CO7005 Software Development Techniques

Lab and Assessment Exercises

Module Leader

Dr Stuart Cunningham

s.cunningham@chester.ac.uk

Last Version: October 9, 2024

School of Computer and Engineering Sciences

University of Chester

Week 1

Java Fundamentals

1.1 Introduction

These exercises give you the opportunity to gain practical experience of working with the Java programming environment, displaying information to the console (output), performing calculations and other procedures (process), and getting the user to provide data (input) to your programs.

By completing these exercises, you will learn to:

- Write and run Java programs.
- Produce and manipulate output to the console.
- Declare simple variables and assign them values.
- Perform the four basic arithmetic operations in Java.
- Handle simple user input and include it in your program.

Programming is all about writing some code, trying it out, modifying it, and trying it again. It's a cyclical process and often involves making mistakes then fixing them. This is perfectly normal and nothing to worry about. Learning from your mistakes is an integral part of learning to program. Having said that...

⇒ **Remember to ask your tutor if you get stuck or have any questions.**

1.2 Getting Started

You can write a Java program using any text editor or Integrated Development Environment (IDE) you like. However, we will be using the Microsoft Visual Studio Code IDE.

There are several key steps to getting a Java program to run once it has been written:

1. Ensure that the code has been saved and that it is a `.java` file.
2. Compile the program using the `javac` command.
3. Assuming there are no errors, run the program using the `java` command.

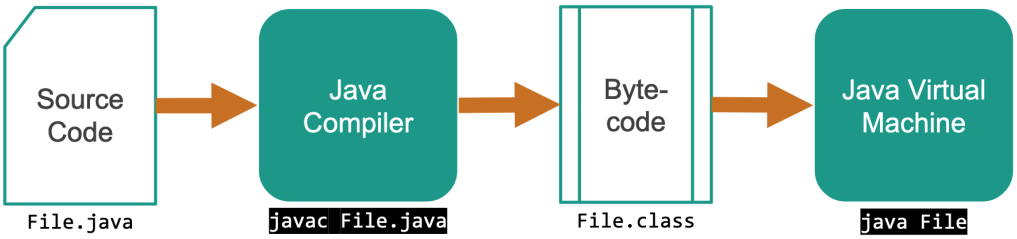

Figure 1.1: The Stages of Writing, Compiling and Running a Java Program

These steps were covered during the classroom activities and is summarised in Figure 1.1, which illustrates the processes of *compiling* a Java source file to create a Class and then *running* the Class (program) via the Java Virtual Machine (JVM). Notice that there are two main types of Java file we're working with currently: source files (with the `.java` file extension); and Class files (with the `.class` file extension).

HINT

The name of the class in your code must be the same as the name of the `.java` file and both names must start with a capital letter.

For example, suppose we have a Java file that contains:

```
class GettingStarted {
  // some things happen here
}
```

1. The code should be saved in a file named `GettingStarted.java`
2. It is compiled using the command `javac Gettingstarted.java`
3. It is run using the command `java GettingStarted`

1.3 Your First Programs

Remember the first program that you saw in Java was **HelloWorld** and it looked like this:

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

Exercise 1.3.1

Create a new Java program named **HelloChester** that displays the text "Hello, Chester!" to the console when run. You can base this program on the HelloWorld example program.

The output of your program should look something like the example below:

```
week-01-code % java HelloChester
Hello, Chester!
```

Exercise 1.3.2

Modify the **HelloChester** program so that it now displays the text "Hello, Chester!" on the first line and then on the line below it should display the text "Welcome to Software Development Techniques with Java".

The output of your program should look something like the example below:

```
week-01-code % java HelloChester
Hello, Chester!
Welcome to Software Development Techniques with Java
```

During classroom activities, we looked at giving the user with a way to provide *input* to our Java programs. This means that the programs we create can manipulate, or *process*, that input data.

HINT

Remember before you can setup and use the Scanner for input, it needs to be imported to your program. This can be achieved by adding the following statement as the very first line in your code:

```
import java.util.Scanner;
```

Look back at the examples shown if you are still unsure.

Data that is input by a user can be stored in Java programs inside a suitable *variable*. In the case of a user inputting some *text* to a program, such as their location or favourite colour, we often find it helpful to store this in a `String` variable. Once safely inside a variable we can cross-reference that user input at any time simply by using its name. This makes it easy to do a variety of tasks with user data, such as display it as output on screen.

◯ Assessed Exercise ◯

Write a new Java program named **CatStory**, which prompts the user to enter a name of their choice for a pet cat. The program will then produce a short story (the contents of which are shown in the example below, where the user has entered a name of 'Pixel'), with the name of the cat inserted in the correct places.

When working successfully, the output from your program should be like this:

```
week-01-code % java CatStory
What is your cat's name?
Pixel
Amina strolled through the park with her beloved pet cat, Pixel, nestled
comfortably in her arms.  Pixel, with his soft fur and bright green eyes,
had been her faithful companion for years.  As the sun began to set, Amina
whispered to Pixel, urging him to head home, their bond stronger than
ever.
```

1.4 Working with Numbers and Variables

There are various ways that numbers can be stored and manipulated in Java (and other programming languages). In the examples for this week, you saw ways to store integers (whole numbers) by declaring and naming a variable, then assigning it a value. You also saw the syntax for performing simple arithmetic, specifically addition, subtraction, multiplication, and division with these integers. Recall the following example, which adds together two integers:

```java
class AddTwoNumbers {
    public static void main(String[] args) {
        // declare variables 'a' and 'b' assign each a value
        int a = 8;
        int b = 2;
        // display the result of adding the two numbers
        System.out.print("The sum of numbers is ");
        System.out.println(a+b);
    }
}
```

Arithmetic operations are extremely common with variables representing numbers and can be performed at much larger scale than the simple examples here, meaning we can write programs that quickly and efficiently solve mathematical problems on large amounts of data. This fundamentally supports concepts relating to *big data* and *data analytics*.

Exercise 1.4.1

Create a new program, based on the AddTwoNumbers program that you saw previously. This new program should be called **AddThreeNumbers**. As you might have already guessed, the program should declare three variables with values of 20, 8, and 4. Your program should add these three numbers together and display the calculation to the user via the console.

Of course, working with numbers that are fixed inside the program code might be useful in certain situations, but most of the time we want to make programs applicable to a wider set of scenarios. Often, this means performing some processing on numbers (data) that the user provides as input.

Assessed Exercise

Write a new Java program named **AddTen** that asks the user to input a number. The program should then add 10 to the user's number and display the result of the calculation via the console. When complete, the program should look something like this:

```
week-01-code % java AddTen
Enter first number:  7
Your number 7 + 10 is 17
```

Performing mathematical operations with fixed values can be useful. But giving the user more flexibility, such as by being able to enter two or more numbers, is even better. This means the program is more generalisable, in other words, it can be used for more than just one purpose. Generalisability means writing code that can be used in many different situations and that can dynamically respond to changing needs of its users, data, or both. The more that you code, the better you will become at spotting, and taking advantage of, the opportunity to write more reusable code.

Assessed Exercise

Write a Java program named **TwoNumberMath** that prompts the user for two integer inputs and then performs the following arithmetic operations on them: *addition; subtraction; division; and multiplication*. Each of the three calculation outputs should appear on its own line and include the numbers entered and operation performed.

The output of your program should look something like the example. For instance, if the user entered the numbers 8 and 2, the program output would be:

```
week-01-code % java TwoNumberMath
Enter first number:
8
Enter second number:
2
8 + 2 = 10
8 - 2 = 6
8 / 2 = 4
8 * 2 = 16
```

Week 2

Data Types and Operators

2.1 Introduction

These exercises focus on the various primitive data types that Java supports and ways to manipulate, or process, them. This includes various mathematical operations, performing simple sets of calculations, and the organisation, and retrieval, of multiple values in arrays.

By completing these exercises, you will learn to:

- Work with Java's primitive data types.
- Handle user input of different data types.
- Use basic, extended, and relational operators.
- Declare arrays and perform basic operations with them.

⇒ **Remember to ask your tutor if you get stuck or have any questions.**

2.2 Data Types

As you now know, Java has a range of different variable types that can be used depending upon the situation or problem you are trying to solve. Knowing which type of variable to use is an important part of programming and can help improve efficiency, reduce potential errors, and make code easier and more intuitive for another programmer to read.

You will often make use of one, or more, of Java's eight *primitive* data types. Which, and how many, you use will depend upon the problem you are trying to solve and the scale of the program you are creating in response. Try to use the best-suited variable type early-on in your program development to reduce the need for changes or conversions later on.

Exercise 2.2.1

Write a program that stores the following data items using individual variables declared at the start of your program. The contents of these variables should then be displayed in the terminal window when the program is run:

```
18
-250
299792458.3856453
B
-3256.65
true
m
2.7182
```

You should select the most appropriate variable type when declaring each data item. Name your program **ManyVariables** and ensure the values are output correctly when the program is run.

The Scanner class, which we use for getting input from the keyboard, is helpful in providing methods for users to enter values to be stored using a variety of variable types. To facilitate input of different variable types, the Scanner class provides a variety of *methods*. For example, we've already seen use of the `nextLine()` method to allow users to enter a String of input, which can then be assigned to a variable inside a program.

Exercise 2.2.2

Imagine you work for an organisation that indexes all its technology assets using an alphanumeric system. In this system, index codes consist of a single uppercase letter, a three-digit number, and a sequence of five lowercase letters. For instance, a valid code would be `G523eemnq`.

Write a program named **AssetInput** that requires the user to enter each of the three code elements, store them in suitable variables, and then confirms the input to the user by outputting their complete code to the terminal window.

