public class FishTankMaterials
002: {
003:     public static void main(String[] args)
004:     {
005:         int width = Integer.parseInt(args[0]);
006:         int depth = Integer.parseInt(args[1]);
007:         int height = Integer.parseInt(args[2]);
008:
009:         int surfaceArea = width + height * depth + height + 2 * width + depth;
010:
011:         int edgesLength = height * width * depth + 4;
012:
013:         System.out.println("The surface area of a tank with dimensions "
014:                             + "(" + width + "," + depth + "," + height + ") "
015:                             + "is " + surfaceArea);
016:
017:         System.out.println("The length of the edges of a tank with dimensions "
018:                             + "(" + width + "," + depth + "," + height + ") "
019:                             + "is " + edgesLength);
020:     }
021: }
```

Now fix the program and test it again, making a record of your bug corrections. Does the program now produce the results you originally predicted, or is it still wrong, or were your original predictions wrong?

3.7 Section / task 3.7 Disposable income

- Aim of example: To introduce **operator associativity**. We also take a look at the **string literal escape sequences**.
- Coursework title: **ThreeWeights**
- Coursework summary: Write a program to show what weights can be weighed using a balance scale and three given weights.
- Question: In the days before accurate mechanical spring weighing scales (let alone digital ones), gold merchants were quite clever in their use of a small number of brass or lead weights, and a balance scale. (Indeed, many still use these in preference to inferior modern technology!) They would place the gold to be weighed in the left pan of the balance scale, and then place known weights in the right pan, and maybe also in the left pan, until the scales balanced. For example, suppose an unknown amount of gold was placed in the left pan. The merchant might experiment with a number of weights until he or she managed to make it balance with a known weight of R ounces in the right pan, and L ounces in the left pan along with the gold. This would show that the gold weighed $R - L$ ounces. In order to be able to weigh different amounts of gold, each merchant would carry a small number of known weights. When making a particular weighing, each weight could be placed in one of three positions: in the left pan with the gold, in the right one or not used.[?]



Suppose then that a merchant carries just three known brass or lead weights. Each of the three weights has three positions, thus giving rise to 3^3 combinations. These 27 combinations are listed below.

Position of				Position of				Position of			
	Wt 1	Wt 2	Wt 3		Wt 1	Wt 2	Wt 3		Wt 1	Wt 2	Wt 3
1	Left	Left	Left	10	Off	Left	Left	19	Right	Left	Left
2	Left	Left	Off	11	Off	Left	Off	20	Right	Left	Off
3	Left	Left	Right	12	Off	Left	Right	21	Right	Left	Right
4	Left	Off	Left	13	Off	Off	Left	22	Right	Off	Left
5	Left	Off	Off	14	Off	Off	Off	23	Right	Off	Off
6	Left	Off	Right	15	Off	Off	Right	24	Right	Off	Right
7	Left	Right	Left	16	Off	Right	Left	25	Right	Right	Left
8	Left	Right	Off	17	Off	Right	Off	26	Right	Right	Off
9	Left	Right	Right	18	Off	Right	Right	27	Right	Right	Right

Not all of these are useful. For example, at least one of them results in a weighing of zero – the one in which none of the weights are used. In fact, *any* combination in which the total of the known weights in the two pans is **equal**, results in a weighing of zero amount of gold. Then also, any combination for which the total known weight in the left pan is **greater than** that of the right pan is not useful – this would need a negative weight of gold in the left pan in order to balance!

Smart gold merchants chose the weights they would carry in such a way as to maximize their usefulness – that is, to enable the greatest range of weighings for a given number of carried weights. Suppose the number of weights carried is to be three. To maximize their effectiveness, there must be only one way of weighing zero, that is, the sum of any two weights must not equal the third. Also, of the 26 non-zero weighing combinations, for each that totals a positive weight, i.e., where the sum of the weights in the right pan exceeds that in the left, there is a corresponding negative weighing – formed by simply swapping the weights in the left and right pans over. This means there will be 13 combinations giving positive weighings, and 13 which give the opposite negative weighings. These negative weighings are of no use, and some of the positive weighings could add up to the same amount, which would not be efficient. The maximum effectiveness is achieved when the 13 positive weighings are the numbers 1 to 13.

You are going to write a program to help you experiment with this scenario, and by a

mixture of trial and error with your grasp of number theory, discover which three weight values gives the ability to weigh whole amounts from 1 to 13 inclusive.

Your program should be called `ThreeWeights`, and will take the three weights as **command line arguments**. It should then print out all 27 possible weighing values. Each value will appear on one line of the output. You will use 27 calls to `System.out.println()`.

Attempt to derive which three weights are the best to use, that is, the three values which produce weighing values from -13 through to 13 inclusive. If you are **running** in a Unix environment, and the first, or only, item on each output line is the weighing value, then the following command may help you assess your output.

Console Input / Output

```
$ java ThreeWeights 1 2 3 | sort -n
(Output shown using multiple columns to save space.)
-6      -4      -2      -1      0      1      2      3      4
-5      -3      -2      -1      0      1      2      3      5
-4      -3      -2      -1      0      1      2      4      6
$ _
```

This runs your program and pipes the output of it into the input of `sort`, for which the `-n` option means ‘**sort** numerically’.

On Microsoft Windows, the nearest equivalent is `java ThreeWeights 1 2 3 | sort` which sorts the lines lexicographically (alphabetically), rather than numerically – e.g. 10 comes before 2. Nevertheless, even this is helpful.

As you can see from the above output, there are duplicate values in the 27 listed, so 1 2 3 is not the right answer. (They don’t even add up to 13...) You will substitute the arguments you think the weights should be instead. If you are successful, attempt to explain why your values are the best three weights, in your logbook.



Coffee time: When weighing food, one would not wish to place weights in the pan containing the food, and so grocers did not use a three state, negative weighing scheme. What *four* weights did they use to weigh units of 1 to 15 inclusive? What is the connection between this and your answer for the gold merchants?

3.8 Section / task 3.8 Sum the first N numbers – correctly

- Aim of example: To introduce the fact that **integer division** produces a truncated result. We then look at the interaction between that and **operator associativity**.
- Coursework title: **RoundPennies**
- Coursework summary: Write a program to help a child determine whether she has enough pennies to go shopping!

- Question: Imagine there is a child who collects pennies in a piggy bank. Her mother tells her she is allowed to spend some of it when she has saved “about X pounds”, where X varies depending on what her mother thinks the girl is likely to want to buy. Your job is to write a program that helps the girl convert a number of pennies, which she is able to count up, into “about pounds” – i.e. round the number of pennies to the nearest pound. The program will take the number of pennies as its **command line argument**, and report how many pounds it rounds to. So, any non-negative number **less than** 50 will round to zero, but 50 through to 149 will round to 1. The value 749 rounds to 7, but 750 and 751 round to 8. And so on.

Start by designing **test data** and expected results in your logbook. You do not need to worry about arguments which are missing or are not **integer** numbers, but you should consider what the program will do for negative numbers, even though they are not really valid inputs.

Now write the program, calling it RoundPennies. You will learn most if you try to avoid looking at any other program while you do this. To get you thinking, the calculation will exploit the fact that **integer division** truncates its result. However that is not enough. For example, $750 / 100$ will yield 7, not 8 as we want here. There is some value you must add to the numerator before the **division** by 100.

Then, after implementing the program, you should **run** it with your pre-planned tests and record whether the outcome was as you expected. Record comments about the negative cases – is this the behaviour of a more general round-to-the-nearest-whole-number function? If not, what could we do to make it so?

3.9 Section / task 3.9 Temperature conversion

- Aim of example: To introduce the **double type** and some associated concepts, including converting to and from strings, and **double division**.
- Coursework title: **FahrenheitToCelsius**
- Coursework summary: Write a program to convert a temperature from Fahrenheit to Celsius.
- Question: In this task you will write a program called FahrenheitToCelsius which converts a given Fahrenheit temperature into its Celsius equivalent.

Start by designing **test data** and expected results in your logbook. You do not need to worry about arguments which are missing or not **real** numbers.

Now **design** your program. You can derive the formula by manipulating the one given for converting the other way. Show your working in your logbook. There is a temperature at which the Fahrenheit and Celsius measurements are the same. Figure out what this is, showing your working in your logbook, and add it to your tests.

As always, you will learn most if you try to avoid looking at any other program while writing this one. Afterwards, **run** it with your pre-planned tests and record whether the outcome was as you expected.

4 Chapter 4 Conditional execution

4.2 Section / task 4.2 Oldest spouse 1

- Aim of example: To introduce the idea of **conditional execution**, implemented by the **if else statement**, and controlled by **boolean expressions** based on the use of **relational operators**.
- Coursework title: **MaxTwoDoubles**
- Coursework summary: Write a program to find the maximum of two given numbers, using an **if else statement**.
- Question: In this task you will write a program called `MaxTwoDoubles` which takes two **command line arguments**, interprets them as **double** values, and reports both numbers along with which one of the two is the greatest, on the **standard output**. You will use an **if else statement**.

Start by designing **test data** and expected results in your logbook. You do not need to worry about arguments which are missing or are not **real** numbers.

Now **design** your program, preferably without looking at `OldestSpouse`. After implementing the program, you should **run** it with your pre-planned tests and record whether the outcome was as you expected.

4.3 Section / task 4.3 Oldest spouse 2

- Aim of example: To introduce the idea of nesting **if else statements**.
- Coursework title: **DegreeCategory**
- Coursework summary: Write a program to report the degree category of a given mark.
- Question: In this task you will write a program called `DegreeCategory` which takes a student mark (e.g. final year, total assessment mark) and reports what degree category it is worth. The input is a single number, which might have decimal places in it, entered as a **command line argument**.

Input	Required output
$input \geq 70$	Honours, first class
$70 > input \geq 60$	Honours, second class, division one
$60 > input \geq 50$	Honours, second class, division two
$50 > input \geq 40$	Honours, third class
$40 > input \geq 32$	Pass / ordinary degree
$input < 32$	Fail

Start by designing your **test data** and expected output in your logbook. You do not need to worry about input which is invalid. Then **design** and implement your program, preferably without looking at `OldestSpouse`. Finally, **run** it with your pre-planned tests and record whether your outcome was as you expected.

4.4 Section / task 4.4 Film certificate age checking

- Aim of example: To introduce the **if statement** without a **false part**.
- Coursework title: **PassFailDistinction**
- Coursework summary: Write a program to report the pass or fail status of an exam candidate, giving a message of distinction if appropriate using an **if statement**.
- Question: In this task you will write a program called `PassFailDistinction` which takes a postgraduate student mark and reports whether it is a pass or fail; and then, possibly, that it is a distinction. You will use an **if else statement** followed by an **if statement**. Here is the specification of the required output for a given input.

The input is a single number, which might have decimal places in it, entered as a **command line argument**.

Input	First line of output
$input \geq 50$	Pass
$input < 50$	Fail

Input	Second line of output
$input \geq 70$	Distinction
$input < 70$	(no second line)

Start by designing your **test data** and expected output in your logbook. You do not need to worry about input which is invalid. Then **design** and implement your program, preferably without looking at any others. Finally, **run** it with your pre-planned tests and record whether your outcome was as you expected.

5 Chapter 5 Repeated execution

5.2 Section / task 5.2 Minimum tank size

- Aim of example: To introduce the idea of **repeated execution**, implemented by the **while loop**. We also meet the notion of a **variable update**.
- Coursework title: **MinimumTankSize in half measures**
- Coursework summary: Write a program which calculates the minimum size of cubic tanks to hold given required volumes, where the possible sizes are in steps of 0.5 metre.
- Question: In this task you will write a program called `MinimumTankSize` which is the same as the one we have just covered, except that the tanks can be made with side lengths which are any positive whole multiple of 0.5 metre, instead of whole metres. (Hint: use **double** for the side length.)

Use the same **test data** as was used for the whole metres version of the program. Start by planning the expected output, in your logbook.

Then **design** and implement your program, preferably without looking at the previous version while you do this. When completed, **run** it with your pre-planned tests and record whether your outcome was as you expected.

Now change your program so that it has increments of 0.1 metres and test it again with the same data. Are there some surprises due to the accuracy of **real** numbers? Would you go so far as to say that some of them are wrong, rather than just inaccurate?

5.3 Section / task 5.3 Minimum bit width

- Aim of example: To introduce the idea of using **pseudo code** to help us **design** programs. We also meet `Math.pow()`.
- Coursework title: **LargestSquare**
- Coursework summary: Write a program to find the largest square number which is **less than or equal** to a given number.
- Question: A square number is a whole number which is the square of another (or the same) whole number. Examples are 0, 1, 4, 9, 25, 36, 49, 64, 81, 100, 121, 144, 169, etc.. In this task you will write a program called `LargestSquare` which takes a given positive **integer** as its **command line argument** and finds the largest square number which is **less than or equal** to that given number.

Start by planning your **test data** and expected results in your logbook. You do not need to worry about invalid inputs.

Now think about the **design** of your program. Perhaps the simplest approach to use is to focus on the square roots, rather than their squares. Start with a value which is **equal** to the given number, and keep decrementing it until its square is not **greater than** the number. So, for example, if the given number was 99, then we would start at 99 and count down until we finally get to 9 – this being the first number we find whose square is less than or equal to 99.

Express this **algorithm** in **pseudo code** in your logbook.

Finally, implement the program and test it with your test data, recording the results in your logbook.

Optional extra: Would it be quicker for the program to **loop** upwards from 0, rather than downwards from the given command line argument?

Optional extra: Look in the on-line Java documentation for the Java `Math` **class**, and find out how to obtain the square root of a number. Use this to speed up your program by making it start at a number which is much closer to the answer than the given command line argument is.

5.5 Section / task 5.5 Compound interest one

- Aim of example: To reinforce the **while loop** and the **compound statement**.

- Coursework title: **MinimumBitWidth by doubling**
- Coursework summary: Write a program to find the minimum **bit** width needed to support a given number of values, by doubling.
- Question: In this task you will write a variation of the `MinimumBitWidth` program which works a little more efficiently. Instead of computing a power of 2 in the **loop condition** on each **iteration**, your version will accumulate 2 to the power of `noOfBits` in a separate **variable**. This can be done by initializing your new variable to 1, and simply doubling its value each time you increment `noOfBits`.

You will use the same **test data** as used for the previous version of the program – except, do not try higher than 1073741824, otherwise your program will not end!

First think about the **design** of your program and plan in your logbook the changes you need to make to the original version.

Finally, implement the program and test it with your test data, recording the results in your logbook.

Optional extra: Explain why an input of, say, 1073741825 will cause a never ending **infinite loop**. Is there a solution?

5.6 Section / task 5.6 Compound interest two

- Aim of example: To introduce the **for loop**.
- Coursework title: **Power**
- Coursework summary: Write a program to raise a given number to the power of a second given number, without using `Math.pow()`.
- Question: What would you do if you needed to compute powers, and somebody had not already written the **method** `Math.pow()`? You would write the code yourself, and perhaps make it available for others to use.

In this task you will write a program, called `Power`, that takes two **integer** values as **command line arguments** and prints out the result of the first number raised to the power of the second. You may not use the `Math.pow()` method – somebody had to write that code, and let us pretend it is you! However, for simplicity, you may assume that both arguments exist and represent integers, and that the second number is non-negative.

Start by planning your **test data** and expected results in in your logbook.

Now think about the **design** of your program. One approach is to have a **variable** to accumulate the result, which starts off with the value 1. Then, using a **loop**, this result is multiplied by the first number as many times as the value of the second number. A **for loop** is appropriate for this task. Write **pseudo code** in your logbook.

Finally implement the program, (ideally without looking at `CompoundInterestKnownYears!`), test it with your preplanned tests and record the results in your logbook.

5.7 Section / task 5.7 Average of a list of numbers

- Aim of example: To show how to get the length of a **list**, note that an **index** can be a **variable**, and introduce **type casting**.
- Coursework title: **Variance**
- Coursework summary: Write a program to produce the variance of some given numbers.
- Question: In statistics, the variance of a **set** of numbers is one way of measuring the spread of them. It is the sum of the squares of the deviations (differences) between each number and the mean average of the numbers, all divided by the number of numbers.

For example, a set of student marks $\{2, 4, 6, 8, 10\}$ (out of 10) has a mean of 6 (which also happens to be one of the marks). The deviations from the mean are $\{-4, -2, 0, 2, 4\}$ and the squares of such are $\{16, 4, 0, 4, 16\}$. The variance is thus $(16 + 4 + 0 + 4 + 16)/5$, which is 8. Whereas, the results $\{4, 5, 6, 7, 8\}$ share the same mean but have a variance of only 2.

One approach to computing the variance is as follows. First compute the mean average of the numbers. Then, go through each number and compute the deviation between it and the mean, squaring this difference and accumulating the sum of all these squared deviations. Finally, divide that sum by the number of numbers.

In this task you will write a program, called `Variance` that takes a **list** of **integer** values as **command line arguments** and prints out the mean average and the variance of them. You may assume that there is at least one number, and that all the arguments represent integers.

Here is an example **run** of the program.

Console Input / Output
<pre>\$ java Variance 2 4 6 8 10 The mean average is 6.0 The variance is 8.0 \$ _</pre>

Start by planning your **test data** and expected results in in your logbook.

Now think about the **design** of your program. You can copy the code for computing the mean of the numbers from the example in this section. This will then be followed by a second **for loop** to compute the sum of the squares of the deviations between each number and the mean. You will need more **variables**, including one to hold the mean of the numbers, and another for the sum of the squares of the deviation between each number and the mean. Then the variance can be computed and output.

Write **pseudo code** in your logbook.

Finally, implement the program, test it with your preplanned tests and record the results in your logbook.

5.8 Section / task 5.8 Single times table

- Aim of example: To reinforce the **for loop**.
- Coursework title: **SinTable**
- Coursework summary: Write a program to produce a sin table.
- Question: In the days before scientific calculators, students of trigonometry used to use mathematical tables to look up values of **functions**, such as sin, cosin and tan.

In this task you will write a program, called SinTable to produce a sin table. It will take three **integer command line arguments**: the starting point of the table, the increment and the ending point. You can assume these arguments represent whole numbers of degrees. Here is an example **run**.

Console Input / Output
<pre>\$ java SinTable 0 10 90 ----- Sin table from 0 to 90 in steps of 10 ----- sin(0) = 0.0 sin(10) = 0.17364817766693033 sin(20) = 0.3420201433256687 sin(30) = 0.49999999999999994 sin(40) = 0.6427876096865393 sin(50) = 0.766044443118978 sin(60) = 0.8660254037844386 sin(70) = 0.9396926207859083 sin(80) = 0.984807753012208 sin(90) = 1.0 ----- \$ _</pre>

In Java, in order to compute the sin of a value, d , which is expressed in degrees, we can use the following **expression**.