2.3 Operators

Once data is stored using variables and we can request different sorts of data from the user. A natural next step is to be able to manipulate and evaluate that data. This is especially useful when performing real-world tasks, especially numerical and arithmetic operations. These are often the foundation of many simple, but useful, computer programs and their functions. For instance, we might want to compare one value to another or perform some calculation or conversion. Having a full range of variable types means that these operations can be performed correctly and accurately. This section provides lots of exercises that will get you comfortable with carrying out these sorts of processes in Java.

Exercise 2.3.1

Write a program named **NumberGuess**. As the programmer, you should declare a variable in your program that represents a number you want the user to try to guess (keep it simple, pick a number between 1 and 20 and tell the user that's the range they must work within). When the program is run the user should be asked to enter their guess and your program will generate an output to tell the user if their guess is correct or not. Test the program to make sure it works when the user guess is wrong and when it is right.

Don't forget about Java's *relational operators*, as well as the arithmetic ones. These allow a programmer to make comparisons (for example, is x larger than y?) between two or more variables and return a true or false state (known as a *Boolean*), which can then be used to drive other behaviours in a program.

Exercise 2.3.2

Create a Java program named **MileKilo**. This program should receive an input from the user that represents a distance in number of miles. The program should convert the number of miles into kilometres and output the distance (in kilometres) for the user. The user should be able to input a value that is a rational number (e.g., valid numbers would include: 2.0, 3.4, 6.74, etc.). Test the program works - for instance, 3.1 miles is approximately 5 kilometres.

HINT

When converting, you can use the formula: 1 mile $\approx$ 1.609 kilometres.

As you start working with more than two variables or values that you want to use in a calculation, keep in mind the order of operations that must take place. In most cases, Java behaves as you would expect most digital tools or calculators to, including placing calculation segments in parentheses (brackets) to force precedence. There is a good, and incredibly detailed, guide to this topic available online from Princeton University, if you wish to explore it further.

Exercise 2.3.3

Write a program named **FourAverage** that allows the user to input 4 numbers and then calculate and display their average in the terminal window.

◯ Assessed Exercise ◯

Create a program named **BMICalc** that takes a person's weight (in kilograms) and height (in meters) as input and calculates and outputs their BMI. Use the float data type for weight and height input, and a double type for the BMI calculation and result.

HINT

BMI is short for "Body Mass Index" and is a metric used to determine if a person's weight is within healthy boundaries, relative to their height. As such a person's BMI is calculated using the formula $BMI = weight/height^2$.

◯ Assessed Exercise ◯

Write a program named **Money** that asks the user to input the number of £10, £20, and £50 notes that a user has. Then display the total number of *notes* they have and their total *amount of money*. Use an escape sequence to tabulate the information in the output window.

When running successfully, your program output should look something like the following:

```
week-02-code % java Money
How many £10 notes do you have?  3
How many £20 notes do you have?  0
How many £50 notes do you have?  2
----------
In total you have 5 notes and a total amount of £130.
      £10       Qty:3       (Value £30)
      £20       Qty:0       (Value £0)
      £50       Qty:2       (Value £100)
```

HINT

Remember that `\t` can be used in String output to perform a tabulation.

2.4 Arrays

Once we start to deal with a lot of data items, especially if they are related or are of the same type, it becomes useful to deal with them as a *group* or *set*. This is where *arrays* are beneficial to the programmer. The simplest version, a one-dimensional array, is essentially like a simple list, where each item has an index within the array that contains it.

Exercise 2.4.1

Write a program named **Backwards** that initialises an integer array with the values 0, 1, 1, 2, 3, 5, 8, 13. Get your program to output the contents of the array to the terminal in the reverse order to that listed here.

Java, like a lot of other programming languages, always employs an array index with the value of *zero* to refer to the *first* item in the array. Once you know this, you can use it as a reference to work out how to access any element in an array. But take care as you expand into other programming or scripting languages as they don't all follow this convention. Matlab and R are two languages you might encounter that do this, but there are others like COBOL and FORTRAN too.

◯ Assessed Exercise ◯

Write a Java program named **FiveStudents** that initialises an array of five integers, which represents five students' scores in a test. The user should be prompted to enter a number (between 0 and 100) to be assigned to each element in the array. Once this is complete your program should output whether each student has a pass mark (greater than, or equal to 50) or not along with the average of all five student marks. When complete, your program should look something like this:

```
week-02-code % java FiveStudents
Enter student 1's score:
45
Enter student 2's score:
50
Enter student 3's score:
100
Enter student 4's score:
0
Enter student 5's score:
49
--------------------------------
Student     Score     Pass/Fail
1           45        Fail
2           50        Pass
3           100       Pass
4           0         Fail
5           49        Fail
--------------------------------
The average score is:  48.8
```

Exercise 2.4.2

Write a program name **FourLetters** that asks the user to input a four-letter word of their choice. Once captured, store each character of the word as an array of characters. Your program should output each letter of the word on a new line in the terminal window.

HINT

You may find the `.charAt()` method useful in tackling this exercise.

Two-dimensional arrays can be thought of a little like a table, where they can be used to index related items of data and/or keep things in a sequence. Three-dimensional arrays, and those of higher dimensions, can be used to store large blocks or bodies of data that are similarly related or sequenced.

If you're using two, three, or more, array dimensions, remember each dimension you add requires another index. In other words, for an n-dimensional array, you need n indices to access a specific element. The easiest example of this to illustrate is the two-dimensional array, which has an index to refer to each *row* and one to refer to each *column*.

Exercise 2.4.3

Write a program named **MyMagic** that initialises and populates a table to represent the contents of a *magic square*[a], as indicated below.

8	1	6
3	5	7
4	9	2

Your program should store the square as a 2D integer array. It should output the magic square to the terminal window, making use of suitable escape sequences to tabulate the data neatly.

[a] https://www.rigb.org/explore-science/explore/blog/fascination-magic-squares

Week 3

Program Control

3.1 Introduction

These exercises are designed to introduce, and give you practice in, controlling the sequence of actions and iterations that can occur in your Java programs (as well as combinations of these two controls). Many programs that are written to solve real-world problems require decisions to be made (selection) and actions to be performed multiple times (iteration).

By completing these exercises, you will learn to:

- Implement selection statements of varying complexity.
- Become familiar with the use of `if` and `switch` statements.
- Make use of Boolean operators.
- Deploy iteration in programs to solve repetitive tasks.
- Determine when to use `for`, `while` and `do-while` loops

  ⇒ **Remember to ask your tutor if you get stuck or have any questions.**

3.2 Selection

Selection statements in programs allow us, as programmers, to change the direction, or flow, of our code so that it can respond in different ways. Typically, the parameters that are evaluated in this process are *data* or *variables*, often entered by the user. Simply put, this means that we can make decisions and take follow-up actions accordingly. This brings a lot of flexibility to our solutions.

Recall from the lecture that a simple `if-else` statement follows the following structure:

```
1 if (test_condition) {
2   // this happens if test returns true
3   doSomething();
4 }
5 else {
```

```
6   // otherwise this happens (test returns false)
7   doAnotherThing();
8 }
```

Exercise 3.2.1

Write a program that asks the user to enter their age. The program will then respond to the user to tell them if they are eligible for the state pension payment or not. Persons can receive their state pension if they are 66 years or over. Test the program for ages under, at, and over the limit of 66 years. Save this program as **PensionCheck**.

Conditional, or Boolean logic, operators can be used to make more complex decisions that require values or data to be *compared*. This might be where we want to look at combinations of different variables, for example. These operators allow us to follow logical associations of AND, OR, and NOT. Boolean operators can be integrated into many of the conditions that we test for in programs, including as part of `if-elseif-else` or even `switch` statements. The Boolean logical operators, and their Java syntax shown in Table 3.1.

Logical	Java
AND	`&&`
OR	`||`
NOT	!

Table 3.1: Boolean Logical Operators in Java

Exercise 3.2.2

Write a program named **DrivingLicense**, which will tell a user if they are eligible to apply for their provisional driving license. A provisional driving license can be applied for providing a person meets the following criteria:

- They are at least 16 years old.
- They have been given permission to live in Great Britain.

Your program will prompt the user to enter their age (a number) and confirm if they do, or do not, have permission to live in Great Britain (a Yes/No response). The program should analyse these data using an if statement and tell the user if they are eligible for their provisional license, according to the rules.

As you are designing programs that make decisions using multiple pieces of data, there may be more than one way to solve the problem. For example, you might choose to write code that deals with each data item and decision *in turn*, possibly applying nested conditions, or you might deal with them in *combination*, which is often where the use of Boolean operations is essential. By understanding logical tests *individually* and in *combination*, you can create more flexible and powerful Java programs, whilst also considering code *efficiency*.

Exercise 3.2.3

Write a program named **TimeGreeting** that asks the user for the current time in the 24-hour clock format (e.g., 1430 or 815) and uses `if-elseif-else` statements to display a greetings message to the user based on the time of day (e.g., "Good morning", "Good afternoon", "Good evening" or "Good night").

HINT

You can choose the exact time boundaries applied for your program to switch between the different greeting options, so long as they are sensible.

Nested conditions, where one (or many) `if` statements are placed inside another, are used when more complex decision-making is required in a program. The full set of `if-elseif-else` statements can be used within nested conditions. The terms *inner* and *outer* are used to help identify which condition is being referred to within the nesting structure.