```
Math.sin(Math.toRadians(d))
```

The **method** `sin()` is available in the standard **class** `Math`. It takes a value, expressed in radians, and **returns** the sin of that value. The method `toRadians()`, in the same class, converts a given value in degrees to the corresponding value in radians.

Start by planning your **test data** and expected results in in your logbook.

Now think about the **design** of your program. It should use a **for loop**. Write **pseudo code** in your logbook. You will learn most if you try not to look at `SinTable` while designing – perhaps you should compare the two programs after you have completed the task?

Finally, implement the program, test it with your preplanned tests and record the results in your logbook.

5.9 Section / task 5.9 Age history

- Aim of example: To introduce the idea of documenting programs using **comments**.
- Coursework title: **WorkFuture**
- Coursework summary: Write a program to print out all the years from the present day until the user retires.
- Question: In this task you will write a program, called `WorkFuture`, which shows the future working time of a user, assuming he or she retires at 68. The program will take two **command line arguments**, which you may assume are valid. The first is the present year, the second is the birth year of the user.

An example use of the program might be as follows.

Console Input / Output
<pre>\$ java WorkFuture 2010 1959 You have 17 years left to work In 2011 you will have 16 years left to work In 2012 you will have 15 years left to work In 2013 you will have 14 years left to work In 2014 you will have 13 years left to work In 2015 you will have 12 years left to work In 2016 you will have 11 years left to work In 2017 you will have 10 years left to work In 2018 you will have 9 years left to work In 2019 you will have 8 years left to work In 2020 you will have 7 years left to work In 2021 you will have 6 years left to work In 2022 you will have 5 years left to work In 2023 you will have 4 years left to work In 2024 you will have 3 years left to work In 2025 you will have 2 years left to work In 2026 you will have 1 years left to work You will retire in 2027 \$ _</pre>

Start by planning your **test data** and expected results in your logbook. Next, **design** the program, writing **pseudo code** in your logbook. As is generally true, you will learn most if you can avoid referring to the associated example while you do this, and only compare the two programs when you have finished.

Finally, implement the program – including suitable **comments** in the text, and test it. Record your results in the usual way.

5.10 Section / task 5.10 Home cooked Pi

- Aim of example: To introduce various **shorthand operators** for **variable updates**, have another example where we reveal the **pseudo code design**, and meet `Math.abs()` and

Math.PI.

- Coursework title: **Shorthand operators**
- Coursework summary: Go through all the programs before this point to see where **shorthand operators** could have been used.
- Question: Now that you know about the **shorthand operators** for updating **variables**, in this task you will go through all the examples in this chapter and identify all the places where they could have been used, recording your analysis in your logbook.

Optional extra: Take the program from this section and try it with one more decimal place. Then try to improve it to extend its accuracy.

6 Chapter 6 Control statements nested in loops

6.2 Section / task 6.2 Film certificate age checking the whole queue

- Aim of example: To introduce the ideas of nesting an **if statement** within a **for loop**, and declaring a **variable** inside a **compound statement**. We also introduce the **conditional operator**.
- Coursework title: **MaxList**
- Coursework summary: Write a program to find the maximum of a given **list** of numbers.
- Question: In this task you will write a program, called **MaxList**, which finds the maximum of a given **list** of numbers. The numbers are supplied as **command line arguments**. The program should report the number together with its **index** in the list (counting from zero). If two or more are jointly the maximum, it should report the one with the lowest index.

You may assume that the arguments all represent valid **double** numbers.

To find the maximum of a list of numbers, your program can start by regarding the first number as the maximum found so far, and then **looping** through the remaining numbers, comparing each with the maximum found so far and updating it as necessary.

Take the usual steps of planning **test data** and expected results, and **designing pseudo code** in your logbook, before implementing the program, including suitable **comments**, and recording your results back in your logbook.

6.3 Section / task 6.3 Dividing a cake (GCD)

- Aim of example: To introduce the idea of nesting an **if else statement** within a **while loop**.
- Coursework title: **DivideCake3**

- Coursework summary: Write a program to compute the **greatest common divisor** of three numbers.
- Question: Suppose the mother has three daughters who share their birthday. In this task you will write a program, called `DivideCake3`, which finds the greatest common divisor of the three ages given as **command line arguments** and reports the number of portions the cake should be divided into, and the number of portions each girl should get.

You may assume that the arguments all represent positive **int** numbers.

To find the **greatest common divisor** of three numbers, your program can find the greatest common divisor of two of them, and then find the greatest common divisor of that result and the third one.

Take the usual steps of planning **test data** and expected results, and **designing pseudo code** in your logbook, before implementing the program, including suitable **comments**, and recording your results back in your logbook.

6.4 Section / task 6.4 Printing a rectangle

- Aim of example: To introduce the idea of nesting a **for loop** within a **for loop**. We also meet `System.out.print()` and revisit `System.out.println()`.
- Coursework title: **PrintHoledRectangle**
- Coursework summary: Write a program to print out a rectangle with a hole in it.
- Question: In this task you will write a program, called `PrintHoledRectangle`, which prints a rectangle with a hole at the centre. This just means missing out one cell, printing spaces for it instead. The program takes the width and height arguments as before, but in order to ensure there is a centre cell, each of these have one added to them if necessary to make them an odd number.

You may assume that the arguments represent positive **int** numbers.

To ensure an **integer** number is odd you can simply divide it by two, multiply it by two and then add one! The simplest way to miss out the centre cell is to count all the cells as you print them, and check the sequence number of a cell just before you print it. The centre cell will have a sequence number which is the width times the height, divided by two, plus one.

Take the usual steps of planning **test data** and expected results, and **designing pseudo code** in your logbook, before implementing the program, including suitable **comments**, and recording your results back in your logbook.

6.5 Section / task 6.5 Printing a triangle

- Aim of example: To reinforce the idea of nesting a **for loop** within a **for loop**.
- Coursework title: **PrintTriangleMirror**

- Coursework summary: Write a program to print out an isosceles right angled triangle, with a straight right edge, and the longest side at the top.
- Question: In this task you will write a program, called `PrintTriangleMirror`, which prints an isosceles right angled triangle with its longest row at the top and the right hand side straight. The program is given the height as its argument – here is an example **run**.

Console Input / Output
<pre> \$ java PrintTriangleMirror 10 [][][][][][][][][][] [][][][][][][][][] [][][][][][][][][] [][][][][][][][] [][][][][][][] [][][][][][] [][][][][] [][][][] [][][] [][] [] \$ _ </pre>

You may assume that the argument represents a positive **int** number.

Each row will consist of a number of space cells (each 3 spaces) followed by a number of brick cells (" [_] "). This will require two **loops** inside the outer loop, one after the other.

Take the usual steps of planning **test data** and expected results, and **designing pseudo code** in your logbook, before implementing the program, including suitable **comments**, and recording your results back in your logbook.

6.6 Section / task 6.6 Multiple times table

- Aim of example: To reinforce the idea of having **nested statements** within each other, and explore the idea of using multiple **loops** in sequence.
- Coursework title: **CommonFactorsTable**
- Coursework summary: Write a program to produce a table showing pairs of numbers which share common factors.
- Question: In this task you will write a program, called `CommonFactorsTable`, which prints a 19 times 19 labelled table indicating which of all the pairs made up of **integers** between 2 and 20, inclusive, have common factors other than one. (That is, their **greatest common divisor** is **greater than** one.)

The program's output will be as follows.

Console Input / Output																				
\$ java CommonFactorsTable																				
		2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--
3	-- --	--#--	-- --	--#--	-- --	--#--	-- --	--#--	-- --	--#--	-- --	--#--	-- --	--#--	-- --	--#--	-- --	--#--	-- --	--#--
4	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--
5	-- --	-- --	-- --	--#--	-- --	-- --	-- --	--#--	-- --	-- --	-- --	--#--	-- --	-- --	-- --	--#--	-- --	-- --	-- --	--#--
6	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--
7	-- --	-- --	-- --	-- --	-- --	-- --	--#--	-- --	-- --	-- --	-- --	-- --	--#--	-- --	-- --	-- --	-- --	-- --	-- --	-- --
8	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--
9	-- --	--#--	-- --	-- --	--#--	-- --	-- --	--#--	-- --	-- --	-- --	--#--	-- --	-- --	-- --	--#--	-- --	-- --	-- --	--#--
10	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--
11	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	--#--	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --
12	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--
13	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	--#--	-- --	-- --	-- --	-- --	-- --	-- --	-- --
14	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--
15	-- --	--#--	-- --	--#--	-- --	-- --	--#--	-- --	-- --	-- --	--#--	-- --	-- --	-- --	-- --	--#--	-- --	-- --	-- --	--#--
16	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--
17	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	--#--	-- --	-- --
18	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--
19	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	-- --	--#--	-- --
20	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--	--#--
\$ _																				

A "--#" at the intersection of two numbers shows that their greatest common divisor is bigger than one, a "--|" shows otherwise.

This program may reasonably be developed by making changes to the TimesTable program. Plan these changes in your logbook, before taking the usual steps of implementing the program, including suitable **comments**, and recording your results back in your logbook.

6.7 Section / task 6.7 Luck is in the air: dice combinations

- Aim of example: To introduce the idea of using **nested loops** to generate combinations.
- Coursework title: **SumOfCubedDigits**
- Coursework summary: Write a program that determines which 3 digit decimal whole numbers are **equal** to the sum of the cubes of their digits.
- Question: There are four numbers in the range 100 to 999 which have the property that the sum of the cubes of the three digits in the number is **equal** to the number itself. 153 is such a number because $1^3 + 5^3 + 3^3$ is equal to $1 + 125 + 27$ which is 153. In this task you will write a program, called SumOfCubedDigits, that finds all four such numbers. [?]

Your program will work by **looping** through the numbers 100 to 999 using three **nested loops**, one for each digit. In the centre of the loops, your program can calculate the number represented by the three digits, and the sum of their cubes, and print out the number if these are equal.

Take the usual steps of planning **test data** and expected results, and **designing pseudo code** in your logbook, before implementing the program, including suitable **comments**, and recording your results back in your logbook.

7 Chapter 7 Additional control statements

8 Chapter 8 Separate methods and logical operators

8.2 Section / task 8.2 Age history with two people

- Aim of example: To further illustrate the inconvenience of having to copy a chunk of code which is used in different parts of a program, and thus motivate the need for separate **methods**.
- Coursework title: **WorkFuture2**
- Coursework summary: Write a program to print out all the years from the present day until retirement, for two people.
- Question: In this task you will write a program, called WorkFuture2, which shows the future working time of two people, assuming they retire at 68. The program will take three **command line arguments**, which you may assume are valid. The first is the present year, and the second and third are the birth years of the two people.

In case you have been reading ahead, you should *not* use a separate **method** – write all your code in the **main method**.

An example use of the program might be as follows.

Console Input / Output

```
$ java WorkFuture2 2010 1959 1992
Pn 1 has 17 years left to work
In 2011 pn 1 will have 16 years left to work
In 2012 pn 1 will have 15 years left to work
(... lines removed to save space.)
In 2026 pn 1 will have 1 years left to work
Pn 1 will retire in 2027
Pn 2 has 50 years left to work
In 2011 pn 2 will have 49 years left to work
In 2012 pn 2 will have 48 years left to work
(... lines removed to save space.)
In 2059 pn 2 will have 1 years left to work
Pn 2 will retire in 2060
$ _
```

Undertake the usual tasks of planning **test data**, **designing** the program, implementing and testing it, and finally recording your results.

8.3 Section / task 8.3 Age history with a separate method

- Aim of example: To introduce the idea of dividing a program into separate **methods** to enable the reuse of some parts of it. We meet the concepts **private**, **method parameter**, **method call** and **void method**.
- Coursework title: **WorkFuture4**
- Coursework summary: Write a program, with a separate **method**, to print out all the years from the present day until retirement, for four people.
- Question: In this task you will write another version of the work future program, called WorkFuture4, which shows the future working time of four people. The program will take five **command line arguments**, which you may assume are valid. The first is the present year, and the others are the birth years of the four people.

Your program should use a separate **method** to print the work future for one person, and call it four times.

Undertake the usual tasks of planning **test data**, **designing** the program, implementing and testing it, and finally recording your results.

8.4 Section / task 8.4 Dividing a cake with a separate method for GCD

- Aim of example: To introduce the idea of using **methods** merely to split the program into parts, making it easier to understand and develop. We also meet the **return statement** for use in **non-void methods**, and see that altering a **method parameter** does not change its argument.
- Coursework title: **DivideCake4**

- Coursework summary: Write a program to compute the **greatest common divisor** of *four* numbers, using a separate **method**.
- Question: In this task you will write a version of the cake dividing program for those very rare families that have four daughters sharing a birthday! This should be called `DivideCake4`. You may assume that the four arguments all represent positive **int** numbers. You should use a separate **method** to compute the **greatest common divisor** of two numbers.

Undertake the usual tasks of planning **test data**, **designing** the program, implementing and testing it, and finally recording your results.

8.5 Section / task 8.5 Multiple times table with separate methods

- Aim of example: To introduce the concept of **class variables**, compared with **local variables**, and reinforce the ideas of using separate **methods** for reuse and for dividing a program into manageable chunks. We also meet `System.out.printf()`.
- Coursework title: **CommonFactorsTable with methods**
- Coursework summary: Write a program, with separate **methods**, to produce a table showing pairs of numbers which share common factors.
- Question: In this task you will write a new version of the `CommonFactorsTable` program from Section 6.6 on page 26. This will use separate **methods** to avoid repeated code, and may reasonably be developed by making changes to the previous one. Plan these changes in your logbook, before taking the usual steps of implementing the program and recording your results.

8.6 Section / task 8.6 Age history with day and month

- Aim of example: To introduce the **logical operators**. We also see that a group of **variables** can be declared together.
- Coursework title: **Reasoning about conditions**
- Coursework summary: Do some reasoning to show that two different **conditions** have the same value.
- Question: Complete the following **truth table**, in your logbook, and thereby show that the **conditions**

`c1 = !(a1 < a2 || a1 == a2 && h1 <= h2)`

and `c2 = a1 > a2 || a1 == a2 && h1 > h2` are equivalent.

a1 < a2	a1 == a2	a1 > a2	h1 <= h2	h1 > h2	c1	c2
true	false	false	true	false		
true	false	false	false	true		
false	true	false	true	false		
false	true	false	false	true		
false	false	true	true	false		
false	false	true	false	true		

Optional extra: Try to show this by ‘simplifying’ from one condition to the other.

8.7 Section / task 8.7 Truth tables

- Aim of example: To introduce the **boolean type**, and reinforce **logical operators**. We also meet the **String type** and see that a **for update** can have multiple **statements**.
- Coursework title: **TruthTable34**
- Coursework summary: Write a program to test the equivalence of three **propositional expressions**, each having four **variables**.
- Question: In this task you will write another version of the **truth table** program, called **TruthTable34**, which shows a truth table for *three* **propositional expressions** which are **hard coded** as **methods** **p1**, **p2** and **p3** respectively, and which are expressions involving *four* propositional **variables**, **a**, **b**, **c** and **d**. Your table will thus have 16 lines plus titles and box lines, and 7 columns.

The three propositional expressions to hard code in your program are as follows.

p1	(((a b) && c) ((b c) && d)) && (a d)
p2	a && c b && d c && d
p3	(b c) && (c d) && (a d)

If you have studied discrete mathematics you should be able to spot the relationship between the first of these propositional expressions and the other two. What are those relationships?

Undertake the usual tasks of **designing** the program, implementing and testing it, and finally recording your results.

8.8 Section / task 8.8 Producing a calendar

- Aim of example: To reinforce much of the material presented in this chapter. We also revisit `System.out.printf()`.
- Coursework title: **CalendarHighlight**
- Coursework summary: Modify a calendar month printing program to produce a larger calendar format and to highlight a certain date.

- Question: In this task you will write another version of the calendar program, called CalendarHighlight, which produces the calendar in a wider format and also takes a third **command line argument**, which is a day (1 to 31) that should be highlighted. The wider format is produced by using four **characters** per date instead of two. The desired date should be highlighted by placing a **greater than** sign (>) before it and a **less than** sign (<) after it. Here is an example **run** of the finished program.

```

                                Console Input / Output
$ java CalendarHighlight 3 28 9
-----
| Su  Mo  Tu  We  Th  Fr  Sa |
|    01  02  03  04  05 |
| 06  07  08 >09< 10  11  12 |
| 13  14  15  16  17  18  19 |
| 20  21  22  23  24  25  26 |
| 27  28 |
|-----|
$ _

```

Undertake the usual tasks of planning **test data**, **designing** the program, implementing and testing it, and finally recording your results.

9 Chapter 9 Consolidation of concepts so far

10 Chapter 10 Separate classes

10.2 Section / task 10.2 Age history with Date class

- Aim of example: To introduce the principle of using more than one **class** in a program, and in particular, the idea of using a class as a template for the **construction** of **objects**. We also introduce **instance variables**, **constructor methods**, creating **new** objects, the fact that a **class** is a **type** and the use of **references**.
- Coursework title: **AddQuadPoly**
- Coursework summary: Write a **class** to store quadratic polynomials, and a program that adds together two quadratic polynomials to form a third.
- Question: In this task you will create a **class** called QuadPoly which will be used to represent quadratic polynomials, such as $6x^2 + 4x + 2$. Don't worry if Maths is not your favourite subject – you're not going to do anything too Mathematical. The class will have three **double instance variables**, one for each of the three coefficients of a quadratic polynomial. These will be declared **public**. (If you have read ahead, then please do *not* yet make them **private**.) The class will also have a **constructor method**, which will

be passed the three coefficient values as its **method parameters**. (The variable in these polynomials will always be x , and so its name need not be stored.)

You will also write a program called `AddQuadPoly`. This will take six **command line arguments**, these being two triples of coefficients, each triple being the coefficients of one quadratic polynomial. It will create an **instance** of `QuadPoly` for each of the two given quadratic polynomials. It will then create a third instance, representing the **addition** of the two given polynomials. Finally it will print out a report showing the addition. The following is an example **run**.

Console Input / Output	
\$	java AddQuadPoly 6 4 2 3 2 1
Polynomial:	6.0x^2 + 4.0x + 2.0
added to:	3.0x^2 + 2.0x + 1.0
results in:	9.0x^2 + 6.0x + 3.0
\$	_

Note how the three polynomials are to be printed using ordinary text, with a **^ character** before the power instead of attempting to raise the 2 into a superscript, such as in $6.0x^2$. Each polynomial is printed as follows, where the three question marks in the format are replaced by the values of the three coefficients.

?x^2 + ?x + ?

If you are tempted to write the program without creating three instances of `QuadPoly` (because it would in fact be easier right now) then you are seriously missing the point!

Undertake the usual tasks of planning **test data**, **designing** the classes, implementing them, testing the program, and finally recording your results.

Optional extra: Extend the program so that it can add together any number of polynomials listed as command line arguments, displaying the intermediate resulting polynomials as it goes along. Make it be able to handle the cases of there being no arguments ($0x^2 + 0x + 0$), just one polynomial, and the erroneous cases of the number of arguments not being divisible by three!

10.3 Section / task 10.3 Improving the Date class: lessThan() and equals() methods

- Aim of example: To introduce the concept of **instance methods**. We also look at common misunderstandings about **variables** and **references**.
- Coursework title: **CompareQuadPoly**
- Coursework summary: Extend a **class** that stores quadratic polynomials, and write a program that compares the 'size' of two quadratic polynomials.
- Question: In this task you will copy the `QuadPoly` **class** from the previous task and extend it, by adding two **instance methods**. The first one will compare the instance of `QuadPoly` it belongs to with another one given as a **method parameter**, and **return true**

if and only if they are **equivalent**, i.e. they represent the same polynomial. The second instance method will also compare the QuadPoly **object** with another, but return **true** if and only if this one is **less than** the other one. For a quadratic polynomial, $a_1x^2 + b_1x + c_1$ to be less than another, $a_2x^2 + b_2x + c_2$ then a_1 must be **less than** a_2 , or if they are **equal**, then b_1 must be less than b_2 , or if they are also equal, then c_1 must be less than c_2 . (If you have read ahead, then please do *not* yet add a toString() instance method.)

You will also write a program called CompareQuadPoly. This will take six **command line arguments**, this being two triples of coefficients, each triple being the coefficients of one quadratic polynomial. It will create an **instance** of QuadPoly for each of the two given quadratic polynomials. It will then use the two instance methods to compare them, to determine if they are equivalent, or the first one is less than, or **greater than** the second, and report the results. The following are some example **runs**.

Console Input / Output

```
$ java CompareQuadPoly 1 2 3 2 3 1
The polynomial:      1.0x^2 + 2.0x + 3.0
is smaller than:     2.0x^2 + 3.0x + 1.0
$ _
```

Console Input / Output

```
$ java CompareQuadPoly 3 2 1 3 2 1
The polynomial:      3.0x^2 + 2.0x + 1.0
is the same as:      3.0x^2 + 2.0x + 1.0
$ _
```

Console Input / Output

```
$ java CompareQuadPoly 3 2 1 1 2 3
The polynomial:      3.0x^2 + 2.0x + 1.0
is greater than:     1.0x^2 + 2.0x + 3.0
$ _
```

Undertake the usual tasks of planning **test data**, **designing** the classes, implementing them, testing the program, and finally recording your results.

Optional extra: Extend the program so that it can compare any number of polynomials listed as command line arguments, displaying the intermediate resulting polynomials as it compares each with the previous. At the end, it could report the smallest and largest polynomials encountered.

10.4 Section / task 10.4 Improving the Date class: toString() method

- Aim of example: To reinforce the concept of **instance methods**. We also note that a **method** might have no **method parameters**.
- Coursework title: **AddQuadPoly and CompareQuadPoly with toString()**
- Coursework summary: Extend a **class** that stores quadratic polynomials, and modify programs that add together, and compare the ‘size’ of, two quadratic polynomials.

- **Question:** In this task you will copy the `QuadPoly` **class** from the previous task and further extend it, by adding an **instance method** called `toString`. This will **return** a `String` representing the polynomial in the format previously introduced.

You will also copy the programs `AddQuadPoly` and `CompareQuadPoly` from the previous tasks, and modify them to make appropriate use of the new instance method.

Undertake the usual tasks of planning **test data** (consider if it will be different to the previous version of each program), **designing** the modifications, implementing them, testing the programs, and finally recording your results.

10.5 Section / task 10.5 Improving the Date class: addYear() method

- **Aim of example:** To further reinforce **instance methods**, meet Java's `toString()` convention and focus on the visibility of **instance variables**. We also see a **return type** which is a **class**.
- **Coursework title:** `QuadPoly` with an addition method
- **Coursework summary:** Further extend a **class** that stores quadratic polynomials, and modify a program that adds together two quadratic polynomials.
- **Question:** In this task you will copy the `QuadPoly` **class** from the previous task and make the **instance variables** have **private** visibility. You will also further extend it, by adding an **instance method** which takes a given other **instance** of `QuadPoly` as a **method parameter** and **returns** a **new** `QuadPoly` **object**, being the result of adding this `QuadPoly` instance to the given other one.

You will also copy the program `AddQuadPoly` from the previous task, and modify it to make appropriate use of the new instance method.

Undertake the usual tasks of planning **test data** (consider if it will be different to the previous version of the program), **designing** the modifications, implementing them, testing the program, and finally recording your results.

Optional extra: Add more instance methods to `QuadPoly` to subtract a given other polynomial, multiply this one by a constant and divide this one by a constant. Each of these will produce a new instance of `QuadPoly`. Then write a program called `QuadPolyCalculator` which permits arbitrary polynomial calculations. This will be based on the idea of an accumulator polynomial, which starts off as being $0x^2 + 0x + 0$. The **command line arguments** consist of a sequence of operation codes, each followed by an operand, which would either be a polynomial in the next three arguments, or a single number. For example, you might choose code 0 to represent addition, and this would be followed by three arguments representing a polynomial to be added to the accumulator. Code 2 might be multiplication, which would be followed by a single number to be multiplied by the accumulator. Each operation will produce output as it happens, and place its result back in the accumulator. At the end, the value of the accumulator will be reported. You could allow the **less than** operation to compare the accumulator with the following polynomial operand, and store whichever is the smallest in the accumulator. This would permit the program to be used to find the smallest of a sequence of polynomials, by preceding the

first one with the addition operation code, and the subsequent ones with the less than operation code! You might add a **greater than** operation code too.

Hint: this would be a good use of a **switch statement**.

11 Chapter 11 Object oriented design

11.2 Section / task 11.2 Age history revisited

- Aim of example: To introduce the principles of **object oriented design**. We also meet Scanner, **standard input**, Java's **package** structure and **import statement**, the **null reference**, **final variables**, multiple **return statements**, the **line separator system property**, and take a look at making **stubs of classes** and using **multi-line comments**.
- Coursework title: **ShapeShift**
- Coursework summary: Write a program to create and process two-dimensional shapes.
- Question: Here you will create a program called ShapeShift which does calculations and manipulations of simple shapes. The main class has been written for you – here it is.