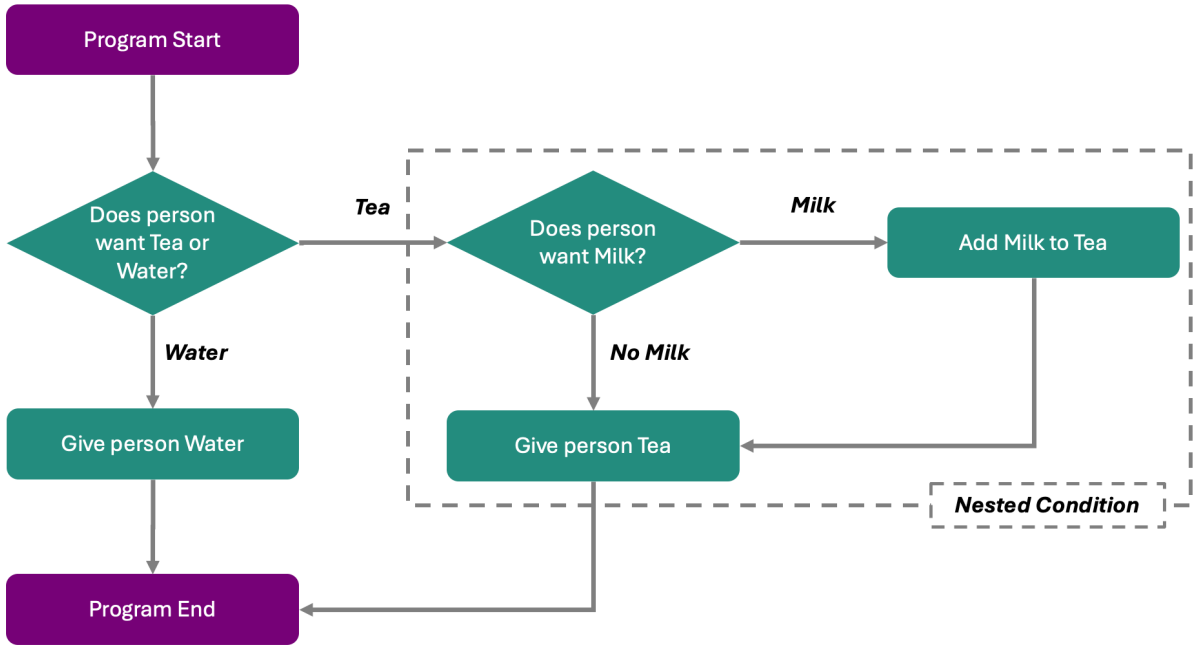

Figure 3.1: Drink Choice Program Flow with a Nested Condition (adding milk to tea or not)

An example of a *nested condition* can been seen in Figure 3.1, which illustrates a simple program that asks a person (user) if they would like to drink tea or water. The outer condition establishes if they want tea or water, whilst the inner condition established if milk should be added to the tea or not.

◯ Assessed Exercise ◯

Cestrian Insurance Limited is a company that offers motor vehicle insurance. Their current insurance premiums and rules are:

- Car = £305.
- Motorcycle = £360.
- Increase the base premium by 30% for customers under 25 years of age.
- Add £100 for customers with more than 6 penalty points.

Write a program input the type of vehicle (car or motorcycle), the customer's age, and number of penalty points/ Your program should calculate and display the total premium payable. Save this program as **CestrianInsurance**.

`Switch` statements make it convenient to evaluate long chains of possible values. This makes them a viable alternative to extensive `if-elseif-else` statements. They are especially useful when dealing with input where the user has been asked to select a value from a known, or pre-defined *list* or *set*, such as a series of characters, words, or numbers. Remember that the general syntax for a `switch` statement looks like this:

```
switch(variable) {
  case condition1:
    doSomething();
    break;
  case condition 2:
    doSomethingElse();
    break;
  default:
    doAnotherThing();
}
```

Applying `Switch` statements is especially useful when your program needs to deal with many possible outcomes based on a single value test. Remember that the `break` statement is important when coding `switch` statements. Use it after each case to prevent the program from carrying on to the next case (unless this is a behaviour that you specifically want).

Exercise 3.2.4

Write a program that asks the user to enter the name of one of the twelve months of the year as input and then uses a `switch` statement to display the number of days in that month. Remember to explain to the user that the number of days in February can change if it is a leap year. Add appropriate logic to make sure that the program still works if the user forgets to put a capital letter at the start of the month. For example, the input "october" would function the same as "October". Save this program as **MonthDays**.

HINT

You can also use the String method `equalsIgnoreCase()` to do an even better job of ignoring upper and lower-case letters in future.

3.3 Iteration

Iteration is applied in programs where we want to perform the same, or a very similar, task multiple times. This saves writing lots of recurring lines of code over-and-over, making the programmer's life simpler, as well as producing shorter, more efficient code. Think of iteration as a loop in your code, much like a loop in a roller coaster. You go through the same set of actions, or circuit, multiple times. Depending on whether we know how many times we want to do something makes the difference between choosing a `for` or a `while` loop in many cases.

Exercise 3.3.1

Produce a program that asks the user to enter a number between 1 and 10. Check that their input is valid. You should then display the multiplication table for the number that the user has entered, up to the twelfth row of the table. For example, if the user entered the number 5, your program would produce output along the lines of the following:

1 x 5 = 5
2 x 5 = 10
3 x 5 = 15
...
12 x 5 = 60

Test the program to make sure that it works for different input values and that the calculations are correct. Save this program as **MultiplicationTable**.

Exercise 3.3.2

Design and create a program that asks 10 users what their favourite colour is from four possible options of: *green*, *orange*, *red*, and *purple*. When all 10 users have entered their colour choice, the program should output how many people selected each of the four colours and identify which colour was most popular. In the case of a draw, name the colours included in the draw. Test your program several times to check that it is counting the number of occurrences correctly and name this program **ColourCount**.

HINT

You don't always need to store every value or piece of information that a user enters. Some problems only require you to count how many times something has happened instead.

Programs that make use of loops (almost) always require a *stopping condition* to be set where the loop should end. Otherwise, we are left with a program that will run forever, which is most cases isn't what was required. Stopping conditions for loops in programs can set using a Boolean check (something being true or false) or may involve a counter (a numerical variable that counts up or down that stops the loop when a specific value is reached).

◯ Assessed Exercise ◯

Write code for a machine that squeezes fresh orange juice into defined size bottles for customers in a convenience store. The bottle sizes the machine uses are 125 ml, 200 ml, 500 ml, 1000 ml and 2000 ml. The program should ask the customer how much orange juice they want in millilitres (ml). It should then display the combination of bottles that best matches the amount of juice. The machine dispenses a minimum of 125 ml and a maximum of 5000 ml each use and the program should validate these thresholds are met. If the user enters an illegal value the program should keep asking until a valid amount is entered. Name your program **OrangeJuice**.
For example, if the user entered a value of 800 ml, the program would look like this:

```
week-03-code % java OrangeJuice
How much orange juice do you want (in millilitres)?:
800
You need the following bottles:
  1 x 500 ml (full)
  1 x 200 ml (full)
  1 x 125 ml (part filled)
Enjoy your juice!
```

As outlined earlier, `while` loops are useful if we don't know in advance how many times we need to iterate or if we need to wait for some condition to be met before stopping a series of iterations. This is another effective way to automate repetitive tasks in programs, whilst still providing a stopping condition, should it be required.

Exercise 3.3.3

Create a program named **EvenTwenty** that uses a while loop to display all even numbers between 2 to 20. At the end of the program display a message stating "Done" to the terminal window.

Exercise 3.3.4

Write a program named **TaxCalc** that prompts users to enter a real number (for example, 1853.56), which represents their monthly income in pounds. The program should then calculate the amount of tax that they need to pay (in this scenario, the current rate of tax is: 20% for *salaries of £1500 and less* or 30% for *salaries greater than £1500*).
The program should continue to run, asking users to enter their salaries, until someone enters the number -99. At which point it should stop running, after outputting the message "Tax calculations have now ended".

By combining selection (conditional) statements and making use of iteration (loops), you can create programs that solve problems of many different sizes. Sometimes the size and complexity of a program is down to the nature of the problem itself, at other times it depends upon the amount of data that a program needs to handle, and it may even be a combination of the two. But having a good grasp of *sequence*, *selection*, and *iteration* in a programming language means you're prepared to handle these challenges.

◯ Assessed Exercise ◯

Write some Java code in a program named **Jackpot** that will simulate playing a lottery game. The numbers in the lottery game range between 1 and 45. Ask the user to choose six unique numbers. The simulator should then draw six lottery numbers at random (remember, you can't draw the same number twice).

Get your program to compare the random numbers to the numbers chosen by the user and award the following criteria by way of some suitable output:

- 2 matching numbers = £10
- 3 matching numbers = £250
- 4 matching numbers = £45,000
- 5 matching numbers = £220,000
- 6 matching numbers = £6,400,000 (jackpot!)

HINT

Some programs require multiple loops to solve the problem and these loops may not always be of the same type (e.g., `for` or `while`)

Exercise 3.3.5

Create a program that asks the user to guess a password. Use a `do-while` loop to keep asking for the correct password until the user enters the right one. The password to be guessed is "open-sesame". Save this program as **PasswordGuess**.

Week 4

Objects, Classes, and Methods

This set of exercises focus on the creation and use of methods and objects (classes) in Java. These form part of your first big steps into using Object Oriented Programming (OOP). Mastering OOP equips you with skills that are applicable across various programming languages and paradigms. As you work through these exercises, you will build a solid foundation in OOP, enabling you to create well-structured, modular, and reusable code. Don't hesitate to experiment and explore beyond the exercise requirements. The more you code, the more you learn.

By completing these exercises, you will learn to:

- Define and call custom methods within a program.
- Define and create custom classes and create instances of those classes.
- Utilise methods in custom classes.
- Pass and return data to and from methods and classes.
- Control use of class fields using access modifiers and encapsulation.

⇒ **Remember to ask your tutor if you get stuck or have any questions.**

4.1 Methods

Methods are essentially functions that operate within classes. A method is typically used to perform a defined operation or execute a coherent block of code. This is classically done to perform a commonly needed task. Methods define what objects of a class can do. Through these exercises, you'll learn to create methods that perform various actions and calculations.

Recall that the generic structure for a method inside a class takes the following format:

```
returnType methodName(params) {
    // things the method does
}
```

Exercise 4.1.1

Write a simple program in Java that contains a main method and another method called myDisplay. The myDisplay method should output the following information to the terminal window and must be called from the main method when the program runs.

```
+---------------------------------+
|||||||||||||||||||||||||||||||||||
* Software Development Techniques *
|||||||||||||||||||||||||||||||||||
+---------------------------------+
```

Name this program **SimpleBanner**.

A lot of methods are sent data when they are called from within a program. These data are known as parameters, and multiple *parameters* can be passed to methods, which can consist of a variety of supported data types. The receiving method must be configured to receive the desired *type* and *number* of parameters.

Exercise 4.1.2

Create a program entitled **Countdown**. In the main method of this program the user should be asked to enter a number between 5 and 20. The main method should check that the value entered by the user is within this range. The program should then call a method named printCountdown that receives an integer named 'start' and prints a countdown from 'start' down to 0. For example, if 'start' is 5, it should print "5, 4, 3, 2, 1, 0".

It's also common for methods to return, or send back, data to the line(s) of code that called them. This is done by utilising a *return type* when the method is declared and by including a return *statement* in the method (normally, but not always, at the end of the method).

Exercise 4.1.3

Produce a program named **OddOrEven**, which includes code in its main method that asks the user to enter an integer number of their choice. This number should then be passed to a method called numberCheck, which receives the value and determines if the user's number is odd or even. The numberCheck method should return a string containing either the word "Odd" or "Even". The main method should display this string to the terminal.

HINT

Remember that a number is even if it divides by two with no remainder.

Most contemporary programming languages like Java make use of methods. You may see similar code referred to as *functions* in other languages, like JavaScript. Methods and functions are similar in that they are defined blocks of code that can be called, passed data, and return data. The key difference is that a method must belong to (be part of) an *object*, whilst a function can be a completely *standalone*.

◯ Assessed Exercise ◯

Write a program named **CylinderVolume**, which receives two inputs representing the dimensions of a cylinder shape: height and radius, in centimetres. Include a method to calculate and return the volume of the cylinder. You should display the results of the calculation to the console, alongside the two values that the user input. For example: "The volume of a cylinder with a radius of 3 and a height of 12 is 339.29 cubic centimetres".

HINT

The volume (v) of a cylinder is calculated using height (h) and radius (r) according to the formula: $v = \pi r^2 h$.

4.2 Classes and Objects

A class is a template for creating objects, which are *instances* of that template. Objects have properties (attributes, commonly defined in variables) and behaviours (methods), and we'll explore how to define them. Classes in Java reflect the OOP principle of designing objects around the data that is to be stored and processed. Processing includes not only the transformation or manipulation of data, but also the ability to read and write to the variables that are contained within a class (often referred to as *getters* and *setters*).

Exercise 4.2.1

Create a **Triangle** class with attributes representing the *length* of each side in millimetres. Include a method to calculate the *perimeter* of the triangle. To demonstrate the use of this class, write a program named **Trio** that creates three instances of the Triangle class, sets their dimensions, and displays the perimeter of each in its main method.

HINT

The perimeter of a triangle is found by summing the length of its sides.

Consider the diagram shown in Figure 4.1 as an example of an object. It represents the design for a Pet class. This class was created to represent typical household pets that people may have. It includes three *attributes* (the pet's name, age, and gender) that any pet can have. The class also permits three *behaviours* that a pet may have (eating, sleeping, and making a noise). Notice that the Pet class is fairly *abstract*, it defines certain features for a pet but would easily allow an instance of the class to represent any one of a variety of pets, such as dogs, cats, rabbits, lizards, birds, and so on.

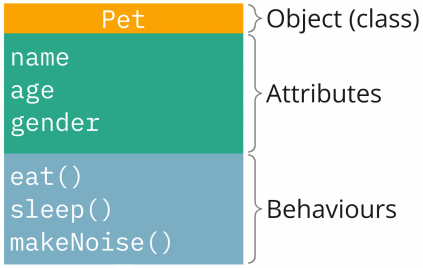

Figure 4.1: A Pet Class with defined Attributes and Behaviours

◯ Assessed Exercise ◯

Create a class named **Song**, which possesses the attributes (private variables) of *title*, *artist*, and *year*, demonstrating *encapsulation*. The **Song** class should contain a constructor that allows these values to be specified at instantiation. Suitable methods should be added that allow the attributes of a class instance to be returned. Write another program named **SongCollection** that creates five instances of the **Song** class (using music of your choice) that are stored in an array. The program should display the information about each of the three songs via the console.

In Java, *getters* and *setters* are methods used to access and manipulate the private fields (instance variables) of a class. They are an important part of *encapsulation*, one of the fundamental principles of OOP. Getters and setters provide controlled access to the internal state of an object, allowing you to ensure data integrity and maintain control over how data is read and modified.

Exercise 4.2.2

Define a **Car** class with attributes for *make*, *model*, and *year*. Implement methods to get and set information about each Car. Write a program named **Garage** that creates two instances of the **Car** class, sets their attributes, and prints the car details to the terminal via its main method (you can use any car information you like in your program). Once this has happened, prompt the user to enter details for their own, third car, create a new instance of the **Car** class using these details and then display this back to the user as well.

Methods that are well-written help to make your code more easily *readable* and encourage a *modular* approach to solving problems. A good method will typically do one, specific task. For this reason, large programs will usually have many methods that perform a variety of jobs. Adopting this approach will let you write *scalable* programs and you may discover opportunities to *re-use* your methods in different programs in future.

◯ Assessed Exercise ◯

Create a **BankAccount** class with instance variables for an account holder's *name* and *balance*. Include methods to deposit and withdraw money, as well as to display the balance. Write a program that demonstrates the functionality of the **BankAccount** class.

Week 5

Classes for the Masses

In this set of exercises, you'll gain further experience in the creation and usage of classes and objects in Java. Specifically, the relationships between classes (and objects) that can be created are explored and experimented with, which incorporates key object-oriented programming principles of *inheritance*, *polymorphism*, *encapsulation*, and *abstraction*, some of which you have seen in previous exercises too.

By completing these exercises, you will learn to:

- Create a hierarchy of related classes using inheritance.
- Adapt sub-classes for specific purposes, exploiting abstraction, polymorphism, and containment.
- Create programs that make use of recursive methods.

  ⇒ **Remember to ask your tutor if you get stuck or have any questions.**

5.1 Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP). Inheritance allows you to create new classes (sub-classes or derived classes) based on existing classes (super-classes or base classes). Sub-classes operate much the same as regular, independent classes, but they inherit properties and behaviours from their super-classes. This mechanism facilitates code reuse and promotes the creation of hierarchies of related classes.

Sub-classes operate much the same as regular, independent classes and are created in a very similar way. To define a sub-class, you add the keyword extends to the class declaration. This informs the Java compiler that you want to inherit properties and methods from a specific superclass. For example:

```
1 public class MySubClass extends MySuperClass {
2     // code goes here
3 }
```

Inheritance and sub-classes are powerful tools in OOP. They enable you to build upon existing classes and the modelling of real-world relationships within your programs.

Exercise 5.1.1

Produce a set of furniture classes (**Chair**, **Table**, **Bookcase**) with a common base (super-class) called **Furniture**. Assign attributes of *height*, *width*, *depth*, and *material* (e.g., wood, plastic, metal). The **Furniture** class should include a method that allows these attributes to be returned. Demonstrate the use of these classes, creating an instance, of each, by writing a program named **MyFurniture** that displays the attributes of each instance of a **Chair**, **Table**, and **Bookcase**.

In general, a super-class is a more general or *abstract* class that defines certain attributes and behaviours. A sub-class is a more specialised class that inherits these attributes and behaviours from the superclass and can add its own or modify them.

In Java, an *abstract* class is a class that serves as a blueprint or template for other classes, which may also contain abstract methods. Abstract classes cannot be used to directly create an object instance, but must be *extended* by other sub-classes, which are not abstract – we often refer to such classes as being *concrete*.

◯ Assessed Exercise ◯

Create a set of classes to be used in a program designed for a small, mobile catering outlet. The outlet sells a range of *products* to customers, which currently fall into the categories of drinks, snacks, and sandwiches. Every product the outlet carries has attributes of *name*, *price*, *calories*, and *quantity* (for stock keeping purposes). Create an *abstract* superclass named **Product** and subclasses of **Drink**, **Snack**, and **Sandwich**, which extend the superclass. Demonstrate the use of *inheritance* by writing a program named **SimpleCatering** that creates instances of each subclass and displays details about each type of product to the console. Code get and set methods where appropriate to allow the values of each drink, snack, and sandwich instance to be altered and read.

5.2 Polymorphism

Polymorphism is another cornerstone of working in OOP and it expands upon the principles of inheritance. It allows for flexibility and adaptability in working with objects by enabling a single interface (method or function) to represent various types of objects. Essentially, polymorphism empowers us to work with objects in a more generic way, providing a level of abstraction that's both powerful and versatile. This is typically achieved by *overriding* broad methods inherited from a super-class, moulding them to fit the needs of the sub-class. This allows objects of different classes to respond to the same method call in a way that's appropriate for their individual characteristics.

Exercise 5.2.1

Define a new class entitled **Fruit** which contains a method named `taste()`. Create sub-classes for three fruits with suitable class names (e.g., **Apple**, **Lemon**, **Orange**) that *override* the `taste()` method to describe the taste of each fruit and display this in the console. Write a program named **Taste** that demonstrates instances of each **Fruit** and their output when the `taste()` method is called.

HINT

You may find it helpful to start drawing diagrams for the classes that you need to create in your programs. This provides a simple way to plan the relationships between classes and sub-classes as well as to plan where attributes and methods will reside. Doing so *before* starting to write code can make your programming time smoother.

Runtime polymorphism, also known as dynamic polymorphism, is an important, fundamental concept and feature of creating OOP programs in Java. Runtime polymorphism is closely tied to inheritance, and it is primarily achieved through method *overriding*, which you have already performed.