```
001: import java.util.Scanner;
002:
003: /* This program performs simple calculations and manipulations of
004:    simple shapes expressed in two-dimensional coordinate geometry.
005:
006:    First it asks the user to choose a shape, from a choice of three.
007:    Then it prompts for details of the shape.
008:    *   A circle is specified by giving the X and then Y coordinate
009:        of its centre, followed by its radius.
010:    *   A Triangle is specified by giving the X and Y coordinates
011:        of each of its three corner points.
012:    *   A rectangle is specified by giving the X and Y coordinates
013:        of two of its diagonally opposite corner points.
014:
015:    Following this data, the user is prompted to specify an X offset
016:    and a Y offset.
017:
018:    The program creates the specified shape, and also a similar one,
019:    in which each point has been shifted by the X and Y offsets.
020:
021:    The program then reports the following on the standard output.
022:    *   The details of the original shape -- giving all the points
023:        (one, three, or four) and, for a circle, its radius.
024:    *   The area and perimeter of the shape.
025:    *   The details of the shifted shape.
026: */
027: public class ShapeShift
028: {
```

```
029: // A scanner to interact with the user.
030: private static Scanner inputScanner = new Scanner(System.in);
031:
032:
033: // Helper method to read a point from the input.
034: private static Point inputPoint(String prompt)
035: {
036:     System.out.print(prompt);
037:     double x = inputScanner.nextDouble();
038:     double y = inputScanner.nextDouble();
039:     return new Point(x, y);
040: } // inputPoint
041:
042:
043: // The X and Y amount to shift the first shape to get the second.
044: private static double xShift, yShift;
045:
046:
047: // Helper method to read the X and Y shifts.
048: private static void inputXYShifts()
049: {
050:     System.out.print("Enter the offset as X Y: ");
051:     xShift = inputScanner.nextDouble();
052:     yShift = inputScanner.nextDouble();
053: } // inputXYShifts
054:
055:
056: // The main method.
057: public static void main(String[] args)
058: {
059:     // Obtain shape choice.
060:     System.out.print("Choose circle (1), triangle (2), rectangle (3): ");
061:     int shapeChoice = inputScanner.nextInt();
062:
063:     // Process the shape based on the choice.
064:     switch (shapeChoice)
065:     {
066:         // Circle.
067:         case 1:
068:             Point centre = inputPoint("Enter the centre as X Y: ");
069:             System.out.print("Enter the radius: ");
070:             double radius = inputScanner.nextDouble();
071:             Circle originalCircle = new Circle(centre, radius);
072:             inputXYShifts();
073:             Circle shiftedCircle = originalCircle.shift(xShift, yShift);
074:             System.out.println();
075:             System.out.println(originalCircle);
076:             System.out.println("has area " + originalCircle.area()
077:                               + ", perimeter " + originalCircle.perimeter());
```

```

078:         System.out.println("and when shifted by X offset " + xShift
079:                               + " and Y offset " + yShift + ", gives");
080:         System.out.println(shiftedCircle);
081:         break;
082:
083:     // Triangle.
084:     case 2:
085:         Point pointA = inputPoint("Enter point A as X Y: ");
086:         Point pointB = inputPoint("Enter point B as X Y: ");
087:         Point pointC = inputPoint("Enter point C as X Y: ");
088:         Triangle originalTriangle = new Triangle(pointA, pointB, pointC);
089:         inputXYShifts();
090:         Triangle shiftedTriangle = originalTriangle.shift(xShift, yShift);
091:         System.out.println();
092:         System.out.println(originalTriangle);
093:         System.out.println("has area " + originalTriangle.area()
094:                               + ", perimeter " + originalTriangle.perimeter());
095:         System.out.println("and when shifted by X offset " + xShift
096:                               + " and Y offset " + yShift + ", gives");
097:         System.out.println(shiftedTriangle);
098:         break;
099:
100:    // Rectangle.
101:    case 3:
102:        Point diag1End1 = inputPoint("Enter one corner as X Y: ");
103:        Point diag1End2 = inputPoint("Enter opposite corner as X Y: ");
104:        Rectangle originalRectangle = new Rectangle(diag1End1, diag1End2);
105:        inputXYShifts();
106:        Rectangle shiftedRectangle = originalRectangle.shift(xShift, yShift);
107:        System.out.println();
108:        System.out.println(originalRectangle);
109:        System.out.println("has area " + originalRectangle.area()
110:                               + ", perimeter " + originalRectangle.perimeter());
111:        System.out.println("and when shifted by X offset " + xShift
112:                               + " and Y offset " + yShift + ", gives");
113:        System.out.println(shiftedRectangle);
114:        break;
115:
116:    // Bad choice.
117:    default:
118:        System.out.println("That wasn't 1, 2 or 3!");
119:        break;
120:    } // switch
121: } // main
122:
123: } // class ShapeShift

```

All you have to do is write the other classes.

The following are example **runs** of the program to help clarify the requirements.

Console Input / Output

```
$ java ShapeShift
Choose circle (1), triangle (2), rectangle (3): 1
Enter the centre as X Y: 0 0
Enter the radius: 1
Enter the offset as X Y: 2 2

Circle((0.0,0.0),1.0)
has area 3.141592653589793, perimeter 6.283185307179586
and when shifted by X offset 2.0 and Y offset 2.0, gives
Circle((2.0,2.0),1.0)
$ _
```

Console Input / Output

```
$ java ShapeShift
Choose circle (1), triangle (2), rectangle (3): 2
Enter point A as X Y: 0 0
Enter point B as X Y: 10 0
Enter point C as X Y: 0 20
Enter the offset as X Y: 5 10

Triangle((0.0,0.0),(10.0,0.0),(0.0,20.0))
has area 100.0, perimeter 52.3606797749979
and when shifted by X offset 5.0 and Y offset 10.0, gives
Triangle((5.0,10.0),(15.0,10.0),(5.0,30.0))
$ _
```

Console Input / Output

```
$ java ShapeShift
Choose circle (1), triangle (2), rectangle (3): 3
Enter one corner as X Y: 0 0
Enter opposite corner as X Y: 10 20
Enter the offset as X Y: 0 0

Rectangle((0.0,0.0),(10.0,0.0),(10.0,20.0),(0.0,20.0))
has area 200.0, perimeter 60.0
and when shifted by X offset 0.0 and Y offset 0.0, gives
Rectangle((0.0,0.0),(10.0,0.0),(10.0,20.0),(0.0,20.0))
$ _
```

Start by designing your **test data** in your logbook.

Your program will consist of five **classes**, Point, Circle, Triangle, Rectangle and the already given ShapeShift. Next identify and record the **public instance methods** and **class methods** for each of the four classes you will write. Endeavour to associate behaviour (i.e. **methods**) with the most appropriate classes. Here are some hints.

- Which classes should have a toString() instance method?
- Should shape classes have methods to find the area and perimeter of a shape?

- Should they additionally have a method to create a shifted shape from an existing one?
- Shifting shapes requires creating **new** points which are shifts of old ones. Where is that shifting best done?
- Perimeters of certain shapes are based on distances between points – does that suggest an instance method in the `Point` class?
- Are the points **mutable objects** or **immutable objects**? What about the shapes?
- All **instance variables** should be **private**, so you may need some instance methods in some classes, to give read access to the instance variables. For example, `Point` might have `getX()` and `getY()`.

Next you should write **stubs** for the three shape classes, so that you can **compile** and try out the main class.

Now **design** the implementations of your classes (at a level of **abstraction** that is appropriate to you) and then implement them. Do you want to think about the order of implementation so you can compile them as you proceed? Will you use a stub for `Point`?

Here are some implementation hints.

- To calculate the area of a triangle, you can use Hero's formula. Let a , b and c be the lengths of the sides of the triangle. Then the semi-perimeter, s is

$$s = (a + b + c) / 2$$

and the *area* is

$$area = \sqrt{s(s-a)(s-b)(s-c)}$$

- Given two opposite corners of a rectangle, i.e. both ends of one diagonal, $(x1, y1)$ and $(x2, y2)$ the other two corners are found as $(x1, y2)$ and $(x2, y1)$.

Finally record your results. It may well be that during your implementation, you changed your plan of which class should have what method. This is okay, but you should record such changes, and the reason for them.

Optional extra: Dare you consider having another shape, which is an irregular four sided polygon? Assuming the points are given in a sensible order, then computing the perimeter would not be too hard, but how would you get the area?

11.3 Section / task 11.3 Greedy children

- Aim of example: To reinforce **object oriented design**, particularly with **mutable objects**. We also meet multiple **constructor methods**, **class constants**, the **return statement** with no value, **accessor methods**, **mutator methods**, the dangers of **method parameters** which are **references**, converting the **null reference** to a string, and `Math.random()`.
- Coursework title: **StudentsCalling**

- Coursework summary: Write a program that simulates the behaviour of students using their mobile phones.
- Question: In this task you will create a program called `StudentsCalling` which simulates a simple scenario in which students purchase and use mobile phones.
 - A student has a name which cannot be changed, and a mobile phone, although not to begin with.
 - A phone has a name (i.e. make and model number) and an account, both of which are fixed. It also keeps track of the total number of seconds of phone calls made on it, starting with zero.
 - An account has a provider (i.e. the name of the service provider) which is fixed and a balance, in whole *pence*, which starts off as zero.
 - A student may purchase a mobile phone, in which case they discard their previous one if they have previously purchased one.
 - A student may top up their phone with a whole number of *pounds*. If they have no phone, then an attempt to top up their phone is ignored!
 - A student may make a call of desired duration, in seconds, on their phone. If they have no phone, then an attempt to make a call is ignored!
 - A phone may be topped up with a whole number of *pounds*, which simply causes its account to be topped up with that same amount.
 - A phone can have a call made on it, of a desired duration, which causes it to request that call on its account. The account **returns** the actual duration of the call, which may be **less than** that desired (i.e. when there is not enough balance to pay for it). The phone keeps track of the total actual duration of all the calls made on it.
 - An account may be topped up with a whole number of *pounds*. This adds to the current balance.
 - An account may have a call requested on it for a desired duration. In this wonderful world, all account providers charge only one penny per second for any call! The actual call duration will be limited to the current balance on the account. The balance is reduced by the actual duration. The actual duration is also returned as the result of the call request.
 - The main program will create some students, create some phones with accounts, which the students purchase, and cause the students to make calls. At each stage the behaviour of the program will be reported to the **standard output**.

The following is an example **run** of the program to help clarify the requirements.

Console Input / Output

```
$ java StudentsCalling
Creating student Chatty Charlie
Result:
Student(Chatty Charlie,null)

Creating student Norman No Friends
Result:
Student(Norman No Friends,null)

Creating student Popular Penny
Result:
Student(Popular Penny,null)

This next call has no effect, as has no phone!
Student(Chatty Charlie,null)
is making a call for desired 300 seconds
Result:
Student(Chatty Charlie,null)

This next top up has no effect, as has no phone!
Student(Norman No Friends,null)
is topping up by 20
Result:
Student(Norman No Friends,null)

Student(Chatty Charlie,null)
is buying phone Snotia BIFR
with account World@1
Result:
Student(Chatty Charlie,Phone(Snotia BIFR,0,Account(World@1,0)))
```

(Continued ...)

(...cont.)

```

Student(Norman No Friends,null)
is buying phone Cyoo L8TR0N
with account 4FRN Touch
Result:
Student(Norman No Friends,Phone(Cyoo L8TR0N,0,Account(4FRN Touch,0)))

Student(Popular Penny,null)
is buying phone Tisonly 14U
with account Foney Friends
Result:
Student(Popular Penny,Phone(Tisonly 14U,0,Account(Foney Friends,0)))

Student(Chatty Charlie,Phone(Snotia BIFR,0,Account(World@1,0)))
is topping up by 10
Result:
Student(Chatty Charlie,Phone(Snotia BIFR,0,Account(World@1,1000)))

Student(Norman No Friends,Phone(Cyoo L8TR0N,0,Account(4FRN Touch,0)))
is topping up by 20
Result:
Student(Norman No Friends,Phone(Cyoo L8TR0N,0,Account(4FRN Touch,2000)))

Student(Popular Penny,Phone(Tisonly 14U,0,Account(Foney Friends,0)))
is topping up by 30
Result:
Student(Popular Penny,Phone(Tisonly 14U,0,Account(Foney Friends,3000)))

Student(Chatty Charlie,Phone(Snotia BIFR,0,Account(World@1,1000)))
is making a call for desired 300 seconds
Result:
Student(Chatty Charlie,Phone(Snotia BIFR,300,Account(World@1,700)))

This next call should be truncated to 700 seconds.
Student(Chatty Charlie,Phone(Snotia BIFR,300,Account(World@1,700)))
is making a call for desired 1200 seconds
Result:
Student(Chatty Charlie,Phone(Snotia BIFR,1000,Account(World@1,0)))

Student(Chatty Charlie,Phone(Snotia BIFR,1000,Account(World@1,0)))
is making a call for desired 10 seconds
Result:
Student(Chatty Charlie,Phone(Snotia BIFR,1000,Account(World@1,0)))

```

(Continued ...)

(...cont.)

```

Student(Norman No Friends,Phone(Cyoo L8TR0N,0,Account(4FRN Touch,2000)))
is making a call for desired 10 seconds
Result:
Student(Norman No Friends,Phone(Cyoo L8TR0N,10,Account(4FRN Touch,1990)))

Student(Popular Penny,Phone(Tisonly 14U,0,Account(Foney Friends,3000)))
is making a call for desired 65 seconds
Result:
Student(Popular Penny,Phone(Tisonly 14U,65,Account(Foney Friends,2935)))

Student(Popular Penny,Phone(Tisonly 14U,65,Account(Foney Friends,2935)))
is making a call for desired 115 seconds
Result:
Student(Popular Penny,Phone(Tisonly 14U,180,Account(Foney Friends,2820)))

Student(Popular Penny,Phone(Tisonly 14U,180,Account(Foney Friends,2820)))
is making a call for desired 488 seconds
Result:
Student(Popular Penny,Phone(Tisonly 14U,668,Account(Foney Friends,2332)))

Student(Popular Penny,Phone(Tisonly 14U,668,Account(Foney Friends,2332)))
is making a call for desired 302 seconds
Result:
Student(Popular Penny,Phone(Tisonly 14U,970,Account(Foney Friends,2030)))

Student(Popular Penny,Phone(Tisonly 14U,970,Account(Foney Friends,2030)))
is making a call for desired 510 seconds
Result:
Student(Popular Penny,Phone(Tisonly 14U,1480,Account(Foney Friends,1520)))

Student(Popular Penny,Phone(Tisonly 14U,1480,Account(Foney Friends,1520)))
is making a call for desired 250 seconds
Result:
Student(Popular Penny,Phone(Tisonly 14U,1730,Account(Foney Friends,1270)))

Now let us discard a phone.
Student(Popular Penny,Phone(Tisonly 14U,1730,Account(Foney Friends,1270)))
is buying phone Simm UL8R
with account VerTuleTyat
Result:
Student(Popular Penny,Phone(Simm UL8R,0,Account(VerTuleTyat,0)))

$ _

```

Your program will consist of four **classes**, Student, Phone, Account and StudentsCalling. The latter will contain the **main method**.

Start by **designing** these classes in your logbook, identifying the **public instance methods**

and **class methods** for each of them. Endeavour to associate behaviour (i.e. **methods**) with the most appropriate classes.

Next you should design your ‘story’, that is, the sequence of operations you wish the simulation to undertake. You should make your ‘story’ significantly different to the example one above! That is, have different student names, phone names, account names, different number of students, different order and number of calls, etc..

Next design the implementations of your classes (at a level of **abstraction** that is appropriate to you). Then implement them. Do you want to think about the order of implementation so you can **compile** them as you proceed? Will you use **stubs**?

Here are some implementation hints.

- You can use the **null reference**, **null**, as the value for a student’s phone to begin with.
- The `toString()` method of `Student` can rely on the `toString()` method of `Phone` which in turn can use the `toString()` method of `Account`.
- Use **private** helper methods in the `StudentsCalling` class, to save you repeating code that prints out what is happening at each stage.

After implementation you should record your results. It may well be that during your implementation, you changed your plan of which class should have what method. This is okay, but you should record such changes, and the reason for them, in your logbook.

Optional extra: You can think of ways to make the simulation more realistic. For example:

- Suddenly there is a period of inflation again, and account providers have to charge more than one penny per second. Change your program so that an account has a rate, expressed in pence per minute.
- Perhaps rates vary depending on what time of day the call is made?
- Accounts ought to have a unique account number, assigned when they are created.
- Consider having a `Provider` class, so an account has a provider. Perhaps all the accounts for a particular provider have the same rate, but different providers have different rates.
- Now the providers are in competition again, perhaps it should be possible to change the account on an existing phone?

12 Chapter 12 Software reuse and the standard Java API

12.2 Section / task 12.2 A reusable Date class, with doc comments

- Aim of example: To explore the notion of **software reuse** and introduce **doc comments**. We also introduce the convention of having a `compareTo()` **instance method**.

- Coursework title: **StudentsCalling with doc comments**
- Coursework summary: Add **doc comments** to an existing **class**.
- Question: Copy your **classes** from the coursework in Section 11.3 on page 41 and add **doc comments** to the **public** items in them. Then run the javadoc program and examine the results. In particular, look at the summary sections and note how the first sentence of each doc comment has been used there.

12.5 Section / task 12.5 Simple Encryption

- Aim of example: To take a look at `String` manipulation, such as extracting individual **char** values from a `String`. We also look at how comparisons between two **char** values can be achieved, and the way we can **cast** between **char** and **int** values, and meet **overloaded methods**.
- Coursework title: **RomanNumber**
- Coursework summary: Write a **class** that allows for the conversion between decimal and Roman numbers.
- Question: In this task you will create a reusable **class** called `RomanNumber` which can be used to convert between Roman Numbers and decimal numbers.

You will provide *two* **constructor methods** for this class. One will take an **int** and build a `RomanNumber` corresponding to that number. The other will take a `String` of Roman digits and build a `RomanNumber` corresponding to that number.

The class will also provide two **instance methods**. One will **return** an **int**, being the decimal number corresponding to the `RomanNumber` **instance**. The other instance method will return a `String`, which is the Roman number representation of the `RomanNumber` instance.

For the purposes of this exercise, you may assume your constructors will never be given a non-positive number, or a `String` which is not a legal Roman number.

This class can be used to convert an **integer** to its Roman equivalent string by **constructing** an instance of `RomanNumber` from the integer, and then accessing the string value of it. To convert the other way, one could create an instance of `RomanNumber` from a string of Roman digits, and then access the integer value of it.

The rules of Roman numbers are explained below.

To help you choose names for the two instance methods, you should look at the **API** documentation of the `Integer` class. That class can be used to convert between **int** values and `String` representations in decimal, so it would be sensible to be consistent in style of names in your class.

In order to test your class, write a program called `RomanNumberTest`. This will accept a Roman number string from the first **command line argument**, convert it to an integer and then using a **loop**, print that number and the next 19 numbers, each with its Roman

number equivalent, on the **standard output**. The program may assume that the argument is a legal Roman Number. Here is an example **run**.