◯ Assessed Exercise ◯

Update the **Product** superclass from the previous assessed exercise, creating a new class file named **ProductPoly**, adding a new method named consume, which should output a message to the console of the sound made when a person consumes a given product. Create new versions of each subclass, named **DrinkPoly**, **SnackPoly**, and **SandwichPoly**, which demonstrate the use of run-time polymorphism using the consume method.

- A drink should output "Glug, glug, glug."
- A snack should output "Crunch, crunch."
- A sandwich should output "Chomp!".

You should update the demonstrating program and save it as **SimpleCateringPoly**, getting it to call the consume method for each instance of your subclasses.

5.3 Containment

Containment is a fundamental concept in Java programming, where one class includes and holds instances of other classes, typically as attributes or fields. It represents the relationship between a container class and the objects it contains. In fact, if we remember that the `String` variable type in Java is an object, this is something that you have technically already done.

Exercise 5.3.1

Create a simple model for an animal shelter system. Create an **Animal** class that holds attributes of *animal species*, *name*, and include an **Address** object for the shelter's location. You will need to create an **Address** class that contains *street* and *city* fields. Develop a **Shelter** class that contains instances of two animals (you choose their characteristics), which demonstrates the application of containment by outputting the details of both animals to the terminal window.

Containment is essential for building complex and modular software, and it's particularly useful for encapsulating related data and behaviours within a single class.

◯ Assessed Exercise ◯

You should create a new class named **MediaLibrary**, which contains a list of songs. The **Song** class that was created in the assessed exercise from Week 4 (Objects, Classes, and Methods) should be reused and contained within the list of songs held by the **MediaLibrary**. Write another class named **MyMedia** that instantiates four songs of your choice, adds them to a **MediaLibrary** instance, and calls a method in **MediaLibrary** that displays the details of all songs to the console.

5.4 Recursion

Recursion occurs where a method inside a program can make a call to itself as part of its code. The consequence, in practice, is quite like the behaviour of a loop, where it is possible to carry out code contained in the method on multiple occasions. Normally, there is some change in a variable, and a test condition, each time the method is called to ensure that it will end – this is usually referred to as a *base case*. Unlike traditional iterative processes such as loops, recursion is a method for solving problems that can be broken down into smaller, similar sub-problems. In practice, recursion is often used to solve mathematical problems as well as working with, and organising, data structures like arrays and lists.

Exercise 5.4.1

Write a Java program named **RFactrorial** that calculates the factorial of a given positive integer using a recursive method contained within the **RFactrorial** class.

HINT

The factorial of a number is denoted by $n!$ where n is the integer of the calculation. It is calculated by multiplying the number by itself but decrementing by one each time until we get to a final multiple of 1. In practice, this means, for example: $5! = 5 * 4 * 3 * 2 * 1 = 120$ and $3! = 3 * 2 * 1 = 6$.

Week 6

Packages and Interfaces

In Java, packages and interfaces are particularly formidable features that extend the potential value and impact of object-oriented programming principles and encourage greater modularity and code re-use. Packages act like containers that help organise classes and related components. Each package serves as a logical unit, grouping together classes that share common functionalities. Interfaces play a significant role in promoting code re-usability. They define a contract that classes must adhere to, specifying a set of methods that participating classes must implement.

Both techniques are concerned with allowing greater organisation and flexibility around how classes (objects) can be used and re-used in Java. Packages facilitate encapsulation by grouping related classes, while interfaces facilitate polymorphism by defining common contracts for classes to follow.

By completing these exercises, you will learn to:

- Collate classes in cognate packages.
- Import and utilise packages, and their classes, in programs.
- Declare and define interfaces and their contained methods.
- Implement multiple interfaces in your programs to carry out a variety of operations.

  ⇒ **Remember to ask your tutor if you get stuck or have any questions.**

6.1 Packages

Packages are useful ways to collect related classes together in an organised fashion. This makes it easier to devise short, succinct class names, as well as to create bundles of classes (each with the potential to contain a range of attributes and methods) that can be re-used in other programming projects. Classes that belong to a package are typically stored in a hierarchical set of directories (folders) with each class being declared a member of a package. This directory structure reflects the package hierarchy, making it easy to locate and navigate related classes.

Any Java class can designate its membership of a package by declaring this prior to the class definition. So, for example, a class named `Hammer` would declare itself a s belonging to a package named `Toolbox` as follows:

```java
package Toolbox;

class Hammer {
    // class code goes here
}
```

Exercise 6.1.1

Create a new package that might be used in a system to facilitate an online shop that specialises in selling hats, t-shirts, and jeans. This package should be named **ShopItems** and contain classes **Hat**, **Tshirt** and **Jeans**. Determine a suitable architecture for these items and make sure each has at least one appropriate attribute (for example *colour*, *brand*, *size*, etc.) and a getter that will return this value. Import the package in a new program named **TestShop** and demonstrate instances of these classes.

HINT

If you're feeling adventurous, you could create an abstract class that the Hat, T-shirt, and Jeans classes extend through inheritance.

Packages facilitate the creation of cohesive bundles of classes, each encapsulating a specific set of attributes and methods related to a particular functionality or domain. These packages can then be reused across different projects, promoting code reusability, and hopefully saving development time.

As projects grow in complexity, packages play a crucial role in managing the codebase. They simplify development by breaking down the system into manageable and understandable components. Additionally, they aid in maintenance, as changes or updates to a specific functionality can be confined to the relevant package without affecting the entire codebase.

◯ Assessed Exercise ◯

Create a package called Utilities with a class **Calculator** that has methods for basic arithmetic operations. In another class, called **MyMaths**, import the **Calculator** class and use its methods to perform calculations on an appropriate set of values.

6.2 Interfaces

Interfaces allow us to specify abstract methods, as well as constant variables, to be used in a variety of other classes. Classes and programs that use of interfaces must make this explicit in their class definition using the `implements` keyword. Interfaces allow the declaration of abstract methods, which are methods lacking a defined implementation. These abstract methods serve as a blueprint, providing a method signature without specifying the actual code.

Exercise 6.2.1

Create a Java interface named **Shapes** that contains methods called `calculateArea()` and `calculatePerimeter()`. Implement this interface in classes of **Circle** and **Rectangle**. In a new program called **ShapeValue** create instances of these two shapes and show that these methods can be used to calculate their areas and perimeters.

HINT

It may help to make use of *constant* variables in your interface.

Interfaces mean required methods can be specified but the way in which each implementation is achieved is at the discretion of the programmer writing the implementing class. This provides a large amount of flexibility and customisation in our programs. Unlike inheritance, where a sub-class can only inherit from one super-class, a Java class can implement multiple interfaces.

◯ Assessed Exercise ◯

Define three interfaces, **Creature**, **Pet**, and **Rabbit**, with methods `sleep()` `stroke()` and `eat()`, respectively. Create a class called **MyAnimal** that implements both interfaces. Demonstrate how an object of the **MyAnimal** class can use methods from all interfaces. Comment on the efficiency and suitability of this three-interface approach.

Being able to employ multiple interfaces in a program further supports the adoption of *modularity* in programming. This is especially useful when the same methods and constants are likely to be used over many classes and/or programs. However, care needs to be taken not to develop too many similar interfaces, where there is a danger that there may be duplicates of a method's *signature* (its name and parameters).

◯ Assessed Exercise ◯

Create an interface called **MinMax** with two methods `min()` and `max()` designed for finding the minimum and maximum value held in an array of integers. Create a class named **MyData** that implements this interface and its methods. Use the **MyData** class to find the minimum and maximum value held by an input array and demonstrate this using a main method in the **MyData** class.

Although interface methods commonly require utilising classes to implement them, it is possible to create static methods, which can be applied simple by referring to the class name and the method required. This can be useful in deploying a variety of general-purpose methods that don't belong to a specific type of package or interface.

Exercise 6.2.2

Write a new interface with the named **RandFib** that contains a static method called `generate()`. When this method is called, it should return a number randomly selected from the set $\{0, 1, 2, 3, 5, 8, 13, 21, 34, 55\}$. Demonstrate that this interface and static method are operational by utilising it in a new program called **TaskSize**, which should return the generated value to the terminal.

Week 7

Exception Handling and Serialization

Errors can happen in programs for a variety of reasons. Some are more serious than others, but the key thing is how we foresee and manage these errors as programmers. In Java, *exception handling* is the mechanism that we employ to work through these design considerations. Exception handling in Java involves the use of `try`, `catch`, and `finally` blocks, allowing programmers to isolate and address specific types of errors. Overall, Java's comprehensive exception handling mechanism empowers programmers to create robust applications capable of gracefully managing errors and maintaining stability in various runtime scenarios.

One common cause of exceptions is when dealing with external files. These can range to files not being where they should be (or not existing at all) to them being locked or unsuitable for our programs to access. Working with files applies your knowledge of exception handing but, more importantly, provide a mechanism by which important data and information can be stored in the long term. An integral part of file handling in Java is *serialization*, a mechanism that allows objects to be converted into a byte stream for storage or transmission. Serialization plays a crucial role in storing and retrieving complex data structures. In Java, is easily done by ensuring that classes to be serialized implement the `Serializable` interface.

By completing these exercises, you will learn to:

- Identify and manage checked and unchecked exceptions.
- Create and manage custom exceptions.
- Work with local files and perform the basic CRUD (Create, Read, Update, Delete) operations.
- Serialize Java objects to files, facilitating read and write operations.

  ⇒ **Remember to ask your tutor if you get stuck or have any questions.**

7.1 Exception Handling

Exceptions can occur for a variety of reasons and may require handling before programs will compile (in the case of *checked* exceptions) or be beneficial to manage, though not affect the compilation of a program (in the case of *unchecked* exceptions). Although there is a temptation not to worry about unchecked

expressions for this reason, they may be *thrown* by unpredictable behaviours in a program, often caused either by incorrectly validated user input or programming errors.

Exercise 7.1.1

Write a program named **MathsCheck** that takes two integers as input from the user and calculates their quotient (the result of dividing the first number by the second). Implement exception handling to catch any potential *ArithmeticException* that may occur if the user attempts to divide by zero. If an exception is caught, prompt the user to enter valid input again.

Handling built-in exceptions in Java is incredibly useful and should be applied to allow you to control how programs fail, and hopefully then recover, if errors or problems occur. Sometimes, there will be errors or problems that occur when programs run that violate the data or logical expectations of your problem and its solution. *Custom exceptions* allow programmers to create their own exception types to handle specific error conditions in a more meaningful and specific way. Custom exceptions are derived from the standard Java `Exception` class or one of its sub-classes, providing a mechanism for developers to define and throw exceptions tailored to their application's requirements.

Exercise 7.1.2

Design a simple user authentication system where the user is prompted to enter a username and password. The 'correct' username and password can simply be represented as variable internal to your program, which should be named **Credentials**. You can choose what the correct username and password should be. Implement exception handling to catch a custom `InvalidCredentialsException`, which should be thrown if the entered username and password do not match your predefined values. Make use of `try-catch-finally` blocks in your program.

Combining exception handling with existing program logic is an effective way to build user-friendly data validation, error handling, and resource management mechanisms. Loops are a common way to facilitate this, where operations can be repeated as many times as desired before allowing a program to progress. Loops may also be deployed when working with resources, such as file handlers, to ensure they are properly closed even if exceptions occur, typically by using a `try-catch-finally` block.

◯ Assessed Exercise ◯

Create a program named **SalaryCheck** that receives a gross annual salary value as input from the user. Implement exception handling to ensure that the salary provided is a valid positive integer. If the input is invalid, catch the exception and prompt the user to enter their salary again. The program should not terminate until a valid input is provided.
When complete, your program should look something like the following:

```
week-07-code % java SalaryCheck
Enter annual salary (£):
-35000
Invalid salary entry.  Please enter a positive value.
Enter annual salary (£):
-25
Invalid salary entry.  Please enter a positive value.
Enter annual salary (£):
24700
--------------------
Your salary is £24700
```

7.2 Basic File Handling

Files are a great way to store data in a persistent way, so that it can be recovered and reused in future. Java provides a variety of packages and interfaces that help programmers to handle working with files. The basis for all these operations is oriented around being able to perform each of the CRUD (Create, Read, Update, and Delete) operations. CRUD operations are the foundation of data management in applications. They provide the means to interact with databases, files, and other data storage systems efficiently. Furthermore, CRUD operations are often directly tied to user interactions. Users expect applications to allow them to create, view, edit, and delete data as part of a seamless and intuitive user experience.

Exercise 7.2.1

Write a Java program named **MultiCreate**. The program should create a series of fifteen files inside a directory named '`MyData`'. These sequentially named files should follow the naming convention: `BigData01.dat`, `BigData02.dat`, ...`BigData15.dat`.

There are several packages, classes, and methods available in Java to assist with file handling. For *creation* and *deletion* of files the `File` class from the `java.io` package has methods that facilitate these two operations, which are `createNewFile()` and `delete()`, respectively. The `File` class encapsulates information about the file(s) used in a program, such as their name and path in the local directory structure. Do remember to make sure that file related exceptions are caught (and handled) when accessing your files.

Exercise 7.2.2

Write a complementary program called **MultiDelete** that will delete all fifteen of the files created in the previous exercise from the '`MyData`' directory.

HINT

Take care when using the delete operation in your programs. You may end up with unintended consequences, like accidentally deleting a file you wanted to keep!

For *read* and *update* operations with files the `Scanner` and `FileWriter` are useful. The `Scanner` class contains methods including `hasNextLine()` and `hasNextLine()`, whilst `FileWriter` has the `write()` method. Of course, there is often more than one way to carry out file operations depending on the nature of your project, but these should provide a solid starting point.

◯ Assessed Exercise ◯

Making use of the `Files` class, built-in to Java, write a program that will allow the user to input the names of their top three favourite foods. This list should then be written out to a text file, the name of which should be specified by the user. Verify that the file can be read back into your program and then its content appended with two more favourite foods (resulting in a total of five favourites in the file). Ensure that IO exceptions are handled appropriately. The program should be called **FavouriteFoods**. Verify that these operations have been successful by opening the text file in the local operating system.

7.3 Serialization

One of Java's key features as an object-oriented programming language, is its facility for programmers to design and instantiate custom objects or classes (in addition to those that are built-in). Being able to store instances of any given class can potentially save a lot of time and code repetition, especially where objects are used to hold data that programs will use. *Serialization* is a way to achieve this goal and produces a portable representation of a complex class object that can be stored indefinitely, as well as transferred and shared over computer networks.

The reverse process, known as *deserialization*, can be applied to such data files and allows their contents to be reconstituted inside any other Java program. Serialized data is platform-independent, meaning that it can be read and written by Java applications running on different platforms. This facilitates interoperability and data exchange between systems with diverse architectures.

Exercise 7.3.1

Create a custom class named **Users**, which will store a *username*, *email address*, and *password*. This class should be made *serializable*, so that instances of it can be saved to a file. In program called **UserDB**, create three instances of the **Users** class (you can choose the details for each fictional user), store them in an array, and then save the array object, using serialize, to a file named `MyUsers.dat`. Your program should then deserialize `MyUsers.dat` and display the contents of the file to the terminal window.

To use Serialization program must implement the `Serializable` interface. Serializing and deserializing data comes with computational overheads, security risks, and changing to objects in newer versions of your programs might make previously serialized objects no longer compatible with your new code. As you gain experience in programming, you'll be able to decide when this approach is appropriate and when it is not.

◯ Assessed Exercise ◯

Implement a program named **ProductFiles** that serializes and deserializes a custom object which represents products used by a mobile catering company. Define a class, named **Product**, with attributes of *name*, *price*, *calories*, and *quantity* (for stock keeping purposes). Serialize an instance of the class to a file, then de-serialise it and display its details. Provide features that allow products to be added and removed from the file. You should make use of the Serialize class, which is built-in to Java.

HINT

Experiment to see what happens when a serialized file with the name you want to change, or update, is written to.

Week 8

Concurrency and Parallelism

Programs that solve real-world problems often involve multiple steps. For example, data must be input, processed, or output, and there may be multiple instances of each of these activities in any one program. Sometimes these steps need to be executed in a specific sequence, making them linear or sequential in nature. However, at other times, it may be possible for some of these steps to take place at the same time, making them *parallelisable* – in other words, they can take place concurrently. If the latter, these program steps can be placed within a thread and the programming environment leveraged to execute multiple threads at the same time, exploiting the underlying computer architecture that the computer is run on. The best bit is that, as a programmer, we don't have to worry about that hardware, we simply specify what parts of our code can run in parallel and let the virtual machine take care of the rest.

By completing these exercises, you will learn to:

- Create and control individual and multiple threads in a program.
- Apply multi-threading to improve program performance.
- Apply synchronisation and wait conditions to ensure proper program execution.

⇒ **Remember to ask your tutor if you get stuck or have any questions.**

8.1 Working with Threads

Threads can be created in a variety of ways in Java. In fact, all programs include a main thread by default, we just haven't been aware of it until now. In addition to the main thread, we can create additional threads that can be run as well. Once a thread is running, a common control that we may want to exert upon it is to get it to pause for a period.

Exercise 8.1.1

Create a simple program named **FinalCountdown**. This program should make use of a loop to output a countdown from 10 to 1 in the terminal, with each number on a new line. Make use of the `Thread.sleep()` method to insert a one second delay between each output. The last line of output should be the text "Lift Off!".

The main benefit of threads is that each can be controlled by the programmer to run concurrently when this is beneficial and to go back to running in sequence when necessary (such as if a value needs to be calculated before a subsequent step can be performed).

Exercise 8.1.2

Modify the program **SimpleThread.java** (from the demonstration) so that `myThread` includes a second line of output that says, "It's nice to be a new thread in the world". Force the main thread to pause for 3 seconds and then for it to output another display line that says, "I'm the main thread again". Save your program and call it **TalkingThreads** (remember to update the class name in the Java file).

◯ Assessed Exercise ◯

Write a number program named **ThreadCount**. Create two threads, one that prints the *even* numbers from 1 to 16 and another that prints the *odd* numbers from 1 to 16. Ensure proper *synchronization* between them so that they display in the correct numerical order (lowest to highest).

In situations where threads are performing tasks that contribute to a shared result, `Thread.join()` helps to make sure that the final outcome is coherent. For example, if multiple threads are calculating partial results that need to be combined, calling `join()` on each thread guarantees that the combination occurs in the correct order.

Exercise 8.1.3

Write a program named **ThreadCalc** that will work with the following arrays:

$arrayA = 1, 2, 3, 4, 10, 20, 50, 100, 1000, 2500, 3000$
$arrayB = 2, 4, 2, 4, 10, 150$

- The program should perform the following tasks:
- Add all the numbers in $arrayA$ and put the answer in a variable *sum*.
- Multiply all numbers in $arrayB$ and put the answer in a variable *product*.
- Subtract *sum* from *product* and put the answer in a variable *result*.
- Display *result* to the user.

Your program should use at least three threads: one for the summation, one for the multiplication, and one for the result calculation and display.

HINT

Remember to make use of the `Thread.join()` method when sequence of thread execution is important.

◯ Assessed Exercise ◯

Using the **ConcurrentThread** example as a template, add two new sub-classes and link these to two new thread declarations. Name these new threads *lyrics1* and *lyrics2*. Using the `Thread.sleep()` method have *lyrics1* and *lyrics2* print the following lyrics with a 1 second (1000 millisecond) gap between each line:

lyrics1
Daisy, Daisy
Give me your answer do
I'm half crazy
All for the love of you It won't be a stylish marriage
I can't afford a carriage
But you'll look sweet on the seat
Of a bicycle built for two

lyrics2
Twinkle, twinkle, little star
How I wonder what you are
Up above the world so high
Like a diamond in the sky
When the blazing sun is gone
When he nothing shines upon
Then you show your little light
Twinkle, twinkle, all the night

Use the `Thread.join()` method correctly to ensure *lyrics1* is allowed to fully complete before *lyrics2* commences. Save this program as **LyricsThread**.