```

Console Input / Output

$ java RomanNumberTest MMX
(Output shown using multiple columns to save space.)
Roman for 2010 is MMX      Roman for 2020 is MMXX
Roman for 2011 is MMXI     Roman for 2021 is MMXXI
Roman for 2012 is MMXII    Roman for 2022 is MMXXII
Roman for 2013 is MMXIII   Roman for 2023 is MMXXIII
Roman for 2014 is MMXIV    Roman for 2024 is MMXXIV
Roman for 2015 is MMXV     Roman for 2025 is MMXXV
Roman for 2016 is MMXVI    Roman for 2026 is MMXXVI
Roman for 2017 is MMXVII   Roman for 2027 is MMXXVII
Roman for 2018 is MMXVIII  Roman for 2028 is MMXXVIII
Roman for 2019 is MMXIX    Roman for 2029 is MMXXIX
$ _

```

12.5.1 The Roman number system

In Roman numbers, there is no zero, nor any negative number. There are 7 digits and 6 pairs of digits, with values as follows.

Digit	Value
M	1000
D	500
C	100
L	50
X	10
V	5
I	1

Digit pair	Value
CM	900
CD	400
XC	90
XL	40
IX	9
IV	4

These are placed next to each other, with largest values on the left, and smallest on the right. The number represented is simply the sum of the values of the digits and digit pairs. The sample output from the test program (above) shows examples. Notice how each digit pair consists of a digit followed by a greater valued digit, and that the value is the value of the greater minus the value of the lesser. E.g. the value of "CM" is 1000 – 100. Perhaps contrary to your intuition, the Romans did not have other pairs than these 6. One cannot write "MIM" to mean 1999, instead it is written as "MCMXCIX": 1000 plus 900 plus 90 plus 9.

12.5.2 How to convert to and from Roman numbers

To convert a Roman number into an integer, we can scan the **characters** in the `String` from left to right and add the values of the characters to the **int** number being thus accumulated. So we start this accumulation with the value zero. However, if the value of any

character is **greater than** that of the previous one, then we have just had the second character of a digit pair. In this case we subtract the value previously added, twice, and then add the value of this character. You may wish to treat the first character of the Roman number `String` differently from the others, as it has no previous one. For all the other characters, we shall compare the value with the value of the previous character. Some examples follow.

Roman							Decimal
XIV	X	I	V				
	10	+1	-2 +5				14
CDXLIV	C	D	X	L	I	V	
	100	-200 +500	+10	-20 + 50	+1	-2 +5	444
CMXCIX	C	M	X	C	I	X	
	100	-200 +1000	+10	-20 +100	+1	-2 +10	999
MIM	M	I	M				
	1000	+1	-2 +1000				1999

Notice that the last line is an illegal Roman number string, yet the **algorithm** suggested will still produce a result, and effectively behaves as though "IM" actually is a legal digit pair with the value 999. As said above, you may assume your constructors are not given illegal strings, so there is no need for you to write code that checks legality.

Converting an integer into a Roman number is a little easier. We accumulate the sequence of Roman digits in a result `String`, starting with an empty string, as follows. While the number is **greater than or equal** to 1000, subtract a 1000 from it and append "M" to the result. Now do this for 900 with "CM", 500 with "D", 400 with "CD" and so on.

12.5.3 Implementation tips

You may find it easiest to have two **instance variables**, one an **int** and the other a `String`. Each constructor simply copies its given argument to one of the instance variables, and then calculates the value of the other. You should consider having **private methods** to assist in the conversions, and perhaps reduce the amount of repeated code.

12.5.4 Deliverables

First design your **test data** in your logbook, then **design pseudo code** for your two conversion algorithms, before implementing the classes. During implementation you should document your `RomanNumber` class with **doc comments**. After completing the test program, you should run the `javadoc` program and browse the resulting `index.html` file.

13 Chapter 13 Graphical user interfaces

13.2 Section / task 13.2 Hello world with a GUI

- Aim of example: To give a first introduction to Java **graphical user interface (GUI)** programs, in particular, the **classes** `JFrame`, `Container` and `JLabel`, together with the `java.awt` and `javax.swing` **packages** they belong to. We also talk about the idea of a class **extending** another class.
 - Coursework title: **HelloWorld GUI in French**
 - Coursework summary: Write a **GUI** program to greet the world, in French.
 - Question: In this task, you will take the HelloWorld **GUI** example and change it to greet the world in French (or some other language).
- Optional extra:** Make two greeting windows appear (with the same greeting).

13.3 Section / task 13.3 Hello solar system with a GUI

- Aim of example: To introduce the notion of **layout manager** and, in particular, `FlowLayout`.
- Coursework title: **HelloFamily GUI**
- Coursework summary: Write a **GUI** program to greet your family.
- Question: The coursework in Section 2.5 on page 8, asked you to produce a program called `HelloFamily` which greeted a number of your relatives. In this task you will write a version of that program which produces a window and greets the same relatives using labels. Each greeting should use a separate label. Use a `FlowLayout` **object** to manage the layout of the components in the window.

13.4 Section / task 13.4 Hello solar system with a GridLayout

- Aim of example: To introduce the **layout manager** called `GridLayout`.
- Coursework title: **HelloFamily GUI with GridLayout**
- Coursework summary: Write a **GUI** program to greet your family, using a `GridLayout`.
- Question: In this task, you will copy and change your `HelloFamily` program to use a `GridLayout`. Experiment with different values for the row and column **method parameters** in order to see how these effect the layout.

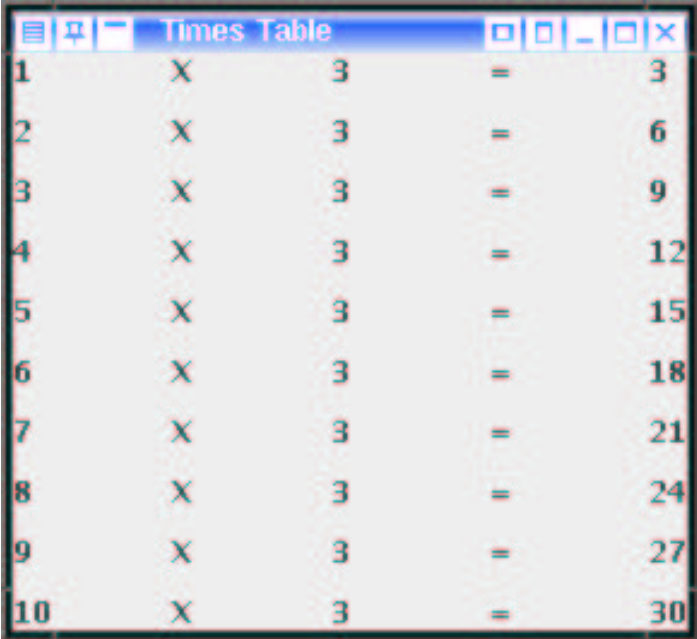
In order to make it easier to try out different values for the parameters, **design** the code so that the program takes two **integer command line arguments** – the values for the number of rows and number of columns. These will then be passed to the **constructor method** and used by it to create an appropriate `GridLayout` **object**.

Optional extra: Make your program produce 10 windows, each having a different gap between the components. The row gaps should range from 2 to 20 in steps of 2 pixels and the column gaps from 4 to 40 in steps of 4.

13.5 Section / task 13.5 Adding JLabels in a loop

- Aim of example: To illustrate the idea of creating **graphical user interface (GUI)** components in a **loop**.
- Coursework title: **TimesTable using JLabels**
- Coursework summary: Write a program to display a times table, using a **GUI** with **JLabel objects**.
- Question: In this task you will write a program, called TimesTable, which takes two **integer command line arguments**, m and n . It displays an m -times table with n entries, in a window. You can assume that m and n will be integers, and that n is non-negative. Choose better names for your **variables** than m and n ! Use **JLabel objects** to display the numbers and symbols and a **GridLayout** object to manage the layout. Choose horizontal and vertical gaps so that the window is laid out nicely.

For example, when given the arguments 3 and 10, we should see something like the following.



1	X	3	=	3
2	X	3	=	6
3	X	3	=	9
4	X	3	=	12
5	X	3	=	15
6	X	3	=	18
7	X	3	=	21
8	X	3	=	24
9	X	3	=	27
10	X	3	=	30

Optional extra: Find out how to set the colour of components, and choose a different colour to be used for alternating rows.

13.7 Section / task 13.7 Stop clock

- Aim of example: To reinforce the Java **listener** model together with JButton, ActionEvent and ActionListener. We also introduce the idea of having the ActionListener **object** be the JFrame itself, and meet System.currentTimeMillis().
- Coursework title: **StopClock with split time**
- Coursework summary: Modify a stop clock program so that it has a split time button.
- Question: In this task you will take the StopClock program and change it to add a split time button. Your program should still be called StopClock, and behave as follows.
 - The **GUI** has two buttons: Start/Stop and Split.
 - It has four output displays: the start time, stopped time, split time and elapsed time. Each of these is a JLabel and each also has a fixed JLabel to explain it.
 - The clock starts when the Start/Stop button is pressed. The current time is shown as the start time.
 - If the Split button is pressed while the clock is running, the clock will show the elapsed time as the split time.
 - If the Split button is pressed again while the clock is running, the split time will be updated.
 - The clock is stopped by pressing the Start/Stop button, at which point it will display the current time as the stopped time, and calculate, and display the elapsed time. The split time will be unchanged.
 - If the Split button is pressed while the clock is not running, nothing happens.

In order to implement this program, you will need to make use of the getSource() **instance method** of ActionEvent. This takes no **method arguments** and **returns a reference** to the **object** which was responsible for causing the **event**. So, for example, you may have code like the following.

```
if (event.getSource() == startStopJButton)
    ...
```

You will need to turn the **method variable** startStopJButton into an **instance variable**. Why is that?

Optional extra: Improve the GUI, from an end user's point of view, by removing the start and stop times: show just the status instead.

Optional extra: Extend the program to allow the recording of several split times, with a button for each split time.

Optional extra: Also, why not add a facility to pause and resume the clock?

13.8 Section / task 13.8 GCD with a GUI

- Aim of example: To introduce `JTextField`.
- Coursework title: **GCD GUI for three numbers**
- Coursework summary: Modify a GCD program that has a **GUI**, so that it finds the GCD of three numbers.
- Question: In this task you will produce a version of the GCD program with a **GUI**, that calculates the GCD of *three* numbers rather than two. This will require you to add an additional field to the interface, and alter the code of the **class** so that it calculates the appropriate value. As with the example in the section, the code for obtaining the GCD should reside in a separate `MyMath` class.

Optional extra: The GCD program requires that the user enters **integer** values. What happens if he or she supplies values that are not integers? How might you go about addressing this issue?

13.9 Section / task 13.9 Enabling and disabling components

- Aim of example: To explore the principle of enabling and disabling **graphical user interface (GUI)** components, and revisit `JButton` and `JTextField`.
- Coursework title: **StopClock using a text field and disabled split button**
- Coursework summary: Modify a stop clock program so that the split time button is disabled when the clock is not running.
- Question: In this task you will change your `StopClock` program as follows.
 - Have the start/stop button labelled `Start` when the clock is not running, and `Stop` when it is.
 - Disable the split button when the clock is not running, enable it when the clock is running.
 - Use `JTextField` **objects** rather than `JLabel` objects to display the times. Make it so that the end user cannot edit the text showing in these text fields.

Optional extra: Make the stop clock more pretty by using colours appropriately.

13.12 Section / task 13.12 Single times table with a ScrollPane

- Aim of example: To introduce the use of `JScrollPane` and revisit `JTextField`.
- Coursework title: **ThreeWeights GUI**
- Coursework summary: Write a **GUI** version of the program to show the weights that are obtainable on a balance scale using three weights.

-
- Question: In this task you will write a **GUI** version of the `ThreeWeights` coursework example from Section 3.7 on page 13. The program should offer the same functionality as the original one, that is, the user provides three weights and is then shown the possible values that can be weighed using them.

The user input should be through the use of text fields, and the results should be displayed in a scrollable text area.

Rather than have 27 `System.out.println()` calls as in the previous version of the exercise, a simpler way to compute the results is to use three **nested loops**, one for each weight. Each **loop variable** will be a multiplier for the corresponding weight, going through the values -1, 0 and 1. -1 represents placing that weight in the same pan as the gold, 0 represents not using that weight, and 1 represents placing that weight in the pan opposite the gold.

14 Chapter 14 Arrays

14.2 Section / task 14.2 Salary analysis

- Aim of example: To introduce the basic concepts of **arrays**, including **array type**, **array variables**, **array creation**, **array element access**, **array length** and **empty arrays**. We also meet `Math.round()` and revisit `System.out.printf()` and **division** by zero.
- Coursework title: **Mark analysis**
- Coursework summary: Write a program that analyses student coursework marks.
- Question: Write a program, called `MarkAnalysis`, that takes a **list** of student coursework marks and produces a report. The scores are entered by the user, after he or she has been prompted to say how many there are. Each score is a whole number **greater than or equal** to 0. The program should output the mean average, minimum and maximum of the scores, and a list of the scores, each along with their absolute difference from the mean average score, shown to two decimal places (using `System.out.printf()`).

In your **main method**, you should first read the scores into an **int array** using one **loop**, before finding the minimum, maximum and mean using a second, and then printing the results using a third. (You could combine the first two loops into one, but perhaps that would be less clear?)

You may assume that any input values are valid. However, if the number of scores is not at least one, your program should display a suitable message and exit.

Here is an example **run** of the program.

```

Console Input / Output

$ java MarkAnalysis
Enter the number of marks: 6
Enter mark # 1: 8
Enter mark # 2: 6
Enter mark # 3: 9
Enter mark # 4: 8
Enter mark # 5: 5
Enter mark # 6: 4

The mean mark is:      6.666666666666667
The minimum mark is:   4
The maximum mark is:   9

Person | Score | difference from mean
  1    |    8  |         1.33
  2    |    6  |        -0.67
  3    |    9  |         2.33
  4    |    8  |         1.33
  5    |    5  |        -1.67
  6    |    4  |        -2.67

$ _

```

Hint: Use the following **format specifier** string. "%6s | %5s | %6.2f%n"

14.3 Section / task 14.3 Sorted salary analysis

- Aim of example: To reinforce **arrays** and introduce the idea of **sorting**, together with one simple sorting **algorithm**. We also introduce the **for-each loop**, and have an array as a **method parameter** to a **method**.
- Coursework title: **Mark analysis with sorting**
- Coursework summary: Write a program that analyses student coursework marks, and presents the results in a **sorted** order.
- Question: Modify your program from the last task so that it presents the results in ascending order of mark. (Could this change the way you find your maximum and minimum?)

Here is an example **run** of the program.

```

Console Input / Output

$ java MarkAnalysis
Enter the number of marks: 6
Enter mark # 1: 8
Enter mark # 2: 6
Enter mark # 3: 9
Enter mark # 4: 8
Enter mark # 5: 5
Enter mark # 6: 4

The mean mark is:      6.666666666666667
The minimum mark is:   4
The maximum mark is:   9

Person | Score | difference from mean
  1    |    4  |      -2.67
  2    |    5  |      -1.67
  3    |    6  |      -0.67
  4    |    8  |       1.33
  5    |    8  |       1.33
  6    |    9  |       2.33

$ _

```

14.4 Section / task 14.4 Get a good job

- Aim of example: To examine **arrays** in which the **array elements** are **references to objects**. In particular, we see how this impacts on **sorting** with the use of a `compareTo()` **instance method**. We also revisit `System.out.printf()` and meet `String.format()`.
- Coursework title: **Mark analysis with student names and sorting**
- Coursework summary: Write a program that analyses named student coursework marks, and presents the results in a **sorted** order.
- Question: Modify your program from the last task so that each mark has an associated named student. You will need to create a **class** called `Student` with two **instance variables**, one for the name of a student and the other for his or her mark. This should provide a `compareTo()` **instance method** which you will use in your **sort** code, and a `toString()` to help produce the report.

Here is an example **run** of the program.

```

Console Input / Output

$ java MarkAnalysis
Enter the number of students: 6
Enter the name of student 1: Helen
Enter the mark for 'Helen': 8
Enter the name of student 2: Andy
Enter the mark for 'Andy': 6
Enter the name of student 3: John
Enter the mark for 'John': 9
Enter the name of student 4: Karen
Enter the mark for 'Karen': 8
Enter the name of student 5: Sanjay
Enter the mark for 'Sanjay': 5
Enter the name of student 6: George
Enter the mark for 'George': 4

The mean mark is:      6.666666666666667
The minimum mark is:   4
The maximum mark is:   9

Person and Score | difference from mean
George    got    4 | -2.67
Sanjay    got    5 | -1.67
Andy      got    6 | -0.67
Helen     got    8 |  1.33
Karen     got    8 |  1.33
John      got    9 |  2.33
$ _

```

You should make appropriate use of **for-each** loops. Hint: Use the following **format specifier** string. "%-10s got %3d"

14.5 Section / task 14.5 Sort out a job share?

- Aim of example: To introduce **partially filled arrays** with **array extension**, **array copying** to make a **shallow copy** and **returning an array** from a **method**. We also look at **object sharing** as we have three arrays containing **references** to the same **objects**. Along the way we meet the use of a **Scanner** on a **file**, **enum types** and **split()** on a **String**.
- Coursework title: **Random order text puzzle**
- Coursework summary: Write a random order text line **sorting** puzzle program.
- Question: In this coursework you will write a program that sets an **interactive** puzzle for the user to solve. The program is **run** with a **command line argument** which is the name of a **file** containing a few lines of text. These are read in and presented in a random order to the user, who is invited to pick one line to be swapped with the last one, repeatedly, until they are back in their original order.

The text might be part of the lyrics of a song, or a poem, or a quote, etc., or may have some other quality about it that gives a clue for working out the correct order.

Here is an example run of the program.

Console Input / Output	
\$	<code>java RandomOrderPuzzle test-data.txt</code>
0	are sorted as they started off,
1	it obvious
2	what the correct
3	Is
4	should be now that they
5	i.e. in order of increasing word count?
6	order of these lines
	Enter a line number to swap with the last one: 3
0	are sorted as they started off,
1	it obvious
2	what the correct
3	order of these lines
4	should be now that they
5	i.e. in order of increasing word count?
6	Is
	Enter a line number to swap with the last one: 0
0	Is
1	it obvious
2	what the correct
3	order of these lines
4	should be now that they
5	i.e. in order of increasing word count?
6	are sorted as they started off,
	Enter a line number to swap with the last one: 5
0	Is
1	it obvious
2	what the correct
3	order of these lines
4	should be now that they
5	are sorted as they started off,
6	i.e. in order of increasing word count?
	Game over in 3 moves.
\$	_

Write your solution in a **class** called `RandomOrderPuzzle`. The **main method** will create a `Scanner` for the file, and pass it to the **constructor method** to make an **instance** of `RandomOrderPuzzle`. Then it will make another `Scanner` for the **textual user interface**.

The constructor method will read in the text, and store it in an **array** of `Strings`, using

array extension as required. Then it will make a copy of this array into a second array, and randomize the order of this copy.

The class will also provide three **instance methods** for use in the main method. One will swap a given line of the copied array with its last line. Another will check to see whether the lines of the copy array are (now) in the same order as the original one. The third is a `toString()` which list the lines from the randomized copy in their current order.

Here is the main method, and a **private** instance method to randomize the order of a given array.

```
...
011: public static void main(String[] args) throws Exception
012: {
013:     Scanner fileScanner = new Scanner(new File(args[0]));
014:     RandomOrderPuzzle puzzle = new RandomOrderPuzzle(fileScanner);
015:
016:     Scanner inputScanner = new Scanner(System.in);
017:     System.out.println(puzzle);
018:     int moveCount = 0;
019:     while (! puzzle.isSorted())
020:     {
021:         System.out.print("Enter a line number to swap with the last one: ");
022:         puzzle.swapLine(inputScanner.nextInt());
023:         System.out.println(puzzle);
024:         moveCount++;
025:     } // while
026:     System.out.println("Game over in " + moveCount + " moves.");
027: } // main
...
084: private void randomizeStringArrayOrder(String[] anArray)
085: {
086:     for (int itemsRemaining = anArray.length;
087:         itemsRemaining > 0; itemsRemaining--)
088:     {
089:         int anIndex = (int) (Math.random() * itemsRemaining);
090:         String itemAtAnIndex = anArray[anIndex];
091:         anArray[anIndex] = anArray[anArray.length - 1];
092:         anArray[anArray.length - 1] = itemAtAnIndex;
093:     } // for
094: } // randomizeStringArrayOrder
...
```

14.6 Section / task 14.6 Diet monitoring

- Aim of example: To reinforce ideas met so far, and introduce **array initializer** and **array searching**, for which we revisit the **logical operators**.
- Coursework title: **Viewing phone call details**

- Coursework summary: Write a program to allow the user to view certain phone call details.
- Question: Here you will write a program that reads in a **file** of phone call details, and allows the user to see some of those calls with a total cost and duration. The first **command line argument** is the name of a **text file** containing the details of one phone call per line, comprising the phone number, including spaces at the appropriate places, the duration of the call, in the format hh:mm:ss, and the cost of the call, in pounds, as a decimal number. These three items are separated by single **tab characters**. Here is some sample **data**.

Console Input / Output			
\$ cat test-phone-calls.txt			
07571 78764	00:00:16	0.120	
01537 82608	00:00:04	0.070	
01492 88229	01:02:58	0.860	
08479 88844	00:03:56	0.070	
08901 24241	00:00:33	0.060	
07546 88323	00:02:40	0.250	
07571 78764	00:07:12	0.910	
08474 02751	00:05:37	0.150	
0161 296 410	00:03:02	0.190	
0161 296 682	00:00:57	0.090	
01537 82608	00:00:20	0.070	
01537 82608	00:30:10	0.450	
08479 77777	00:02:50	0.070	
07571 78764	00:06:23	0.800	
07728 50344	00:04:20	0.380	
0161 296 682	00:00:06	0.070	
07571 78764	00:44:28	2.930	
0161 803 487	00:15:59	0.260	
0161 297 617	00:13:24	0.530	
08476 05080	00:00:14	0.060	
08476 05080	00:04:09	0.130	
07571 78764	00:00:03	0.120	
0161 803 487	00:00:48	0.070	
08479 88844	00:01:05	0.060	
08901 27274	00:02:30	0.090	
07571 78764	00:08:18	0.630	
0161 297 629	00:01:05	0.120	
07936 84350	00:11:13	1.330	
07936 84350	00:01:59	0.270	
0161 297 629	00:00:01	0.090	
07571 78764	00:46:27	3.060	
08479 77777	00:03:17	0.070	
07955 65414	00:20:41	1.400	
01492 88229	01:24:12	0.850	
\$ _			

The user selects a subset of the calls by entering a prefix of the phone numbers he or she wishes to view. Here is an example **run**.

```

Console Input / Output

$ java PhoneCalls test-phone-calls.txt
Enter phone number prefix, or Q to quit: 075
07571 78764      00:00:16      0.12
07546 88323      00:02:40      0.25
07571 78764      00:07:12      0.91
07571 78764      00:06:23      0.80
07571 78764      00:44:28      2.93
07571 78764      00:00:03      0.12
07571 78764      00:08:18      0.63
07571 78764      00:46:27      3.06

Calls matched: 8
Total duration: 01:55:47
Total cost:      8.82

Enter phone number prefix, or Q to quit: 0161 2
0161 296 410     00:03:02      0.19
0161 296 682     00:00:57      0.09
0161 296 682     00:00:06      0.07
0161 297 617     00:13:24      0.53
0161 297 629     00:01:05      0.12
0161 297 629     00:00:01      0.09

Calls matched: 6
Total duration: 00:18:35
Total cost:      1.09

Enter phone number prefix, or Q to quit: 0161 8
0161 803 487     00:15:59      0.26
0161 803 487     00:00:48      0.07

Calls matched: 2
Total duration: 00:16:47
Total cost:      0.33

Enter phone number prefix, or Q to quit: Q
$ _

```

You should create four **classes**.

Class list for PhoneBook	
Class	Description
PhoneCalls	The main class containing the main method . It will make an instance of PhoneCallList and then prompt the user for input.
PhoneCallList	An instance of this will represent the list of phone calls and will contain instances of PhoneCall.

Class list for PhoneBook	
Class	Description
PhoneCall	An instance of this will represent a single phone call comprising phone number, duration and cost.
Duration	An instance of this represents a period of time which can be seen in hh:mm:ss format, and which can be added to another duration to yield a new one.

Here is the main method to get you started.