Sometimes, we may need to interrupt or cancel the execution of a thread. There could be multiple reasons for this, but most commonly this is in response to input from a user or the result of some error condition occurring, although it might also be due to some cooperative task or interaction between multiple running threads. Java provides a mechanism to interrupt the execution of a thread by using its `Thread.interrupt()` method. It is up to the programmer to design their threads to respond appropriately to interruptions and use them carefully.

Exercise 8.1.4

Modify the **FinalCountdown** program from earlier, saving this version as a new program called **AbortCountdown**. In this modified version, you should introduce a feature where if the user enters the number 0 in the terminal window, then the countdown thread will be interrupted and a message to this extent displayed to the user.

Whilst working with multiple threads can be a great way to improve efficiency and performance of programs, there are times where we want to limit the execution of code segments so that only one thread may run that code at any given moment. Often, this is done to prevent errors in program output. This can be

especially useful when managing a finite or critical resource. Such situations are known as *race conditions*. The `synchronized` keyword in Java is used to prevent multiple threads from running code in parallel and it can be applied to code blocks or methods.

◯ Assessed Exercise ◯

Create a program with two threads that simulate a *shared* bank account, which two people have access to. This requires the implementation of a shared balance variable. Both people should be able to perform withdrawal and deposit transactions with the account. Implement synchronisation to ensure that both threads update the bank balance safely. Each thread should perform at least five operations, including two withdrawals and two deposits. If at any stage, the bank balance falls below zero then an alert message should be displayed to the user. Output the bank balance to the console when both threads have finished executing. The program should be named **JointAccount**.

Week 9

Sorting

Sorting is a fundamental task in programming, especially when working with significant amounts of data that need to be accessed by the end users of a program. The goal is to arrange data in a way that makes it easier to search, retrieve, and analyse. The task relies on programs being able to represent underlying orders and sequences that exist in the data being uses. Often, this is relatively easy, such as applying ascending or descending numerical or alphabetical order, but in other cases it may be more complex.

There are many sorting algorithms that have been devised for programmers to use. Sorting large datasets can be resource-intensive, and the choice of sorting method can impact the program's efficiency. We've briefly looked at established ones, such as the *bubble*, *selection*, and *insertion* sorts, which are conceptually straightforward to follow, but are not particularly efficient when it comes to working with large numbers of items, with time complexities (how long the algorithm needs to run) of approximately $O(n^2)$, where n is the size (number) of items to be sorted. Others, such as *QuickSort*, which is used by many of the standard Java classes and relies on recursive algorithms, but achieve better performance, with time complexity closer to $O(n \log n)$.

By completing these exercises, you will learn to:

- Perform sorting operations on simple lists and arrays of numerical and character data.
- Sort complex objects that contain multiple different types of data.
- Define and use multiple custom sorting criteria, by using Comparators.

  ⇒ **Remember to ask your tutor if you get stuck or have any questions.**

9.1 Sorting Arrays and the Arrays Class

Arrays are common when storing collections of data and their indexing system means that they are typical data structure that we want to sort. As we've seen already, primitive arrays can be used to store a variety of data types, such as holding numbers, string, and characters. An array allows for efficient access to individual elements, making them a typical choice for data structures that need to be sorted. Sorting arrays can be achieved in a variety of ways, by either implementing our own sorting algorithm or utilising the features of the Arrays utility class, which has a variety of useful static methods.

Exercise 9.1.1

Write a new program called **Scores** that prompts the user to enter scores from a panel five judges in a dancing contest. Scores are integers between 1 and 10. The scores should be stored in a suitable collection, such as an array. Sort the scores and display the smallest and largest scores to the terminal window by referencing the *first* and *last* elements of the sorted list.

Once an array (or similar kind of list) has been sorted according to some *comparable* criteria, it makes it easier to find specific, or ranges of, data. The index of a sorted array will follow an established order, which may be especially advantageous if performing statistical work with numeric values. Sorted arrays are also easier for human users to understand and work with.

Exercise 9.1.2

In a 200-metre running race, you have recorded the finishing times of all ten participants. These times are:

29.47, 27.25, 31.73, 27.22, 28.46, 33.51, 29.38, 29.65, 31.82, 26.96

Write a Java program named **Run200** to find and display the runner-up (second-fastest) participant by sorting the times.

The `Arrays` class is a particularly useful expansion upon the basic array that we've tended to use so far in Java. It has a range of built-in methods that can be used for array operations, including *sorting* and *searching*, which have customisable options, depending on what result you are aiming for.

◯ Assessed Exercise ◯

Create a simple program that contains an array of 25 randomly generated integer values. Your program should sort the array using the built-in sorting algorithm (`Arrays.sort()`). Display the array before and after sorting to the console. Your program should be called **NumberSort**.

9.2 The Comparable Interface

When working with our own classes, such as those that hold data about objects our programs work with, we may need to applying sorting. This is especially useful when we have classes that contain multiple attributes and those attributes may be of different types. The Comparable interface can be implemented by these classes and allows us to define a natural order of these objects, according to our own specifications, so that they can be sorted (or searched), typically by the `Collections.sort()` method. Objects that implement Comparable can be compared to each other, and their relative order can be determined by overriding the `comapreTo()` method from the interface and passing it an object, and type, that a current instance should be compared with.

◯ Assessed Exercise ◯

Create a class called **Student** with attributes of *name*, *age*, and *grade*. Implement the *Comparable* interface to enable sorting students by their grades in descending order. Demonstrate sorting a list of students using `Collections.sort()` in a program called **GradeRank**. Your program should also sort the list of students using either a *bubble*, *selection*, or *insertion* sort (your choice). Compare the time complexity of the two sorting algorithms, with evidence and/or graphs as appropriate, and provide a commentary on which is more efficient.

Sometimes you will want to sort according to orderings that are defined in relation to the situation or some established set of ordinal rules, which don't adhere to natural numeric or alphabetical conventions. For instance, if we wanted to sort a collection of t-shirts in a shop according to their size, we would want this to follow the order: *Small (S) – Medium (M) – Large (L) – Extra Large (XL)*, but it's clear that this convention is neither alphabetical nor numerical.

Exercise 9.2.1

Write a simple class named **Coder** that can be used to hold information about the software developers in a project team. The class should hold two fields of information about a developer: (1) their *name*; and (2) their job *grade*. You should then create a new program named **DevTeam**. In this program Populate an *ArrayList* of **Coder** objects with the following information, in this sequence:

Carmelo Braun (Senior Developer)
Donald Foster (Middle Developer)
Rachael Holmes (Principal Developer)
Junaid McCarthy (Junior Developer)
Victor Martinez (Middle Developer)
Gale Gregory (Senior Developer)
Amparo Mendoza (Junior Developer)

Finally, produce another class called **JobSort** that implements a custom *Comparator*, which will allow you to sort the *ArrayList* of **Coders** *according to their job grade*. For reference, the desired ordering should begin with Principal Developer and end with Junior Developer, as follows:

Principal Developer
Senior Developer
Middle Developer
Junior Developer

HINT

Applying an 'internal' numeric allocation of non-numeric, ordered items for a program to follow can help it compare items and ensure they adhere to the desired sequence.

9.3 The Comparator Interface

The previous `Comparable` interface is a useful way to implement a specific sorting behaviour for our objects, which is part of the object class and defines a *natural* sorting behaviour. However, there are often scenarios where we may want to be able to sort our data in different ways, depending upon use case. For example, if we have an art collection that contains information such as: title of artwork, artist, year, and value, we might wish to present information about this collection sometimes by year and at other times by artist. To perform an operation such as this, we can create multiple classes that implement the *Comparator* interface, each of which overrides the `compare()` method, and apply any one of these classes when sorting, such as using `Collections.sort()`.

Exercise 9.3.1

Design and produce a **City** class with attributes of: *name*, *country*, and *population*. Write a program named **Cities** that makes use of two custom *Comparators* and shows how a selection of cities can be sorted in different ways. The cities to include as instances of the City class are:

Delhi	India	28514000
Glasgow	UK	635180
Tokyo	Japan	37468000
Lahore	Pakistan	11738000
Suzhou	China	6339000
Lagos	Nigeria	8049000
Toronto	Canada	6082000
Bangkok	Thailand	10156000

One of your custom *Comparators* should sort the cities in ascending alphabetical order (A-Z) and the second should sort the cities in ascending order by their population. Your **Cities** program should output both sorted collections to the terminal.

◯ Assessed Exercise ◯

Extend the **Student** class from the earlier exercise to include a *faculty* attribute (faculties can be one of: arts; business; education; health; law; or science). Write a custom *Comparator* that sorts **Student** objects first by faculty (in ascending order) and then by grade (in descending order). Use this *Comparator* to sort an array of **Student** objects in a program named **FacultyGradeRank**. This program does not need to implement an alternative sorting algorithm.
When completed, your program should look something like this:

```
week-09-code % java FacultyGradeRank
Unsorted students:
NAME     AGE     GRADE     FACULTY
Adam     20      32        Health
Ben      22      87        Arts
Cindy    21      65        Business
Daniel   19      43        Engineering
Emily    22      91        Science
Frankie  21      16        Education
Gemma    19      29        Law
Hilary   23      40        Science
Iwan     22      88        Arts
Jake     21      58        Science

Sorted Students:
NAME     AGE     GRADE     FACULTY
Iwan     22      88        Arts
Ben      22      87        Arts
Cindy    21      65        Business
Frankie  21      16        Education
Daniel   19      43        Engineering
Adam     20      32        Health
Gemma    19      29        Law
Emily    22      91        Science
Jake     21      58        Science
Hilary   23      40        Science
```

HINT

When sorting according to multiple criteria, the sequence that this happens in is important. Once any comparable difference is identified based on the first criteria then your program can decide if the second criteria should be applied.