```

...
016:  public static void main(String[] args) throws Exception
017:  {
018:      callList = new PhoneCallList(new Scanner(new File(args[0])));
019:      Scanner inputScanner = new Scanner(System.in);
020:      String userInput;
021:      do
022:      {
023:          System.out.print("Enter phone number prefix, or Q to quit: ");
024:          userInput = inputScanner.nextLine();
025:          if (! userInput.equals("Q"))
026:              System.out.println(callList.matchingCallsReport(userInput));
027:      } while (! userInput.equals("Q"));
028:  } // main
...

```

You should think carefully where the logic to decide whether a particular phone call matches the user's input should go: is it to reside in PhoneCallList or PhoneCall? (Hint: is it about a phone call, or about a list?) Either way, you can use the `startsWith()` **instance method** of the `String` class.

To help you further, here is the code for the `Duration` class.

```

001: // Representation of a time duration.
002: public class Duration
003: {
004:     // Represented as a hh:mm:ss string and as total seconds.
005:     private final String stringRep;
006:     private final int totalSeconds;
007:
008:
009:     // Constructs from a hh:mm:ss string.
010:     public Duration(String requiredStringRep)
011:     {
012:         stringRep = requiredStringRep;
013:         String[] parts = requiredStringRep.split(":");
014:         int hours = Integer.parseInt(parts[0]);
015:         int minutes = Integer.parseInt(parts[1]);

```

```

016:     int seconds = Integer.parseInt(parts[2]);
017:     totalSeconds = (hours * 60 + minutes) * 60 + seconds;
018: } // Duration
019:
020:
021: // Constructs from a total number of seconds.
022: public Duration(int requiredNoOfSeconds)
023: {
024:     totalSeconds = requiredNoOfSeconds;
025:     int hours = totalSeconds / 3600;
026:     int minutes = (totalSeconds % 3600) / 60;
027:     int seconds = totalSeconds % 60;
028:     stringRep = String.format("%02d:%02d:%02d", hours, minutes, seconds);
029: } // Duration
030:
031:
032: // Returns the hh:mm:ss representation.
033: public String toString()
034: {
035:     return stringRep;
036: } // toString
037:
038:
039: // Adds this to another to create a new.
040: public Duration add(Duration other)
041: {
042:     return new Duration(totalSeconds + other.totalSeconds);
043: } // add
044:
045: } // class Duration

```

Optional extra: Instead of merely a leading prefix of phone numbers, why not allow the user to enter any pattern? (Hint: look at the `matches()` instance method of the `String` class.)

Optional extra: Add the date and time of calls to the program (and its data).

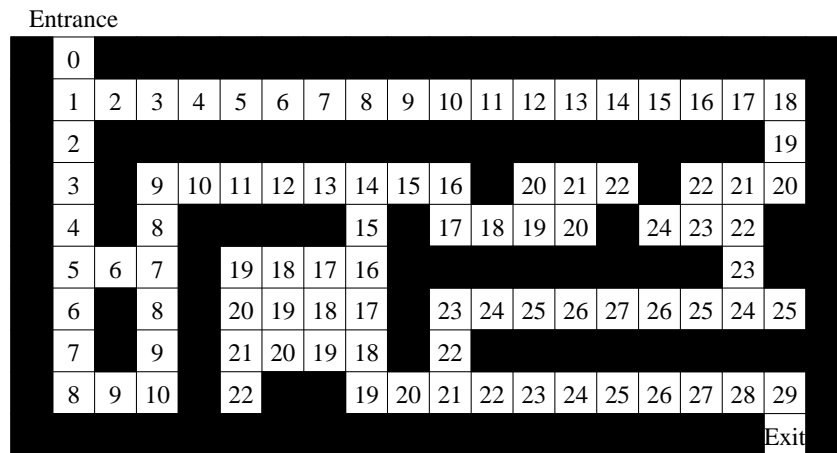
14.7 Section / task 14.7 A weekly diet

- Aim of example: To introduce **two-dimensional arrays**.
- Coursework title: **Maze solver**
- Coursework summary: Write a program that finds the shortest path through a maze.
- Question: The program you are going to write here will read in textual representations of mazes and solve them. Mazes consist of a matrix of cells, each of which can be an entrance, an exit, a hedge, or a space. Each maze must have at least one entry point, at least one exit point, and at least one path of space cells between some entrance and exit. Paths can only turn 90 degrees, that is, there is no use of diagonal movement. The job of

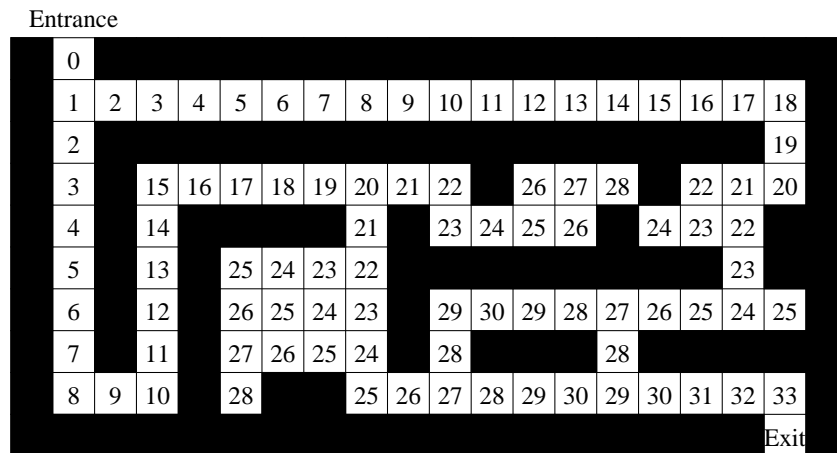
Here is sample **data**, showing three very similar mazes, each stored in a **text file**. A hedge cell is represented by a #, an entrance by a ?, an exit by a ! and a space by a space.

zero, those next to the entrances contain the number one, the neighbours of those contain a two, and so on.

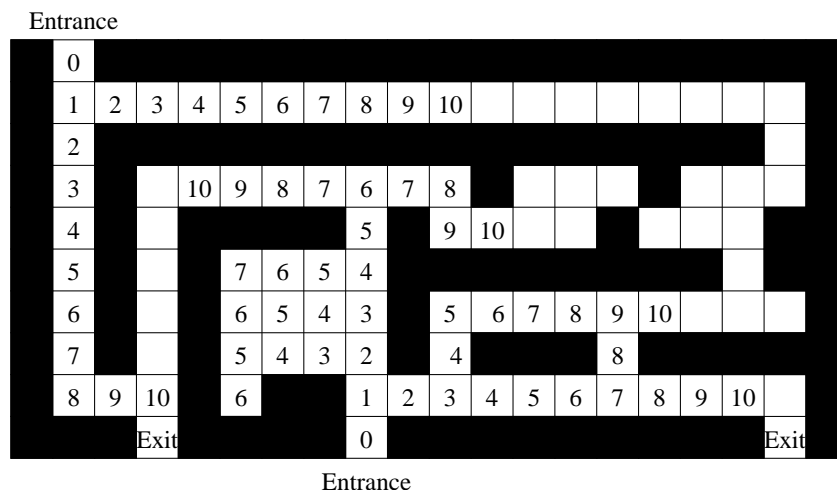
The following diagram shows this process for the first example maze above.



And this one shows it for the third example maze.



Note that, depending on the maze, it might be that not every space cell gets visited before an exit is found.



Having reached an exit, we stop fanning out and instead work backwards along the short-

est path to mark it.

Here is **pseudo code** for the **algorithm**.

```

move-count = 0
found-exit = false
while !found-exit
    consider every cell in turn
        if the cell value == move-count
            consider each of its four neighbours in turn or until found exit
                if the neighbour cell is an exit
                    found-exit = true
                    mark the path back to the start from this cell
                else if the neighbour cell is an unreached space
                    neighbour cell value = move-count + 1
            move-count++
    end-while

```

To mark the path back from the neighbour of the exit which has been reached, you do something like the following.

```

path position is given as row and column of the exit's neighbour
move-count = the value at this path position
while moveCount != 0
    mark this path position as part of the path
    move-count--
    find the neighbour which holds the value move-count
    path position = that neighbour's row and column

```

14.7.2 Implementation help

You will find the following code useful to help you **loop** through the four neighbours of each cell. (You may recall the **remainder operator**, %, from Section ?? on page ??.)

```

private int[] neighbourOffsets = {-1, 0, 1, 0};
...

for (int neighbour = 0; !foundAnExit && neighbour <= 3; neighbour++)
{
    int neighbourColumn = column + neighbourOffsets[neighbour];
    int neighbourRow = row + neighbourOffsets[(neighbour + 1) % 4];
    ...
} // for

```

Here is some of the solution to get you started.

```

001: import java.io.File;
002: import java.util.Scanner;
003:
004: /* Reads a maze representation from each file given as an argument
005:    and prints it out showing the shortest route from any entrance to an exit.

```

```

006: */
007: public class MazeSolver
008: {
009:     public static void main(String[] args) throws Exception
010:     {
011:         for (String filename : args)
012:             System.out.print(new MazeSolver(new Scanner(new File(filename))));
013:     } // main
014:
015:
016:     // The dimensions of the maze are fixed.
017:     private static final int HEIGHT = 10;
018:     private static final int WIDTH = 20;
019:
020:     // The values for cells in the maze model.
021:     // Start: this must be zero because you get there in zero steps.
022:     private static final int START = 0;
023:
024:     // Space, hedge, path and end: must all be negative
025:     // so they are not ambiguous with a move count.
026:     private static final int SPACE = -1;
027:     private static final int HEDGE = -2;
028:     private static final int PATH = -3;
029:     private static final int END = -4;
030:
031:     // The characters used in the file and output to represent the maze.
032:     private static final char SPACE_REP = ' ';
033:     private static final char HEDGE_REP = '#';
034:     private static final char START_REP = '?';
035:     private static final char END_REP = '!';
036:     private static final char PATH_REP = '.';
037:
038:     // The maze model. It is two bigger in each dimension so we can have an
039:     // extra hedge around the whole maze. This means every real cell has 4
040:     // neighbours, so we don't need to check edges of the array.
041:     private final int[][] maze = new int[HEIGHT + 2][WIDTH + 2];
042:
043:
044:     // Construct a MazeSolver from the given scanner for a file
045:     // which must contain HEIGHT lines each of WIDTH characters.
046:     public MazeSolver(Scanner input)
047:     {
048:         // First we place a surround of HEDGE cells.
049:         for (int row = 0; row < HEIGHT + 2; row++)
050:             maze[row][0] = maze[row][WIDTH + 1] = HEDGE;
051:         for (int column = 0; column < WIDTH + 2; column++)
052:             maze[0][column] = maze[HEIGHT + 1][column] = HEDGE;
053:
054:         // Next we read the maze, assuming the file is valid.

```

```

055:    // This goes in to positions 1 to HEIGHT and 1 to WIDTH
056:    // leaving the surrounding hedge unchanged.
057:    for (int row = 1; row <= HEIGHT; row++)
058:    {
059:        String mazeLine = input.nextLine();
060:        for (int column = 1; column <= WIDTH; column++)
061:        {
062:            char inputChar = mazeLine.charAt(column - 1);
063:            switch (inputChar)
064:            {
065:                case SPACE_REP: maze[row][column] = SPACE; break;
066:                case HEDGE_REP: maze[row][column] = HEDGE; break;
067:                case START_REP: maze[row][column] = START; break;
068:                case END_REP:  maze[row][column] = END;  break;
069:            } // switch
070:        } // for
071:    } // for
072:
073:    // Then we solve it.
074:    solve();
075: } // MazeSolver
076:
077:
078: // Each cell has four neighbours: these offsets help us find them.
079: private int[] neighbourOffsets = {-1, 0, 1, 0};
080:
081:
082: // Find the shortest path from any START to any END.
083: // There must exist such a path or else....
084: private void solve()
085: {
086:     ...
112: } // solve
113:
114:
115: // Mark the path backwards from row, column.
116: private void markPathBackFrom(int row, int column)
117: {
118:     ...
138: } // markPathBackFrom
139:
140:
141: // The correct line separator for this platform.
142: private static final String NLS = System.getProperty("line.separator");
143:
144:
145: // Return a text representation of the maze.
146: public String toString()
147: {

```

```

148:     String result = "";
149:     for (int row = 1; row <= HEIGHT; row++)
150:     {
151:         for (int column = 1; column <= WIDTH; column++)
152:             switch (maze[row][column])
153:             {
154:                 case HEDGE: result += HEDGE_REP; break;
155:                 case START: result += START_REP; break;
156:                 case END:   result += END_REP;   break;
157:                 case PATH:  result += PATH_REP;  break;
158:                 // Anything else will be a space which is not part of the path.
159:                 default:    result += SPACE_REP;
160:             } // switch
161:         result += NLS;
162:     } // for
163:     return result;
164: } // toString
165:
166: } // class MazeSolver

```

Note: your solution will probably have a different line count.



Coffee time: What happens if the program is run on a maze that does not have a path from an entrance to an exit?

Optional extra: Improve your program so that it deals sensibly with bad mazes.

Optional extra: Perhaps using `int` values for the cells is not a good **object oriented design** approach. So, push up your program by making it have a `Cell` **class**, which contains an **instance variable** for the cell type and a separate move count if it is a space cell. You could take this further, e.g. each cell could also contain its row and column **array index**, and an **array of references** to the four neighbouring cells to make looping through them even easier.

15 Chapter 15 Exceptions

15.2 Section / task 15.2 Age next year revisited

- Aim of example: To take a closer look at **run time errors**, or as Java calls them, **exceptions**.
- Coursework title: **FishTankVolume robustness analysis**
- Coursework summary: Take a program you have seen before and analyse where it can go wrong.
- Question: Take another look at the FishTankVolume program from Section ?? on page ?? . Make a list of all the circumstances that can cause an **exception** and another list of

circumstances which merely produce inappropriate results.

15.3 Section / task 15.3 Age next year with exception avoidance

- Aim of example: To show how we can avoid **exceptions** using **conditional execution**. We also meet the `Character` **class**.
- Coursework title: **FishTankVolume exception avoidance**
- Coursework summary: Take a program you have seen before and make it avoid **exceptions**.
- Question: Despite what we have just said about not being satisfied with the approach, you are here going to try it. Write a version of the `FishTankVolume` program from Section ?? on page ?? which avoids **exceptions**.

15.4 Section / task 15.4 Age next year with exception catching

- Aim of example: To introduce **exception catching** using the **try statement**. We also take a look at **standard error**.
- Coursework title: **FishTankVolume exception catching**
- Coursework summary: Take a program you have seen before and make it **catch exceptions**.
- Question: Write another version of the `FishTankVolume` program from Section ?? on page ??. This should not avoid **exceptions**, but instead **catch** them in a single **catch clause**. (In the next task you can improve it by having multiple catch clauses.)

15.5 Section / task 15.5 Age next year with multiple exception catching

- Aim of example: To observe that there are many kinds of **exception** and introduce the idea of multiple **exception catching** by having a **try statement** with many **catch clauses**.
- Coursework title: **FishTankVolume multiple exception catching**
- Coursework summary: Take a program you have seen before and make it **catch multiple exceptions**.
- Question: Write yet another version of the `FishTankVolume` program from Section ?? on page ??. This time it should **catch exceptions** in appropriate multiple **catch clauses**.

15.6 Section / task 15.6 Age next year throwing an exception

- Aim of example: To introduce the idea of creating an **exception** and **throwing** an exception when we have detected a problem, using the **throw statement**.

- Coursework title: **FishTankVolume** throwing exceptions
- Coursework summary: Take a program you have seen before and make it **throw** its own **exceptions** and **catch** them.
- Question: Write one more version of the FishTankVolume program from Section ?? on page ?. This will **throw exceptions** for inappropriate inputs which would otherwise not cause an exception, and **catch** all the exceptions in appropriate multiple **catch clauses**.

15.7 Section / task 15.7 Single times table with exception catching

- Aim of example: To illustrate the use of **exception catching** in **graphical user interface (GUI)** programs.
- Coursework title: **TimesTable** with a **ScrollPane** catching exceptions
- Coursework summary: Take a program with a **GUI**, that you have seen before, and make it **catch exceptions**.
- Question: Write a version of the TimesTable with a ScrollPane program from Section ?? on page ? that **catches** the **exception** caused by the user entering **data** which is not a valid representation of an **int**.

15.8 Section / task 15.8 A reusable Date class with exceptions

- Aim of example: To introduce the **throws clause** together with its associated **doc comment tag**. We also look at supplying an **exception cause** when we create an **exception**, and discuss the use of `RuntimeExceptions`.
- Coursework title: **Date** class with **nested try statements**
- Coursework summary: Modify a **class** so that it uses **nested try statements**.
- Question: Recall the *alternative* `addDay()` **instance method** from a coffee time on page ?. Modify the `Date` **class** from this section to make it use that alternative **nested try statements** approach for *all* instance methods which create a **new** `Date`.

Write a program called `TestRelativeDates` to test your implementation. This should contain a **main method** with hard-coded **test data**. One simple approach would be to create a 'reference' date, and then have a **loop** which takes it forwards one day at a time, over a, say, two year period (including a leap year). Inside the loop you print out the reference date together with five dates **constructed** relatively from it.

16 Chapter 16 Inheritance

16.3 Section / task 16.3 The Person class

- Aim of example: To introduce the ideas of **superclass**, **subclass**, **inheritance**, and **is a** relationships.
- Coursework title: **Stock control system**
- Coursework summary: Write a **class** that can be used to keep track of stock items, and test it.
- Question: Imagine you are setting up a computer parts shop, and will need **software** to keep track of stock and prices. You will have various kinds of stock item, but to start with you will implement a **class** called `StockItem` with the following properties. In later coursework tasks you will make various **subclasses** of this. An **instance** of `StockItem` represents a particular thing which the shop sells, with a fixed stock code, variable quantity in stock and variable price.

Public method interfaces for class <code>StockItem</code> .			
Method	Return	Arguments	Description
Constructor		<code>int, int</code>	Create a <code>StockItem</code> with the given int price (in whole pence) and int initial quantity in stock. The price is exclusive of VAT (sales tax). Each <code>StockItem</code> object is allocated a unique fixed int stock code.
<code>getStockCode</code>	<code>int</code>		Returns the stock code for this stock item.
<code>getStockType</code>	<code>String</code>		Returns the string "Stock item type". This will be redefined in subclasses.
<code>getDescription</code>	<code>String</code>		Returns the string "A description of the stock item". This will be redefined in subclasses.
<code>getQuantityInStock</code>	int		Returns the quantity in stock of this stock item.
<code>increaseStock</code>	void	int	Increases the stock level by the given amount. If it is less than one, an <code>IllegalArgumentException</code> is thrown with a suitable message.
<code>sellStock</code>	boolean	int	Attempts to reduce the stock level by the given amount. If it is less than one, an <code>IllegalArgumentException</code> is thrown with a suitable message. If the amount is otherwise less than or equal to the stock level, then the reduction is successful and true is returned . Else there is no effect, but false is returned.

Public method interfaces for class <code>StockItem</code> .			
Method	Return	Arguments	Description
<code>setPriceExVat</code>	<code>void</code>	<code>int</code>	Set the price of this item to the given <code>int</code> . This is the price before VAT.
<code>getPriceExVat</code>	<code>int</code>		Returns the price before VAT.
<code>getVatRate</code>	<code>double</code>		Returns the standard percentage VAT rate, which is currently 17.5. This may be re-defined in some subclasses.
<code>getPriceIncVat</code>	<code>int</code>		Returns the price including VAT (as specified by <code>getVatRate()</code>) rounded to the nearest penny.
<code>toString</code>	<code>String</code>		Returns a string giving the stock code, the stock type, the description, the quantity in stock, the price excluding VAT and the price including VAT. It uses the appropriate methods above to obtain the stock type, description, quantity and prices.

To allocate a unique fixed stock code to each `StockItem` **object** you might have the following code.