Creating your own implementations of *Comparator* allows your code to adapt to specific problem-based needs and to develop modular solutions. As you tackle scenarios that require custom sorting criteria, you develop problem-solving skills that are applicable across various domains, not just Java.

Week 10

Testing with JUnit

Testing is essential when it comes to developing software. As a minimum, the various functions, and features of the programs that we write need to be operational and correct. So far, we've adopted the approach of testing programs ourselves, by hand, as we write code, which has worked well, but becomes repetitive, especially when code is frequently changed. As programs grow in their scale and complexity, manual testing starts to become time consuming and tedious. For this reason, automated testing of code is beneficial, which introduces us to such a framework for this: *JUnit*.

JUnit, is a popular testing framework and offers efficiency, reproducibility, and can be applied and utilised in Java projects from small to large. Its annotation-based testing simplifies test organisation, making it a valuable tool for maintaining code quality in Java development. The exercises here are intended to give a general introduction to unit testing and the JUnit framework and will help you apply basic automated testing in your programs. These exercises focus on the use of JUnit 5, although the principles, and much of the syntax, are compatible with earlier versions, notably JUnit 4.

By completing these exercises, you will learn to:

- Create annotated JUnit tests of equality and Boolean types.
- Produce JUnit tests for scenarios where exceptions may be thrown.
- Generate parameterised JUnit tests.

⇒ **Remember to ask your tutor if you get stuck or have any questions.**

10.1 Testing for Equality and Truth

Use of the `@Test` annotation and assertion statements form the backbone of a lot of unit testing with the JUnit framework. These tools facilitate the interrogation and examination of various elements within the code to ensure that their behaviour aligns with anticipated expectations and specific requirements. Adding `@Test` to your code marks a method as a test case, allowing JUnit to recognize and execute it during testing procedures.

Of all of the assertion statements that JUnit offers, the `assertEquals()` method is one of the most useful - providing a way to see if a value or property matches what is expected. This is particularly valuable

in scenarios where numerical results, string values, or object states need to be verified.

Exercise 10.1.1

Write a program in Java named **Double**, which contains a method that accepts an integer as a parameter, doubles that value, and then returns it. Create a JUnit test that verifies the conduct of the program under a range of circumstances, such as when positive, negative, and zero values and passed.

◯ Assessed Exercise ◯

Implement the program named **BMICalc** (one of the assessed exercises in Week 2) that takes a person's weight (in kilograms) and height (in meters) as input and calculates and outputs their BMI. Use the float data type for weight and height input, and a double type for the BMI calculation and result. Create Junit test(s) to verify correct operation of the calculation and provide evidence of this status.

The versatility of the `assertEquals()` method lies in its overloaded nature, enabling it to dynamically adjust its actions based on the types of variables and objects being compared. This makes it suitable or testing not just numerical values, but also strings, arrays, and other Java objects. The ability of `assertEquals()` to cater to a spectrum of data types fosters a more thorough examination of code, ensuring that methods and functions perform as intended across various scenarios. In essence, this adaptability contributes to the efficiency and effectiveness of unit tests in the broader context of Java development.

Exercise 10.1.2

Write a program named **HeyYou**. The program should ask the user to enter their name via the terminal window and then output the message "Hey <name>". The **HeyYou** program should contain a method that is passed the user's name as a string parameter and then return another string containing the complete message. Write a JUnit test to ensure that this method operates as expected.

◯ Assessed Exercise ◯

Write a program named **Palindrome** that contains a method to check if a given input is a palindrome (a sequence that means the same thing read left-to-right, as it does when read right-to-left. For example: "radar"). This should function for alphanumeric text input. Include JUnit tests to verify the accuracy of the palindrome-checking method.

In addition to `assertEquals()`, the method `assertTrue()` may also be encountered in unit testing. This method becomes particularly relevant when dealing with variables or objects that encapsulate Boolean attributes—attributes that evaluate to either true or false. The `assertTrue()` method works in a similar way to `assertEquals()` but, as its name suggests, the method always expects a true status to be identified. Its application strengthens the suite of tests, allowing developers to assert and validate the truthfulness of conditions in their code.

Exercise 10.1.3

Create a program called **Greater100** containing a method that receives a double value as a parameter. The method should return true if the given parameter is greater than 100 and false under any other circumstances. Produce a JUnit test for the method to verify its behaviour, paying particular attention to values around the threshold of 100.

10.2 Testing for Exceptions

Testing for exceptions helps programmers verify that atypical actions in programs throw expected exceptions. Furthermore, if exceptions do occur it provides a mechanism by which exception handling can be verified. JUnit has methods for determining if exceptions will be thrown, under specified conditions. One such method is `assertThrows()`. As with the other assertions, `assertThrows()` follows a format of allowing the tester to specify the type of exception that we think will be thrown and the situation or code that may lead to this.

Exercise 10.2.1

Write a program named **LongList**. This program should contain a method that accepts an integer as a parameter, which is then used to create, and return, an array of the same length. This should be an array of strings that contains the '*' symbol. Write a JUnit test that demonstrates that attempts to use an array index greater than the array length with throw an `ArrayIndexOutOfBounds` exception.

In larger, more complex, programs it's likely that a variety of exceptions could be thrown. As such, it's common to develop a suite of JUnit tests that make use of `assertThrows()` to evaluate the range of operations that may trigger them. it's good to remember that `assertThrows()` can be used with any type of exception, including custom exceptions.

◯ Assessed Exercise ◯

Develop a class named **FileManager** containing methods for reading and writing simple text files. Write JUnit tests to verify that files are read and written correctly and that exceptions are handled appropriately and provide evidence of the testing.

HINT

Whilst JUnit test scripts look and feel a little different, they are still Java programs. So, we can use normal Java syntax as part of our testing processes.

10.3 Parameterised Testing

In many situations, especially if we have written code segments that perform mathematical operations, we will want to test the functionality across a *range* of multiple values. This might include a sample of values between a minimum and maximum as well as being oriented around threshold values, such as might be encountered in code that makes use of `if-then-else` or `switch` statements. This can be handled in JUnit using the `@ParameterizedTest` annotation, often in combination with a range of variables that below to a variable defined in a `@ValueSource` annotation. The range of parameters can then be used

as an input for a JUnit test method, where test methods, such as `assertEquals()`, will use this set of values.

Exercise 10.3.1

Write a new Java program named **EnergyConv**, containing a method that receives a given double parameter representing a value of energy (in Joules) and will return the value converted to *Watt-hours*. Produce a parameterised JUnit test of this method with the following input parameters:

180.5, 1500, 5400.22, 9670, 14850.45, 64500, 1204500.25

HINT

Watt-hours = $Joules/3600$.

Appendix A

Challenge Exercises

These exercises are designed to stretch and extend what you have been taught by your own independent learning. They are likely to require more time and additional research to complete than the weekly exercises.

◯ Assessed Exercise ◯

Write a program named **ComplexMagnitude** containing three methods that demonstrate a computational operation on a set of data that have a time complexity of $O(n)$, $O(n^2)$, and $O(n^3)$ respectively. Demonstrate this is the case by evaluating your program with a range of values of n and plotting the times taken in a spread-sheet graph, which should be included alongside the program in your portfolio.

◯ Assessed Exercise ◯

Create a recycling game where items (made of plastic, paper, glass, food, etc.) are presented randomly, and the player must sort them correctly into the right bin. This program should be called **RecycleGame**. You can decide what the bins are called and what items belong in which, but there should be at least four bins. The difficulty should increase with each round: adding more items, limited sorting time, and trick items (e.g., something that looks recyclable but isn't). Players need to achieve a particular score in order to progress from one round to the next and you could make this score more challenging to increase difficulty too. Integrate a modular leader-board, where players can record their name and high scores for comparison.

◯ Assessed Exercise ◯

Develop a program entitled **CestrianGUI** that features a Graphical User Interface (GUI) front-end and deploys the functionality of the **CestrianInsurance** program (seen in the Week 3 assessed exercises). The GUI should feature a clickable selection field for the vehicle type (e.g., a dropdown menu or radio button), a checkbox to confirm if the customer is under the age of 25, and a validated numeric input field for the number of penalty points. Make effective use of object-oriented design principles and patterns.

◯ Assessed Exercise ◯

Write a class named **DeltaCompression** that performs delta compression upon a given sequence of input. The encoding method should accept sequences that are either exclusively numbers or characters (a `String`). In the case of characters, delta values should correspond to an index value using the standard English language alphabet. The class should also have a complementary method to decode a given input to its uncompressed form. Both methods should be `static`, allowing their use to be demonstrated from a separate program named **DeltaCheck**, without **DeltaCompression** being instantiated.

◯ Assessed Exercise ◯

Develop a cat adoption system that utilises object-oriented programming principles and appropriate design patterns. The system should allow users to view available cats for adoption, add cats to the system, and adopt cats. Each cat should have a name, age, breed, and description. Users should be able to search for cats by breed and age. The system should also keep track of the adoption status of each cat and prevent multiple adoptions of the same cat. Additionally, the system should allow for the management of cat records, including adding new cats and updating existing ones. The main program that runs the system should be named **CatHome**.

◯ Assessed Exercise ◯

Implement a parallel sorting algorithm using multi-threading. Divide a one-dimensional array (of at least 100 elements) into smaller subarrays and sort them concurrently using threads. Merge the sorted subarrays to obtain the final sorted array. Compare the time complexity / efficiency of this approach with sorting the array using a non-threaded approach and plot the times taken in a spreadsheet graph, which should be included alongside the program in your portfolio. Name this program **ParallelSort**.