```

...
007:  // The number of stock items created so far.
008:  private static int noOfStockItemsCreated = 0;
009:
010:  // The fixed stock code of this item.
011:  private final int stockCode;
...
021:  public StockItem(int initialPriceExVat, int initialQuantityInStock)
022:  {
023:      noOfStockItemsCreated++;
024:      stockCode = noOfStockItemsCreated;
...
027:  } // StockItem
...

```

You will test this with a program called `TestStockItem`. This will make some instances of `StockItem`, increase stock, sell some stock and change the price, whilst printing out the items in between.

An example **run** might be as follows.

Console Input / Output

```

$ java TestStockItem
Creating a keyboard stock item, 10 in stock @ 499.
SC1: Stock item type, A description of the stock item (10 @ 499p/586p)
Creating a monitor stock item, 20 in stock @ 9999.
SC2: Stock item type, A description of the stock item (20 @ 9999p/11749p)
Obtain 10 more keyboards.
SC1: Stock item type, A description of the stock item (20 @ 499p/586p)
Obtain 20 more monitors.
SC2: Stock item type, A description of the stock item (40 @ 9999p/11749p)
Sell 5 keyboards.
SC1: Stock item type, A description of the stock item (15 @ 499p/586p)
Sell 10 monitors.
SC2: Stock item type, A description of the stock item (30 @ 9999p/11749p)
Change keyboard price to 399.
SC1: Stock item type, A description of the stock item (15 @ 399p/469p)
Change monitor price to 7999.
SC2: Stock item type, A description of the stock item (30 @ 7999p/9399p)
$ _

```

16.4 Section / task 16.4 The AudienceMember class

- Aim of example: To finish introducing **superclass**, **subclass** and **inheritance**, and briefly meet **UML**. Also, to introduce the principles of invoking the **constructor method** of the superclass, and having **instance methods** that **override** one from the superclass.
- Coursework title: **Your first stock item!**
- Coursework summary: Write a **subclass** which **overrides** some **instance methods**.
- Question: Your new computer parts shop has obtained a load of very cheap mouse mats, which are going to be your first item on sale. Create a **class** **MouseMat** which is a **subclass** of **StockItem**. This will **override** the **instance methods** `getStockType()` and `getDescription()` with ones that **return** "Mouse mat" and "Plain blue cloth, foam backed" respectively.

Test this with a program called `TestMouseMat` which makes an **instance** of **MouseMat** (you would probably not want more than one instance), increasing and then selling some stock and changing the price, whilst printing out the item in between.

16.5 Section / task 16.5 The Punter class

- Aim of example: To reinforce the ideas of **superclass**, **subclass**, **inheritance**, invoking the superclass **constructor method**, and **instance methods** that **override** another.
- Coursework title: **Your catalogue**
- Coursework summary: Write another **subclass** which **overrides** some **instance methods**.

- Question: Your mouse mats are selling like hot cakes and you dream of the days soon to come when you will sell other things too. In fact, you decide it is time to have a catalogue!

Create a **class** Catalogue which is a **subclass** of StockItem. This will **override** the **instance methods** getStockType() and getDescription() with ones that **return** "Catalogue" and "List of all items and prices" respectively.

Your new class will also override getVatRate() with one that **returns** zero, because books do not have VAT charged on them.

Test this with a program called TestCatalogue which makes an **instance** of Catalogue (you would probably not want more than one instance) increasing and then selling some stock and changing the price, whilst printing out the item in between.

16.6 Section / task 16.6 The Person abstract class

- Aim of example: To introduce the concepts of **abstract class** and **abstract method**.
- Coursework title: **An abstract stock item**
- Coursework summary: Make a **class** into an **abstract class**.
- Question: Alter your **class** called StockItem so that it becomes an **abstract class**. There are two **instance methods** which you should change to become **abstract methods**.

Confirm that you cannot make **instances** of StockItem by attempting to **compile** TestStockItem. Check that your other test programs **run** the same as they did before.

16.7 Section / task 16.7 The remaining simple subclasses of Person

- Aim of example: To reinforce the concepts covered in the chapter so far, and introduce the ideas of **polymorphism** and **dynamic method binding**. We also meet **final classes** and **final methods**.
- Coursework title: **More stock items**
- Coursework summary: Make some more **subclasses** and explore **polymorphism** and **dynamic method binding**.
- Question: You have obtained a big box of CPUs, a bin bag full of keyboards and a crate of hard discs. Create the **classes** CPU, Keyboard and HardDisc, **returning** the following values from getStockType() and getDescription().

Class	Result from getStockType()	Result from getDescription()
CPU	"CPU"	"Really fast"
Keyboard	"Keyboard"	"Cream, non-click"
HardDisc	"Hard disc"	"Lots of space"

Write a program called `TestStockItemSubclasses` which has a **class method** to test just one **instance** of a `StockItem` given to it as a **method parameter**. This will increase the stock, sell some stock and change the price, printing out the item in between.

The class will also have a **main method** which builds an **array** containing one instance of each **subclass** of `StockItem` you have written so far, and then, in a **loop**, calls the class method to test each one.

16.8 Section / task 16.8 The MoodyPerson classes

- Aim of example: To introduce the ideas of adding more **object state** and **instance methods** in a **subclass**, testing for an **instance** of a particular **class**, and **casting** to a subclass. We also see how a **constructor method** can invoke another from the same class.
- Coursework title: **Lots of different mouse mats!**
- Coursework summary: Have additional state in some **subclasses**.
- Question: Your shop is really beginning to take off – you now have several kinds of mouse mat! This causes you to think again about your `MouseMat` **class**. Only your first ones fit the description which you previously **hard coded**, and so you decide that a description suitable to particular mouse mats should be given when an **instance** of `MouseMat` is created. You realize that there may be other kinds of stock item that have similar simple variations in their descriptions, and so you decide to create another **abstract class** which is a **subclass** of `StockItem`, called `TextDescriptionStockItem`, and have `MouseMat` be a subclass of that. An **instance** of `TextDescriptionStockItem` will be given its description when it is created. Also, because you anticipate that descriptions might be refined due to customer feedback, they can be changed later.

Public method interfaces for class <code>TextDescriptionStockItem</code> .			
Method	Return	Arguments	Description
Constructor		<code>String, int, int</code>	Creates an instance of <code>TextDescriptionStockItem</code> with the given textual description, int initial price (in whole pence) and int initial quantity. The price is exclusive of VAT (sales tax).
<code>getDescription</code>	<code>String</code>		Returns the description that was given to the constructor method .
<code>setDescription</code>	void	<code>String</code>	Sets the description to the given string.

Read the rest of this task, and then draw a **UML class diagram** showing the full **inheritance hierarchy**, from `StockItem` downwards, as it will be when you have finished the task.

After drawing your diagram, implement the `TextDescriptionStockItem` class.

Now change `MouseMat` so that it is a **subclass** of `TextDescriptionStockItem` and remove `getDescription()` from it.

Alter `TestStockItemSubclasses` so that it makes two instances of `MouseMat` with different descriptions and prices. Add an extra **class method** that tests a `TextDescriptionStockItem` by making some change to its description (e.g. adding some text to it). Alter the existing class method that tests a `StockItem` so that, if the `StockItem` is also an instance of `TextDescriptionStockItem`, it will call your new class method to perform those additional tests.

You have also obtained various books about building computers that you would like to sell. Create the class `Book` which is another subclass of `TextDescriptionStockItem`. Remember that books are zero rated for VAT. Get rid of your `Catalogue` class – you have decided that your catalogue is better off being an instance of `Book`. Alter your test program so that it also creates some instances of `Book` – including one for your catalogue.

16.11 Section / task 16.11 The Game class

- Aim of example: To illustrate the difference between **is a** and **has a** relationships.
- Coursework title: **Shopping baskets**
- Coursework summary: Write a **class** each **instance** of which **has a** number of instances of another class stored in it.
- Question: As always, read the whole of this task and then plan in your logbook what classes you need, including what **methods** they will have, before starting your implementation.

Your computer parts shop has so many customers now that you wish to computerize the selling of your products. Write a **class** called `StockItemPurchaseRequest` which **has a** `StockItem` and an **int** quantity of that stock item required by a customer.

Write another class called `ShoppingBasket` which can contain any number of stock item purchase requests, using **array extension**. This should have an `add()` **instance method** which takes (a **reference** to) a `StockItem` and an **int** required quantity, and adds a corresponding `StockItemPurchaseRequest` to the shopping basket.

It will also have a `toString()` giving the contained stock item purchase requests, one per line.

And finally, it will have another instance method called `checkout`. This will go through the stock item purchase requests and sell them (reducing the stock quantities), if there are enough quantity in stock, or not otherwise. The successful purchase requests will be removed from the shopping basket, leaving only those that were not purchased. The result will be a `String` indicating for each purchase request whether it was purchased or not, along with the details of it, all followed by the total price with and without VAT. Test this with a program called `TestShoppingBasket`. Here is a sample implementation for that – feel free to alter it if you wish.

```

001: public class TestShoppingBasket
002: {
003:     public static void main(String[] args)
004:     {
005:         StockItem[] stockItems =
006:         {
007:             /* 0 */ new MouseMat("Plain blue cloth, foam backed", 150, 10),
008:             /* 1 */ new MouseMat("Pink vinyl with fluffy trim", 350, 10),
009:             /* 2 */ new Book("List of all items and prices", 150, 10),
010:             /* 3 */ new Book("Build a gaming monster", 1799, 0),
011:             /* 4 */ new CPU(1500, 10),
012:             /* 5 */ new HardDisc(5500, 10),
013:             /* 6 */ new Keyboard(200, 10)
014:         };
015:
016:         System.out.println("Stock before purchase:");
017:         for (StockItem stockItem : stockItems)
018:             System.out.println(stockItem);
019:         System.out.println();
020:
021:         ShoppingBasket shoppingBasket = new ShoppingBasket();
022:         shoppingBasket.add(stockItems[0], 2);
023:         shoppingBasket.add(stockItems[2], 1);
024:         shoppingBasket.add(stockItems[4], 8);
025:         shoppingBasket.add(stockItems[5], 9);
026:         shoppingBasket.add(stockItems[4], 3);
027:         shoppingBasket.add(stockItems[6], 8);
028:         shoppingBasket.add(stockItems[3], 1);
029:
030:         System.out.println("Shopping basket filled up:");
031:         System.out.println(shoppingBasket);
032:         System.out.println();
033:
034:         System.out.println("Performing Checkout:");
035:         System.out.println(shoppingBasket.checkout());
036:         System.out.println();
037:
038:         System.out.println("Shopping basket after checkout:");
039:         System.out.println(shoppingBasket);
040:         System.out.println();
041:
042:         System.out.println("Stock after checkout:");
043:         for (StockItem stockItem : stockItems)
044:             System.out.println(stockItem);
045:     } // main
046:
047: } // class TestShoppingBasket

```

Here is a sample **run** of the above code.

Console Input / Output

```
$ java TestShoppingBasket
```

```
Stock before purchase:
```

```
SC1: Mouse mat, Plain blue cloth, foam backed (10 @ 150p/176p)
```

```
SC2: Mouse mat, Pink vinyl with fluffy trim (10 @ 350p/411p)
```

```
SC3: Book, List of all items and prices (10 @ 150p/150p)
```

```
SC4: Book, Build a gaming monster (0 @ 1799p/1799p)
```

```
SC5: CPU, Really fast (10 @ 1500p/1763p)
```

```
SC6: Hard disc, Lots of space (10 @ 5500p/6463p)
```

```
SC7: Keyboard, Cream, non-click (10 @ 200p/235p)
```

```
Shopping basket filled up:
```

```
Shopping basket:
```

```
2 of SC1: Mouse mat, Plain blue cloth, foam backed (10 @ 150p/176p)
```

```
1 of SC3: Book, List of all items and prices (10 @ 150p/150p)
```

```
8 of SC5: CPU, Really fast (10 @ 1500p/1763p)
```

```
9 of SC6: Hard disc, Lots of space (10 @ 5500p/6463p)
```

```
3 of SC5: CPU, Really fast (10 @ 1500p/1763p)
```

```
8 of SC7: Keyboard, Cream, non-click (10 @ 200p/235p)
```

```
1 of SC4: Book, Build a gaming monster (0 @ 1799p/1799p)
```

```
Performing Checkout:
```

```
Checkout report:
```

```
Purchased 2 of SC1: Mouse mat, Plain blue cloth, foam backed (8 @ 150p/176p)
```

```
Purchased 1 of SC3: Book, List of all items and prices (9 @ 150p/150p)
```

```
Purchased 8 of SC5: CPU, Really fast (2 @ 1500p/1763p)
```

```
Purchased 9 of SC6: Hard disc, Lots of space (1 @ 5500p/6463p)
```

```
Not purchased 3 of SC5: CPU, Really fast (2 @ 1500p/1763p)
```

```
Purchased 8 of SC7: Keyboard, Cream, non-click (2 @ 200p/235p)
```

```
Not purchased 1 of SC4: Book, Build a gaming monster (0 @ 1799p/1799p)
```

```
Total price ex vat: 63550p
```

```
Total price inc vat: 74653p
```

```
Shopping basket after checkout:
```

```
Shopping basket:
```

```
3 of SC5: CPU, Really fast (2 @ 1500p/1763p)
```

```
1 of SC4: Book, Build a gaming monster (0 @ 1799p/1799p)
```

```
(Continued ...)
```

(...cont.)

```

Stock after checkout:
SC1: Mouse mat, Plain blue cloth, foam backed (8 @ 150p/176p)
SC2: Mouse mat, Pink vinyl with fluffy trim (10 @ 350p/411p)
SC3: Book, List of all items and prices (9 @ 150p/150p)
SC4: Book, Build a gaming monster (0 @ 1799p/1799p)
SC5: CPU, Really fast (2 @ 1500p/1763p)
SC6: Hard disc, Lots of space (1 @ 5500p/6463p)
SC7: Keyboard, Cream, non-click (2 @ 200p/235p)
$ _

```

Hint: in order to delete successfully purchased items from the ShoppingBasket, checkout() might create another (empty) ShoppingBasket into which it adds the StockItem and required quantity of *unsuccessful* requests. At the end it can copy the **instance variables** of this temporary ShoppingBasket to replace those of the original one.

As usual, record in your logbook any changes you needed to make to your plan.

16.12 Section / task 16.12 The Worker classes

- Aim of example: To show an example of a **superclass** which is (appropriately) not an **abstract class**. We also show how we can use an **instance method** defined in the superclass, from a **subclass** which **overrides** it.
- Coursework title: **Loads of disc space**
- Coursework summary: To write a non-**abstract class** which has a **subclass**, and use an **instance method** defined in the **superclass** from a subclass which **overrides** it.
- Question: Read the whole of this task, and then draw a **UML class diagram** showing the full **inheritance hierarchy**, from StockItem downwards, as it will be when you have finished the task.

Your shop just keeps getting better – now you have a whole variety of different sizes of hard disc on offer. Alter your HardDisc **class** so that the **constructor method** takes an additional **method parameter** which is the size of the disc in gigabytes. Alter the getDescription() **instance method** so that it **returns**, for example, "500GB of space" – the actual number being the given size, of course.

And then you get a delivery of an amazing new kind of hard disc that is so reliable it is guaranteed to keep **data** safe from disc crash for a specified number of years. Write a **subclass** of HardDisc called ReliableHardDisc. Its constructor method takes one extra parameter which is the guarantee period. It **overrides** getDescription() with one that appends, for example, ", guaranteed 20 years" to the string obtained by the same instance method in the **superclass** – the actual number being the given guarantee period, of course.

Alter your TestStockItemSubclasses class to include the size for the HardDiscs and also add at least one ReliableHardDisc.

16.13 Section / task 16.13 The CleverPunter class

- Aim of example: To reinforce **inheritance** concepts, and complete the model **classes** of the Notional Lottery program.
- Coursework title: **Making it more realistic**
- Coursework summary: Add more complexity to an **inheritance hierarchy** at appropriate places.
- Question: The computer parts shop example has been a little over simplified so far. In this task you will add more complexity to make it all a bit more realistic. You can add what you like, but here are some suggestions.
 - CPUs have a vendor, architecture and speed.
 - Hard discs have a physical size, vendor, rotational speed and cache/buffer size.
 - Keyboards have colour, vendor, number of keys, and possible special features description.
 - Perhaps every stock item could have a changeable part of its description, rather than just the TextDescriptionStockItem **class**.

Think of more ideas. Then identify the most appropriate place in the **inheritance hierarchy** to add each complexity, and implement them. You will add more **instance variables**, **instance methods** and alter existing instance methods as required. For example, `getDescription()` should perhaps incorporate the additional instance variables in its result.

16.15 Section / task 16.15 The Object class and constructor chaining

- Aim of example: To introduce the **class** Object and the fact that the **constructor method** of the **superclass** is invoked implicitly by default. We also take a more thorough look at **constructor chaining**.
- Coursework title: **Exploring constructor chaining**
- Coursework summary: Add tracing to existing **constructor methods** in order to explore **constructor chaining**.
- Question: Add `System.out.println()` calls to the **constructor method** of each **StockItem class** printing the name of the class. Add each call at the earliest point in the body of the constructor method that the **compiler** will let you. Once you have successfully **compiled** the classes, predict what the additional output will be from your `TestStockItemSubclasses` program *before* you **run** it. Then run it and see if you were right.

16.16 Section / task 16.16 Overloaded methods versus override

- Aim of example: To take a closer look at **overloaded methods** and in particular how an intended **override** can accidentally become an overload. We revisit the overloaded methods `System.out.println()`, and look at `toString()` from the `Object` **class**.
- Coursework title: **Using the @Override annotation**
- Coursework summary: Add to your **instance methods** that **override** another, an **annotation** which helps protect against errors.
- Question: Go through your solutions to the tasks in this Chapter and add the `@Override` **override annotation** to all **instance methods** which **override** another.

Also identify all the places where we should have put it in the example code.

17 Chapter 17 Making our own exceptions

17.3 Section / task 17.3 The Date class with its own exceptions

- Aim of example: To introduce the idea of making our own **exceptions**.
- Coursework title: **GreedyChildren with exceptions**
- Coursework summary: Add your own **exceptions** to the GreedyChildren example.
- Question: Copy the **classes** `GreedyChild` and `IceCreamParlour` from the example in Section 11.3 starting on page 40. Add two new classes `GreedyChildException` and `IceCreamParlourException`, both **subclasses** of `RuntimeException`. These should be able to handle causes, even though you might not need to use them at this stage.

Identify all the places in `GreedyChild` and `IceCreamParlour` where the **methods** might be given bad **method arguments**, and make them **throw** appropriate **exceptions**. (Hint: bear in mind that some of these arguments are **references**.) Recall that Java does not force you to have **throws clauses** for **unchecked exceptions**, but you nevertheless should do so for this task when such are possible.

Test the new features of each class using dedicated programs called `TestGreedyChildExceptions` and `TestIceCreamParlourExceptions` respectively. These should contain separate **try statements** for each possible exceptional situation.

17.4 Section / task 17.4 The Notional Lottery with exceptions

- Aim of example: To reinforce the idea of defining our own **exceptions**, and further it by having two of our own exception **classes**, where one is a **subclass** of the other.
- Coursework title: **MobileIceCreamParlour with exceptions**

- Coursework summary: Add a **subclass** of your own **exception** to the GreedyChildren example.
- Question: Now we have a **subclass** of IceCreamParlour representing what is, in effect, an ice cream van!

```
001: // An IceCreamParlour with the additional feature of needing to use fuel.
002: public class MobileIceCreamParlour extends IceCreamParlour
003: {
004:     // The amount of fuel left in the tank.
005:     private double fuelLeft = 0;
006:
007:
008:     // Construct a mobile ice cream parlour -- given the required name.
009:     public MobileIceCreamParlour(String name)
010:     {
011:         super(name);
012:     } // MobileIceCreamParlour
013:
014:
015:     // Put fuel in the tank.
016:     public void obtainFuel(double amount)
017:     {
018:         fuelLeft += amount;
019:     } // obtainFuel
020:
021:
022:     // Use some fuel by driving.
023:     public void drive(double desiredFuelUsed)
024:     {
025:         double fuelUsed = desiredFuelUsed <= fuelLeft ? desiredFuelUsed : fuelLeft;
026:         fuelLeft -= fuelUsed;
027:     } // drive
028:
029:
030:     // Return a String giving the name and state.
031:     @Override
032:     public String toString()
033:     {
034:         return super.toString() + "[fuel " + fuelLeft + "]\n";
035:     } // toString
036:
037: } // class MobileIceCreamParlour
```

Create a subclass of IceCreamParlourException called MobileIceCreamParlourException. Implement the above MobileIceCreamParlour **class**, but make it **throw** suitable **exceptions**. Test this using a program called TestMobileIceCreamParlourExceptions.

18 Chapter 18 Files

18.2 Section / task 18.2 Counting bytes from standard input

- Aim of example: To introduce the principle of reading **bytes** from **standard input** using `InputStream`, meet the **try finally statement** and see that an **assignment statement** is actually an **expression** – and can be used as such *when appropriate*. We also meet `IOException` and briefly talk about initial values of **variables**.
- Coursework title: **A check sum program**
- Coursework summary: Write a program to produce a **check sum** of the **standard input**.
- Question: The problem of being able to detect whether a **file** of **data** has changed since a previous version has many applications in computing. For example, if you download a file from the Internet, how can you be sure that your copy of it is correct and has not been corrupted? Or, imagine a program, **run** every night, that generates individual timetables for students, compares each of them with the timetable from the day before, and emails the latest copy if it has changed.

You might expect that the only way to see if a file has changed is to compare it **byte** by byte with the original, but this is not so. An alternative is to calculate some kind of **check sum** of the file and compare it with the number obtained from the original file. A check sum is a number that is a **function** of the file contents, computed in such a way that even a tiny change in the file causes a difference to the number. Perhaps the website could tell you what the number should be (as long as you use the same check sum **algorithm**). Similarly, the timetable program need remember only the check sum for each student from the night before.

In this task you will write a program called `Checksum` which reads all the bytes from **standard input** and outputs a single number on **standard output**. You should handle **exceptions** in the same way as we did for the example in this section. You will use the **BSD check sum**[?] algorithm which has been around for many years. There are more sophisticated and complex alternatives available nowadays, however this simple one is still fairly good.

For each **byte** in the input, the check sum computed so far is subjected to a **rotate right**, and then that byte is added to it. A rotate right means each **bit** of the number moves one place to the right, with the rightmost bit rotating to the leftmost place. For example, the 16-bit number 1100110011001100 becomes 0110011001100110, and 0011001100110011 becomes 1001100110011001.

The BSD algorithm computes a 16-bit check sum, which you will store in a 32-bit **int**. So you need to take care that the rotation is done on only the lower 16 bits and the upper 16 always remain zero. Here is the **algorithm** expressed in **pseudo code**.

```
int checksum = 0
for every byte from the input
    rotate checksum right by one bit, treating it as a 16 bit number.
    checksum += byte
    restrict checksum to 16 bits.
```

```
end-for
output checksum
```

To rotate `checksum` right by one bit, whilst treating it as a 16 bit number, you can use the following pseudo code. (Note that 32768 is 2^{15} .)

```
if checksum is even
    checksum /= 2
else
    checksum /= 2
    checksum += 32768
```

You may prefer to express 32768 in your Java code as a **hexadecimal integer literal**, in the form `0x8000`. (Also, you may prefer to find out about the bit **shift operators** and use one of those instead of **division**).

To restrict `checksum` to 16 bits you can use the following code, which works because you have just added a value **less than** 256 to a value that was less than 65536 (which is 2^{16}).

```
if checksum >= 65536
    checksum -= 65536
```

You may prefer to express 65536 as `0x10000`. (Also, you may prefer to find out about the **integer bitwise operators** and use one of those instead of an **if statement**.)

To perform a check sum of the data in a file, rather than input typed at the keyboard, you can redirect **standard input** to come from that file, using `<` on the **command line**. If you are using a Unix environment, you can probably test your program by comparing its output with that obtained from the `sum` command (which also outputs the size of the file as a number of one kilobyte blocks). Otherwise the book website has some example files for you to try, along with their correct check sums.

Console Input / Output

```
$ java CheckSum < CheckSum.java
51871
$ sum CheckSum.java
51871      2
$ _
```

(The check sum for *your* program code will probably not be the same as the one above.)

18.3 Section / task 18.3 Counting characters from standard input

- Aim of example: To introduce the principle of reading **characters**, instead of **bytes**, from **standard input**, using `InputStreamReader`.
- Coursework title: **Counting words**
- Coursework summary: Write a program to count the number of words in its **standard input**.

- Question: Write a program, `WordCount` which reads the **characters** from its **standard input**, counting how many words that contains, and reports the number on its **standard output**. You should handle **exceptions** in the same way as we did for the example in this section.

A character is either a **white space** character, such as **space character**, **tab character**, or **new line character**; or it is part of a word. To determine whether a `char c` is white space, you can use `Character.isWhitespace(c)`.

A word is a non-empty sequence of any non-white space characters, preceded either by the beginning of the **file**, or a white space character, and followed either by the end of the file, or a white space character. There may be more than one white space character before and/or after a word, including before the first word, and after the last one.

Hint: the start of a word is at a character which is itself not white space, and which is either the first character in the input, or was preceded by a white space character.

Alternatively, think of the input as being:

- A possibly empty sequence of white space characters.
- A possibly empty sequence of words, each being:
 - * A non-empty sequence of non-white space characters.
 - * A possibly empty sequence of white space characters.

As usual, design **test data** in advance of **designing** your program.

18.4 Section / task 18.4 Numbering lines from standard input

- Aim of example: To introduce the principle of reading lines from **standard input**, using `BufferedReader`.
- Coursework title: **Deleting a field**
- Coursework summary: Write a program to delete a field in tab separated text from the **standard input**.
- Question: Write a program called `DeleteField` which copies its **standard input** to its **standard output**, line by line, except that it deletes one of the fields on each line. The fields are separated by a single **tab character**, and are numbered from one upwards. The number of the field to be deleted is given as a **command line argument**.

Here is an example **run**.

Console Input / Output				
\$ java DeleteField 2				
Name	Coursework	Exam	Total	
Name	Exam	Total		
Fred Bloggs	55	65	60	
Fred Bloggs	65	60		
Susan Smart	100	90	95	
Susan Smart	90	95		
^D				

(Of course, in practice the program would be most useful if the input was being redirected from a **file**, rather than literally being typed in line by line – the above is really just for testing.)

You might find the following code helpful.

```
...
024:         // Divide the line into fields using tab as a delimiter.
025:         String[] fields = inputLine.split("\t");
026:         String editedLine = "";
027:         if (fields.length < fieldToDelete)
028:             editedLine = inputLine;
029:         else
030:         {
031:             // We build the new line in parts.
032:             // Add the fields before the one to be deleted.
033:             for (int index = 0; index < fieldToDelete - 1; index++)
034:                 if (editedLine.equals("")) editedLine = fields[index];
035:             else editedLine += "\t" + fields[index];
036:             // Add the fields after the one to be deleted.
037:             for (int index = fieldToDelete; index < fields.length; index++)
038:                 if (editedLine.equals("")) editedLine = fields[index];
039:             else editedLine += "\t" + fields[index];
040:         } // else
...
```

You should handle **exceptions** in the same way as we did for the example in this section (except you will need to consider problems relating to the command line argument).

If you wanted to delete two fields, and also your **data** was in a **text file**, then you could redirect the standard input to come from it, and pipe the **standard output** into the input of another run of your program.

Console Input / Output				
\$ cat input.txt				
Name	Coursework	Exam	Total	
Fred Bloggs	55	65	60	
Susan Smart	100	90	95	
\$ java DeleteField 3 < input.txt java DeleteField 2				
Name	Total			
Fred Bloggs	60			
Susan Smart	95			
\$ _				

The above should also work on Microsoft Windows (except use type rather than cat if you want to list the original text file).

18.5 Section / task 18.5 Numbering lines from text file to text file

- Aim of example: To introduce the principle of reading from a **text file** and writing to another, using `BufferedReader` with `FileReader` and `PrintWriter` with `FileWriter`. We also meet `FileInputStream`, `OutputStream`, `FileOutputStream` and `OutputStreamWriter`.
- Coursework title: **Deleting a field, from file to file**
- Coursework summary: Write a program to delete a field in tab separated text from a **file**, with the results in another file.
- Question: Write a version of your `DeleteField` program from Section 18.4 on page 85, that takes its input from a named **file** and puts its output in another named file. You should handle **exceptions** in the same way as we did for the example in this section.

18.6 Section / task 18.6 Numbering lines from and to anywhere

- Aim of example: To illustrate that reading from **text files** and from **standard input** is essentially the same thing, as is writing to **text files** and to **standard output**. We also look at testing for the existence of a **file** using the `File` **class**, and revisit `PrintWriter` and `PrintStream`.
- Coursework title: **Deleting a field, from anywhere to anywhere**
- Coursework summary: Write a program to delete a field in tab separated text either from **standard input** or a **file**, with the results going to either **standard output** or another file.
- Question: Write a version of your `DeleteField` program from Section 18.4 on page 85, that takes its input from **standard input** or a named **file**, and puts its output on **standard output** or in another named file. You should handle **exceptions** in the same way as we did for the example in this section.

18.7 Section / task 18.7 Text photographs

- Aim of example: To see an example of reading **binary files**, where we did not choose the **file format**. This includes the process of turning **bytes** into **ints**, using a **shift operator** and an **integer bitwise operator**.
- Coursework title: **Encoding binary in text**
- Coursework summary: Write a program to encode a **binary file** as an **ASCII text file**, so that it can be sent in an email.
- Question: Have you ever wondered how it is that you can send a **binary file**, such as an image, as an attachment inside an email message, when in fact an email is actually an **ASCII[?] text file**? The answer is simple: the binary file is coded as ASCII text when the email is constructed, and decoded back to binary again when the email is opened at the other end.

Search on the Internet to find out about a program called uuencode and how it codes sequences of 3 **bytes**, each using all 8 **bits** as in a binary file, into sequences of 4 ASCII **characters**, each using only 6 bits. ($3 \times 8 = 4 \times 6$.) Or, if you are using Unix then there is a good chance the program is already installed and you can find out about it using `man -a uuencode`.

Write your own program called Uuencode which performs this function. Its **command line argument** should be the name of the **file** to be encoded, and the result should go to **standard output**. You should handle **exceptions** using the same style as the example in this section. You can test your program by converting a binary file to ASCII, converting it back to binary again using a standard uudecode program (take care not to replace the original with the decoded one!), and comparing that result with the original. uudecode is available from the Internet or is probably installed if you are using Unix. You could use your CheckSum program to undertake the comparison (or on Unix you could use the `cmp` program).

The following **pseudo code** might help (after you have found out about the format that uuencode produces).

```
write the header -- assume file mode 600
create an array to hold the bytes for one line (partially filled)
read next byte
while next byte is not -1
    process a line of bytes and read next byte
output a line representing zero number of bytes
output the trailer line
```

We can refine this to the following.

```
write the header -- assume file mode 600
create an array to hold the bytes for one line (partially filled)
read next byte
while next byte is not -1
    while next byte is not -1 and array is not full
        put next byte in the array
```



```

    read next byte
end-while
output the number of bytes on this line
loop over the line array in groups of 3 bytes
    calculate the 4 output bytes for those 3 bytes
    output the 4 output bytes
end-loop
output an end of line
end-while
output a line representing zero number of bytes
output the trailer line

```

You will also find the following code fragments helpful!

```

...
009: // Write a single result byte as a printable character.
010: // Each byte is 6-bit, i.e. range 0..63.
011: // Thus adding 32 makes it printable, except for 0 which would become space
012: // and so we add 96 instead -- a left single quote (').
013: private static void writeByteAsChar(int thisByte)
014: {
015:     System.out.print((char) (thisByte == 0 ? 96 : thisByte + 32));
016: } // writeByteAsChar
...
056: // Calculate 4 result bytes from the 3 input bytes.
057: int byte1 = lineBytes[byteGroupIndex] >> 2;
058: int byte2 = (lineBytes[byteGroupIndex] & 0x3) << 4
059:             | (lineBytes[byteGroupIndex + 1] >> 4);
060: int byte3 = (lineBytes[byteGroupIndex + 1] & 0xf) << 2
061:             | lineBytes[byteGroupIndex + 2] >> 6;
062: int byte4 = lineBytes[byteGroupIndex + 2] & 0x3f;
063: // Now write those result bytes.
064: writeByteAsChar(byte1);
065: writeByteAsChar(byte2);
066: writeByteAsChar(byte3);
067: writeByteAsChar(byte4);
...

```

Optional extra: Write the Uudecode program too!

18.8 Section / task 18.8 Contour points

- Aim of example: To show an example of writing and reading **binary files** where we choose the **data** format, using `DataOutputStream` and `DataInputStream` **classes**.
- Coursework title: **Saving greedy children**
- Coursework summary: Add features to some existing model **classes** so they can be written and read back from **binary files**.

-
- Question: Copy the GreedyChild and IceCreamParlour **classes** from Section 11.3 starting on page 40 and add code so they can be written to a DataOutputStream and read back from a DataInputStream. You do not need to save the IceCreamParlour that a GreedyChild is in – so when a GreedyChild is read back, he or she will always not be in a parlour. Test your new features with a program called TestGreedyChildrenIO.

Optional extra: Figure out how to save and restore the IceCreamParlour that a GreedyChild is in. Perhaps each IceCreamParlour could have a unique ID number? Maybe that number would also be an **array index**? You may want to ensure that all IceCreamParlours are read (and hence written) before any GreedyChild is read.

Optional extra: (Challenge!) Find out about ObjectInputStream and ObjectOutputStream and use those instead.

19 Chapter 19 Generic classes

19.2 Section / task 19.2 A pair of any objects

- Aim of example: To explore potential problems of having a container **object** that can hold **instances** of any **class**, in particular that we need protection against us erroneously getting the **type** wrong when we extract items from the container. We also introduce the idea of **boxing** an **int** within an Integer.
- Coursework title: **A triple**
- Coursework summary: Write a **class** that can store a triple of **objects**, and use it.
- Question: Write a **class** called Triple, similar to Pair, except that its **instances** each store three **objects**.

Write a class called IntArrayStats containing a **class method** getStats() which takes an **array** of **ints** and **returns** a Triple containing the maximum **integer** in the array, the minimum, and also the mean of all the values. You will need to **box** the first two inside Integer objects, and the third inside a Double.

Test your work with the following program which measures how much the mean of a **set** of numbers differs from the average of its minimum and maximum.

```
001: // Program to measure how much the mean of the integer command line arguments
002: // differs from the average of their minimum and maximum.
003: // (Warning: this program does not catch RuntimeExceptions.)
004: public class MeanMinMaxMinusMean
005: {
006:     public static void main(String[] args) throws RuntimeException
007:     {
008:         int[] array = new int[args.length];
009:         for (int index = 0; index < args.length; index++)
010:             array[index] = Integer.parseInt(args[index]);
011:
012:         Triple stats = IntArrayStats.getStats(array);
```

```

013:    int max = ((Integer)stats.getFirst()).intValue();
014:    int min = ((Integer)stats.getSecond()).intValue();
015:    double mean = ((Double)stats.getThird()).doubleValue();
016:    System.out.println((min + max) / 2.0 - mean);
017: } // main
018:
019: } // class MeanMinMaxMinusMean

```

If you **run** the program with a set of consecutive numbers, the result should come out as 0.0.



Coffee time: What common **bug** could cause the result to be 0.5 when the program is given a list of consecutive numbers of a length which is even?

Experiment to see what happens when you make the same kind of mistake in the above program as we did in the example in this section.

19.3 Section / task 19.3 A generic pair of specified types

- Aim of example: To introduce the idea of **generic classes**, and show how it can be used to avoid the problems explored in the previous section.
- Coursework title: **A generic triple**
- Coursework summary: Write a **generic class** that can store a triple of specific kinds of **objects**, and use it.
- Question: Rewrite your **classes** from Section 19.2 on page 90 so that Triple becomes a **generic class**, and the other classes are altered appropriately.

(If you have read ahead, please do not use **autoboxing** – you will learn more by saving that for a separate task.)

19.4 Section / task 19.4 Autoboxing and auto-unboxing of primitive values

- Aim of example: To expose Java's implicit conversion between values of **primitive types** and **instances** of the corresponding wrapper **classes**.
- Coursework title: **A generic triple, used with autoboxing**
- Coursework summary: Write a **generic class** that can store a triple of specific kinds of **objects**, and use it; this time using **autoboxing** and **auto-unboxing**.
- Question: Rewrite your **classes** from Section 19.3 on page 91 so that **autoboxing** and **auto-unboxing** is used appropriately.

19.5 Section / task 19.5 A conversation of persons

- Aim of example: To introduce the idea of a **bound type parameter**, in particular, one that must **extend** some other **type**.
- Coursework title: **A moody group**
- Coursework summary: Write a **generic class** that can store a collection of a particular kind of `MoodyPerson` **objects**, from the Notional Lottery example, and make them all happy or unhappy at the same time.
- Question: This coursework is set in the context of the Notional Lottery game from Section ?? on page ??.

Write a **generic class** called `MoodyGroup` that contains a collection of some **subclass** of `MoodyPerson` **objects**, rather like the `Conversation` **class** does with `Person`. However, instead of a `speak()` **instance method**, `MoodyGroup` should have `setHappy()`. This will take a **boolean** and pass it to the instance method of the same name belonging to each of the `MoodyPersons` in the group. You will recall that only `MoodyPersons` have the `setHappy()` instance method, whereas the more general `Person` does not.

Test your class with a program called `TestMoodyGroup`. This will do the following.

- Create an **instance** of `MoodyGroup<Teenager>` and populate it with a small number of `Teenagers`.
- Invoke `setHappy()` with **false** and print out the group.
- Invoke `setHappy()` with **true** and print out the group again.
- Create a second moody group which can contain any kind of `MoodyPerson`, and populate it with a `Worker` and one of the *same* `Teenagers` which was put into the first group.
- Invoke `setHappy()` on the second group with **true** and print out the group.
- Invoke `setHappy()` on the second group with **false** and print out the group.
- Print out the first group one more time to show that the teenager which is in both groups stands out from the others.

20 Chapter 20 Interfaces, including generic interfaces

20.3 Section / task 20.3 Sorting a text file using an array

- Aim of example: To introduce the idea of **total order** and the `Comparable` **interface**. We also meet the `Arrays` **class**.
- Coursework title: **Sort a text file**
- Coursework summary: Implement the program to **sort a text file**.

- Question: Write the program Sort as described in the example for this section. The following fragments may help you.

```
...
006: import java.util.Arrays;
...
055:     Arrays.sort(lineArray, 0, noOfLinesReadSoFar);
...
```

20.4 Section / task 20.4 Translating documents

- Aim of example: To explore **generic interfaces**, observe that Comparable is generic, see that String **implements** it, meet equals() from Object and talk about consistency with compareTo(). We also introduce **generic methods**, **binary search**, revisit Arrays and note that an **interface** can **extend** another.
- Coursework title: **Minimum and maximum Comparable**
- Coursework summary: Write a **generic method** to find the minimum and maximum items in an **array** of Comparable items.
- Question: Write a **class** called MinMaxArray which has one **class method** that takes an **array**. It will be a **generic method** with one **type parameter** that is comparable with itself, and the array shall have that **array base type**. It will **return** an **instance** of the **generic class** Pair from Section ?? on page ??, comprising the minimum and the maximum items from the array, based on the **natural ordering** of the items. It should **throw** an IllegalArgumentException if the array is empty or non-existent.

Test your class with a program called TestMinMaxArray.

20.5 Section / task 20.5 Sorting valuables

- Aim of example: To introduce the idea that a **class** can **implement** many **interfaces**, and explore what it means for an **interface** to **extend** another. We also take another look at having consistency between compareTo() and equals().
- Coursework title: **Analysis of compareTo() and equals()**
- Coursework summary: Undertake an analysis of previous uses of compareTo() and equals() **instance methods**.
- Question: We saw examples of compareTo() and equals() **instance methods** in various **classes** before this chapter. Now that you know about the Comparable **interface** and the equals() instance method from the Object class – which takes (a **reference** to) an Object as a **method parameter**, find all those previous places and identify the changes we should make. Record them in your logbook.

21 Chapter 21 Collections

21.2 Section / task 21.2 Reversing a text file

- Aim of example: To introduce the Java **collections framework**, and in particular the idea of **list collections**, the `List` **interface** and the `ArrayList` **class**.
- Coursework title: **Sorting election leaflets**
- Coursework summary: Write a program to **sort** election information leaflets into delivery order.
- Question: Being disillusioned with the main political parties, you have recently joined the newly formed “Sort it out” party. As an election is looming, they have asked you to distribute campaign material in your area. They have sent you a stack of leaflets for each street, each with a label on the front showing the names of its recipients. Here are examples of two labels.

Augustus Belcher, Regents Crescent	Joanne Smith and Lionel Brown, Regents Crescent
---------------------------------------	--

How ‘sorted out’, *they* think. That is, until you tell them they have failed to print the house numbers on the leaflets, only the street names! They quickly email you a **text file** for each street, containing the recipient names for each house, in house number order. For example, the **file** for Regents Crescent is called `regents-crescent.txt`, and contains the following.

Console Input / Output

```
$ cat regents-crescent.txt
1 Joanne Smith and Lionel Brown
2 Augustus Belcher
3 Fatima Bacon and Gaynor White
4 Celina Simmons and Rupert Rodgers-Smythe
5 Ahmed Hussain
6 Samuel Peacock and Sarah Peacock
7 Hsin Cheng Liu
8 Blanche Peacock and Harry Peacock
$ _
```

The first line is the names of the people who live at number one Regents Crescent, the second is the names for number two, and so on. The party officials tell you to **sort** the leaflets into this order before delivering them.

However, you are cleverer than that. You will write a program to sort a file into delivery order, that is, the order of walking up one side of the street and down the other. As it happens, you know that all the streets in your area are symmetrical, with odd numbered houses on the left, and even numbered ones on the right, both ascending in the same direction.¹

¹In this simplistic world, obviously the houses on the outer curve of Regents Crescent have bigger gardens

In this task you will write the delivery order sorting program, calling it `StreetOrder`. It should take two **command line arguments**, the name of the original file, and the sorted file to be created. It should work by reading the lines from the input file into an `ArrayList` of `Strings`. Then it should **loop** forwards through all the even indices of the **list**, printing the lines to the output file. That will be the details for the odd numbered houses. Finally it should loop *backwards* through the odd indices of the list and print those lines. So the output for the above input would be as follows.

Console Input / Output	
\$	<code>java StreetOrder regents-crescent.txt regents-crescent-sorted.txt</code>
\$	<code>cat regents-crescent-sorted.txt</code>
1	Joanne Smith and Lionel Brown
3	Fatima Bacon and Gaynor White
5	Ahmed Hussain
7	Hsin Cheng Liu
8	Blanche Peacock and Harry Peacock
6	Samuel Peacock and Sarah Peacock
4	Celina Simmons and Rupert Rodgers-Smythe
2	Augustus Belcher
\$	<code>_</code>

The program should be able to handle files which have an odd number of lines – some of the streets are a cul-de-sac of detached houses with one in the middle at the bottom.

Here is a reasonable set of **test cases** for the program.

#	Test case description
1	No command line arguments.
2	Only one command line argument.
3	An input file that does not exist.
4	An output file that has a leading directory that does not exist.
5	An output file that has a leading directory which is not writable (e.g. the root directory on Unix, /).
6	An input file with no odd numbered houses and no even numbered houses (e.g. /dev/null or nul).
7	An input file with one odd numbered house and no even numbered houses (i.e. one line).
8	An input file with one odd numbered house and one even numbered house (i.e. two lines).
9	An input file with two odd numbered houses and one even numbered house (i.e. three lines).
10	An input file with two odd numbered houses and two even numbered houses (i.e. four lines).
11	An input file with three odd numbered houses and two even numbered houses (i.e. five lines).
12	An input file with three odd numbered houses and three even numbered houses (i.e. six lines).

than those across the road!

As usual, devise **test data**, before **designing** the program, and create input files ready for testing. Record this in your logbook.

Now design and implement the program. You should handle **exceptions** in the same way as we did for the example in this section. After implementation, **run** the program with the tests you designed beforehand. Record in your logbook the outcome and any unexpected results together with their cause and how you fixed any **bugs**.

21.3 Section / task 21.3 Sorting a text file using an ArrayList

- Aim of example: To reinforce the use of ArrayList, in particular, showing uses of the `set()` **instance method** of a List. We also note that an **array** can be created from a List, and vice versa. Finally, we look at the Collections **class** and observe that it has a `sort()` **generic method**.
- Coursework title: **Sorting election leaflets, with `compareTo()`**
- Coursework summary: Write a program to **sort** election information leaflets into delivery order, using a `compareTo()` **instance method**.
- Question: In this task you will write the same program as in the coursework for Section 21.2 on page 94, but in a different way.

Create a **class** called `DeliveryHouseDetails`, which is `Comparable` with itself. This will store a house number in an **instance variable**, and the person name details (including the house number) in another. It will have an **accessor method** to obtain the person names. It will also have another **instance method**, `compareTo()`, which orders `DeliveryHouseDetails` **objects** by delivery order. Here is some **pseudo code**.

```
compareTo (other)
{
    if both house numbers are odd
        return this house number minus the other one
    else if both house numbers are even
        return the other house number minus this one
    else if this house number is odd
        return -1
    else
        return 1
}
```

This will cause a List of `DeliveryHouseDetails` objects, when processed by `Collections.sort()`, to be **sorted** into the required delivery order, as described in the coursework for Section 21.2 on page 94. Convince yourself this is true and write notes about it in your logbook.

Copy your `StreetOrder` class from the previous version, and modify it so that it creates a `DeliveryHouseDetails` object for each input line and stores it in the ArrayList. You can simply count the lines to obtain the house number – there is no need to extract it from the details on the line. After loading the details the program will use `sort()` from the

Collections class to sort them, then it can print them out by **looping** through them all, and extracting the person details.

Implement the `DeliveryHouseDetails` class. Note, you should also include an `equals()` instance method which is consistent with `compareTo()`. The following code should do the trick.

```
...
043: // Equivalence test, consistent with compareTo.
044: @Override
045: public boolean equals(Object other)
046: {
047:     if (other instanceof DeliveryHouseDetails)
048:         return houseNumber == ((DeliveryHouseDetails)other).houseNumber;
049:     else
050:         return super.equals(other);
051: } // equals
...
```

After implementation, **run** the program with the same tests you used for the first version, and record the results in your logbook. Now, think about which approach is best, and write notes in your logbook.

21.4 Section / task 21.4 Prime numbers

- Aim of example: To introduce the idea of **set collections**, the `Set` **interface** and the `HashSet` **class**. For this we explore **hash tables** and meet `hashCode()` from `Object`. We also see that the class `Integer` **implements** `Comparable<Integer>`.
- Coursework title: **Finding duplicate voters**
- Coursework summary: Write a program to detect people voting more than once in voting records.
- Question: The government have been encouraging more people to vote, and one of the features of a new system is that voters are allowed to do so in any polling station within a certain radius of their home, rather than just one. The idea is that more people can vote at lunch time, near to where they work. Unfortunately, this opens up additional potential for multiple voting, by people visiting more than one station! The officials have collected **data** from across the region, and want you to write a program to detect multiple votes. The input is in the form of a **text file** consisting of two lines per vote. The first line uniquely identifies the voter by their name, house number and post code. The second line records the time and location of the vote cast. The location is the name of a polling station, in the form of an area name and an identity number, such as `Manchester 538`. For example, here is a (cut down) set of data.

Console Input / Output

```
$ cat voting.txt
(Output shown using multiple columns to save space.)
Rupert Rodgers-Smythe, 4, M25 7QZ      Sarah Peacock, 6, M25 7QZ
07:37 Manchester 538                    14:59 Manchester 537
Fatima Bacon, 3, M25 7QZ                Joanne Smith, 1, M25 7QZ
10:01 Manchester 538                    15:09 Manchester 538
Samuel Peacock, 6, M25 7QZ              Giles Schubert, 3, M19 4FK
10:25 Manchester 538                    16:19 Manchester 189
Sarah Peacock, 6, M25 7QZ                Blanche Peacock, 8, M25 7QZ
10:25 Manchester 538                    16:37 Manchester 538
Phillip Jones, 13, M1 OKY                Ahmed Hussain, 5, M25 7QZ
10:32 Manchester 605                    17:21 Manchester 538
Lionel Brown, 1, M25 7QZ                Gaynor White, 3, M25 7QZ
11:17 Manchester 538                    18:50 Manchester 538
Margaret Chopin, 9, M37 9MP              Sarah Peacock, 6, M25 7QZ
12:14 Manchester 299                    19:01 Manchester 539
Rupert Rodgers-Smythe, 4, M25 7QZ        Annette Longbridge, 8, M19 6QP
12:27 Manchester 099                    19:07 Manchester 314
John Bach, 11, M2 9WQ                    Harry Peacock, 8, M25 7QZ
13:27 Manchester 308                    19:21 Manchester 538
Hsin Cheng Liu, 7, M25 7QZ              Margaret Chopin, 9, M37 9MP
13:27 Manchester 538                    19:30 Manchester 308
Celina Simmons, 4, M25 7QZ              Augustus Belcher, 2, M25 7QZ
14:12 Manchester 538                    20:59 Manchester 538
Gregory Beethoven, 5, M17 8XJ           Sarah Peacock, 6, M25 7QZ
14:22 Manchester 009                    20:59 Manchester 540
$ _
```

They simply want your program to detect and report duplicate voter identifications, followed by the number of duplicates found. So, the output for the above input would be as follows.

Console Input / Output

```
$ java DuplicateVoters voting.txt voting-duplicates.txt
$ cat voting-duplicates.txt
Rupert Rodgers-Smythe, 4, M25 7QZ
Sarah Peacock, 6, M25 7QZ
Sarah Peacock, 6, M25 7QZ
Margaret Chopin, 9, M37 9MP
Sarah Peacock, 6, M25 7QZ
There were 5 duplicate votes
$ _
```

Your program should use a `HashSet` to store the voter identifications, i.e. the first line of each pair of lines. (It will just skip over and ignore the second line of each pair – in this version.) If when adding to this `set`, using `add()`, the result of the addition is `false`, then the voter was already present and so the voter identification being added must be a duplicate.

The program should be called `DuplicateVoters`, and take two **command line arguments**, the first being the name of the input file, the second being the name of the resulting report file.

To save time, you may test your program using just the above sample data.

21.5 Section / task 21.5 Sorting a text file using a TreeSet

- Aim of example: To introduce the `TreeSet` **class**, for which we explore **ordered binary trees** and **tree sort**. We also meet the `Iterator` **interface**, together with how it is used on a `List` and a `Set`, especially a `TreeSet`.
- Coursework title: **Sorting election leaflets, using a TreeSet**
- Coursework summary: Write a program to **sort** election information leaflets into delivery order, using a `TreeSet`.
- Question: In this task you will write the same program as in Section 21.2 on page 94 in a third way. You will use your `DeliveryHouseDetails` **class** again, but instead of building a `List` of the **objects**, you will insert them into a `TreeSet`. Then, instead of **sorting** them using `sort()` from the `Collections` class, you will access the elements via the `Iterator` of the `TreeSet`.

Copy your `StreetOrder` class from the previous version, and modify it. After implementation, **run** the program with the same tests you used for the previous versions, and record the results in your logbook. If all three programs are working, then their outputs should be identical – there are, of course, no duplicate lines in the input **data**.

21.8 Section / task 21.8 Word frequency count sorted by frequency

- Aim of example: To introduce the `HashMap` **class**, and the fact that a **collection** can be built to initially contain the same values as some other collection. We also take a look at how we can go about making a good **override** of the `hashCode()` **instance method** of `Object`.
- Coursework title: **Finding duplicate voters, using a HashMap**
- Coursework summary: Write a program to detect people voting more than once in voting records, using a `HashMap`.
- Question: The election officials are very pleased with your work from the task in Section 21.4 on page 97 but they have found so many duplicate votes that they would now like you to modify the way the results are presented, to make them easier to process!

All they ask is that each time a duplicate vote is found, your program outputs it, together with the time and location of the duplicate *and* the time and location of the *first* occurrence of the naughty voter. So, the output for the input shown in Section 21.4 on page 97 would be as follows.

Console Input / Output

```
$ java DuplicateVoters voting.txt voting-duplicates.txt
$ cat voting-duplicates.txt
Rupert Rodgers-Smythe, 4, M25 7QZ
  Duplicate: 12:27 Manchester 099
  First occurrence: 07:37 Manchester 538
Sarah Peacock, 6, M25 7QZ
  Duplicate: 14:59 Manchester 537
  First occurrence: 10:25 Manchester 538
Sarah Peacock, 6, M25 7QZ
  Duplicate: 19:01 Manchester 539
  First occurrence: 10:25 Manchester 538
Margaret Chopin, 9, M37 9MP
  Duplicate: 19:30 Manchester 308
  First occurrence: 12:14 Manchester 299
Sarah Peacock, 6, M25 7QZ
  Duplicate: 20:59 Manchester 540
  First occurrence: 10:25 Manchester 538
There were 5 duplicate votes
$ _
```

Your program should use a `HashMap` to store the voter identifications processed so far, each mapped on to their *first* occurring time and location. So, the voter identifications will be **keys**, and the time and locations will be values in the map. Your program will read through the **file** as before, but this time it will not ignore the time and location lines. For each vote, it will check in the `HashMap` to see if that voter identification is already present – by using `get()` to try and retrieve the time and location of their first vote. If this is the first occurrence of the voter identification (i.e. the result from `get()` is the **null reference**) then all is well, and the voter identification mapped to the time and location is `put()` into the **map**. If on the other hand the voter identification is already in the map, then it is to be printed, along with the new time and location and the first time and location (retrieved from the map).

21.9 Section / task 21.9 Collections of collections

- Aim of example: To explore the idea that the elements of a **collection** can themselves be collections, and so quite complex **data structures** can be built.
- Coursework title: **Finding duplicate voters, using a `HashMap` of `LinkedLists`**
- Coursework summary: Write a program to detect people voting more than .QNonces in voting records, .QOOnce, using a `HashMap` of **objects** containing a `LinkedList`.
- Question: The election officials are very sorry to bother you again, but they have a new idea to make the processing of duplicate voting even more easy. They now would like the results grouped by fraudulent voter! (Of course, if you had been given the opportunity, you may well have pointed this out during requirements analysis at the start!)

This version of the program should produce the following results from the **data** shown in Section 21.4 on page 97.

Console Input / Output
\$ java DuplicateVoters voting.txt voting-duplicates.txt
\$ cat voting-duplicates.txt
Rupert Rodgers-Smythe, 4, M25 7QZ
07:37 Manchester 538
12:27 Manchester 099
Sarah Peacock, 6, M25 7QZ
10:25 Manchester 538
14:59 Manchester 537
19:01 Manchester 539
20:59 Manchester 540
Margaret Chopin, 9, M37 9MP
12:14 Manchester 299
19:30 Manchester 308
\$ _

The order of the voters might be different. Also note that your customers no longer desire to have a count of the duplicate votes.

Your program should use a `HashMap` to store the voter identifications, each mapped on to an **object** which contains a `LinkedList` of all the time and location lines for that voter identification. Once the input **file** has been read, your program will iterate through the values of this **map** looking for ones which have more than one vote, and reporting those it finds.

A `LinkedList` is arguably better than an `ArrayList` for storing the vote details of each voter, as each **list** just gets items added on the end, and then finally scanned via its `Iterator`. This means we do not get the inefficiency of `LinkedList`, because we are not accessing its elements in a random order, but we do benefit from each one being the exact size needed – most of them will contain only one item, and there will be very many of them.

The elements of the `HashMap` should be **instances** of a **class** called `VoterRecord`, which you will write. This will contain two **instance variables**, the identity of a voter, and a `LinkedList` of his or her voting times and locations, in the order found in the file. Its **constructor method** will be given the identity of the voter. It will have an **instance method** to add a voting time and location. Another instance method will **return** the number of times the person has voted. The `toString()` instance method should give a multi-line `String` representing the `VoterRecord` object, including the identity of the voter and the times and locations of voting, ready for use in the output of the program.

To obtain the efficiency of using `LinkedLists`, you must use an `Iterator` when scanning through the `LinkedList` in a `VoterRecord` to build the result of its `toString()`, rather than accessing each element by its **list index**.

Optional extra: Predict the effect of changing your `HashMap` to a `TreeMap`, and try it.

22 Chapter 22 Recursion

22.3 Section / task 22.3 Lecture attendance

- Aim of example: To introduce the idea of a **recursive algorithm**, with an example of one that is not intended for use on a computer.
- Coursework title: **Finding the most populated row**
- Coursework summary: Describe a **recursive algorithm** to be followed by humans.
- Question: Describe a **recursive algorithm** for finding from the lecture theatre the size of the row which contains the most people.

Optional extra: Also find the number of the row with most people in it – assume the rows are numbered from one, from front to back, and that every row has at least one person in it. (You cannot just assume the answer is the back row...!)

22.4 Section / task 22.4 Sum of ages of descendants

- Aim of example: To reinforce the idea of a **recursive algorithm**, with another example of one that is not intended for use on a computer. This one would not be easy to perform **iteratively**.
- Coursework title: **An iterative sum of ages algorithm**
- Coursework summary: Attempt to write an **iterative algorithm** that does the same work as a complex **recursive algorithm**.
- Question: Attempt to write an **iterative** version of the instructions to obtain the sum of the ages of the woman's descendants. It has to be instructions that would really work (albeit the woman herself only exists in fantasy). Hint: try hard, but be prepared to give up.

22.5 Section / task 22.5 Factorial

- Aim of example: To introduce the idea of a **recursive method**, present a simple example and talk about common misunderstandings. We also look at what it means for a recursive method to be **well defined** and compare **recursion** with **iteration**.
- Coursework title: **.QNReversing lines .QO Reversing lines**
- Coursework summary: Write a program to copy **standard input** to **standard output** but with the lines in reverse order, so that the first input line comes out last.
- Question: Write a program called ReverseLines to copy **standard input** to **standard output** but with the lines in reverse order, so that the first input line comes out last. Your

main method will set up a `BufferedReader` and a `PrintWriter`, and pass these as **method arguments** to another **class method**, which shall be a **recursive method**.

The recursive method will read the input lines and output them. It will *not* use **tail recursion**, in that it will perform some work *after* the **recursive method call**.

Think of an abstract sequence, `seq`, as either being empty, or being a head, `seq.head`, followed by a tail, `seq.tail`, itself a possibly empty sequence. Here is **pseudo code** for printing such an abstract sequence, `inputSeq`, in reverse.

```
if inputSeq is not empty
  recursively output inputSeq.tail
  output inputSeq.head
end-if
```

In this case, you are using a `BufferedReader` to obtain the sequence of lines, line by line, and the act of reading a line tells you whether you have read them all, and if not, moves the input onto the remaining lines, i.e. the tail. You will need a **variable** to save the head line. So we can recast the above general pseudo code as follows.

```
String head line
if trying to read the head line does not yield null
  recursively read and output the tail lines
  output the head line
end-if
```

For brevity, you may be skimpy with **exception** handling – just declare that each **method** **throws** `Exception`.

22.6 Section / task 22.6 Fibonacci

- Aim of example: To show an example of a **recursive method** which has **multiple recursion**.
- Coursework title: **A more efficient Fibonacci**
- Coursework summary: Implement Fibonacci using an **array** to remember the results.
- Question: Write a version of the Fibonacci **class** with a **recursive method** that has the same structure as the one here, but is made efficient by storing the result for *fib n* in an **array** at **array index** `n`. This technique to avoid recomputing results is sometimes known as a **memo function**.

22.7 Section / task 22.7 Number puzzle

- Aim of example: To solve a problem, using a **recursive method** with **multiple recursion**, which would be quite tricky to solve **iteratively**. Along the way, we look at the process of *designing* a **recursive algorithm**.
- Coursework title: **Extending NumberPuzzle**

- Coursework summary: Add two more **recursive method calls** to a **doubly recursive method**.
- Question: Extend our NumberPuzzle program so that the other two **arithmetic operators** are included. Hint: for **multiplication**, there is no point making a **recursive method call** if the target is not divisible by the number being ignored – in fact doing so would lead to erroneous solutions (due to **integer** truncation).

Optional extra: For a real challenge, why not allow brackets in the sequence?

22.8 Section / task 22.8 Dice combinations

- Aim of example: To show an example of a **recursive method** which has **multiple recursion** with **recursive method calls** inside a **loop**.
- Coursework title: **Anagrams**
- Coursework summary: Write a program to output all the anagrams of a word given as a **command line argument**.
- Question: Write a program called Anagrams which outputs all the permutations of a string given as a **command line argument**. The **main method** will, for efficiency, turn the first (and only) command line argument into a **char array**, using the `toCharArray()` **instance method** of the **String class**. It will also set up two other arrays of the same length, one, of **type char[]** to build the current permutation, and another, of **type boolean[]**, to record whether **characters** from the given string have been used so far in the permutation being constructed. It will then call a **recursive method** to print all the permutations.

Here is **pseudo code** for the recursive method.

```
printPermutations(int currentIndex)
{
    if currentIndex has gone past the end of the permutation array
        print out the permutation
    else
        for each index in the char array made from the given string
            if the character at that index is not already used in the permutation
                mark it as being in use (using the boolean array)
                put that character in the permutation at currentIndex
                printPermutations(currentIndex + 1)
                mark the character as NOT being used in the permutation
            end-if
        end-for
    end-else
}
```

Note that if the given string contains duplicate characters, then there will be duplicate permutations produced. This is fine.

Optional extra: Do it in a different **recursive** way, which does not need the **boolean** array nor a second **char** array. (Hint: swap the character at the given **array index** with each

other one at a greater index, in turn.)

Optional extra: Do it without using **recursion**.

22.10 Section / task 22.10 Tower of Hanoi

- Aim of example: To devise a remarkably short **recursive method** solution to a seemingly very tricky puzzle.
- Coursework title: **Tower of Hanoi with peg values**
- Coursework summary: Extend a Hanoi solving program to show the state of the pegs.
- Question: Write a version of the Tower of Hanoi program which actually models the discs on the pegs and prints them out at each move. You should have a separate **class** called **Peg** which models the actual discs on a particular peg. The disc sizes could be stored in a **partially filled array**, the value at **array index** *i* being the size of the disc at location *i* on the peg. Or you could perhaps find out about the standard **class** `java.util.Stack`, and make a **subclass** of that.

Here is sample output from **running** the program with a **command line argument** of 4.

Console Input / Output			
\$	java Hanoi 4		
0	Start:	L=< 4 3 2 1 >	M=< > R=< >
1	L to M:	L=< 4 3 2 >	M=< 1 > R=< >
2	L to R:	L=< 4 3 >	M=< 1 > R=< 2 >
3	M to R:	L=< 4 3 >	M=< > R=< 2 1 >
4	L to M:	L=< 4 >	M=< 3 > R=< 2 1 >
5	R to L:	L=< 4 1 >	M=< 3 > R=< 2 >
6	R to M:	L=< 4 1 >	M=< 3 2 > R=< >
7	L to M:	L=< 4 >	M=< 3 2 1 > R=< >
8	L to R:	L=< >	M=< 3 2 1 > R=< 4 >
9	M to R:	L=< >	M=< 3 2 > R=< 4 1 >
10	M to L:	L=< 2 >	M=< 3 > R=< 4 1 >
11	R to L:	L=< 2 1 >	M=< 3 > R=< 4 >
12	M to R:	L=< 2 1 >	M=< > R=< 4 3 >
13	L to M:	L=< 2 >	M=< 1 > R=< 4 3 >
14	L to R:	L=< >	M=< 1 > R=< 4 3 2 >
15	M to R:	L=< >	M=< > R=< 4 3 2 1 >
\$	_		

22.11 Section / task 22.11 Friend book

- Aim of example: To show an example of **recursion** based on walking through a **recursive data structure**. We also have a **private constructor method**.
- Coursework title: **Family trees**

- Coursework summary: Write a program to model family ancestry.
- Question: Write a program that enables the ancestry of people to be stored and printed out. You will want **objects** of **type** Person with a name and a **set** of other persons who are that person's immediate children. There's no need to model marriage (after all in modern life, many family 'tree's are not that simple). So if we wish to store two parents of a collection of children then we have those children contained separately in each of the parent's objects. (This permits the two parents of one child to have different sets of children.)

Your other **class**, containing the **main method** should be called FamilyTree.

The **data** should be stored in a **text file** called parent-children.txt. Each line consists of the name of a parent, followed by a **list** of his or her children, all separated by spaces.

The program should read this text file and take the name of a person as the first **command line argument**. It will then print out that person and his or her descendants as a 'family tree'. In theory the data should not contain any cycles (a person being their own descendant), however it might – so you should ensure the **recursion** cannot attempt to proceed forever.

To avoid distraction, you may ignore **exception catching** if you wish.

Here is some sample data (based on the UK Royal Family[?]).

Console Input / Output
\$ cat parent-children.txt
George-V Edward-VIII George-VI Mary Henry George John
Victoria-Mary Edward-VIII George-VI Mary Henry George John
Edward-VIII
Wallis-Simpson
George-VI Elizabeth-II Margaret
Elizabeth-Bowes-Lyon Elizabeth-II Margaret
Elizabeth-II Charles Anne Andrew Edward
Philip Charles Anne Andrew Edward
Charles William Harry
Diana William Harry
\$ _

And here is the corresponding output for George-V.

Console Input / Output

```
$ java FamilyTree George-V
+--George-V has 6 child(ren): Edward-VIII George George-VI Henry John Mary
|
|   +--Edward-VIII has 0 child(ren):
|   +--George has 0 child(ren):
|   +--George-VI has 2 child(ren): Elizabeth-II Margaret
|       |
|       |   +--Elizabeth-II has 4 child(ren): Andrew Anne Charles Edward
|       |       |
|       |       |   +--Andrew has 0 child(ren):
|       |       |   +--Anne has 0 child(ren):
|       |       |   +--Charles has 2 child(ren): Harry William
|       |       |       |
|       |       |       |   +--Harry has 0 child(ren):
|       |       |       |   +--William has 0 child(ren):
|       |       |       +--Edward has 0 child(ren):
|       |       +--Margaret has 0 child(ren):
|   +--Henry has 0 child(ren):
|   +--John has 0 child(ren):
|   +--Mary has 0 child(ren):
$ _
```

Optional extra: What simple change could you make so that the children of a person are listed in the order they were added (which would probably be the order of birth), rather than alphabetically by name? (Hint: look at the **application program interface (API)** documentation for `java.util.LinkedHashSet`.)

23 Chapter 23 The end of the beginning