Introduction to Java and Maven

Syllabus and course notes

Morgan Walsh

June 6, 2023

Contents

Part I

Syllabus

1 Course overview

This course will introduce the Java programming language and how we can use the Maven build tool.

Course title	Introduction to Java and Maven
Course length	5 days
Course area	Programming & Software Development
Language	English (GB)
Required equipment and software	Laptop (Windows 10/11), IntelliJ Community Edition, Java 17 JDK, Maven
Course materials	MS PowerPoint presentations, Exercise guide and Trainer handouts
Pre-requisite knowledge	You will be expected to have basic computer literacy and a high-school knowledge of mathematics.

Course materials will be handed out on the first day of the course.

2 Schedule

Day	Morning Session	Afternoon Session
Monday	10:00 - 13:00	14:00 - 17:30
Tuesday	9:00 - 13:00	14:00 - 17:30
Wednesday	9:00 - 13:00	14:00 - 17:30
Thursday	9:00 - 13:00	14:00 - 17:30
Friday	9:00 - 12:00	13:00 - 15:30
Saturday	N/A	N/A
Sunday	N/A	N/A

⋆ Times may differ for your course!

3 Objectives

By the end of the course, you:

- Will understand the basics of Java programming and how to apply code to solve problems
- Develop applications in Java using the IntelliJ IDE
- Should be able to apply object-oriented programming techniques to modularise your applications, reduce duplication and solve problems
- Should be able to discuss the concepts of functional programming and apply functional programming to solve basic programming problems

4 Course outline (topics covered)

1. Introduction to Java
2. Basic Java syntax
3. Flow of control/control flow
4. Introduction to Object-Oriented Programming (OOP)
5. Inheritance
6. Interfaces
7. Collections
8. Exception handling
9. File handling
10. String handling and manipulation
11. Introduction to Functional Programming (FP)
12. Introduction to streams
13. JavaDocs (Program documentation)
14. Introduction to Maven
15. Packaging applications with Maven as distributable .jar files
16. Introduction to Test Driven Development (TDD) with Maven

5 Timetable

Days of the course are numbered **1** through **5** as there is no guarantee that the course starts on a specific day of the week, for instance when a week begins with a bank holiday.

Day	Topics	Resources	Assignments
1	Introduction to Java, Basic Java syntax, Control flow	MS PowerPoints & exercise guide	Class notes, participation in labs
2	Introduction to OOP, Inheritance, Encapsulation, Collections	MS PowerPoints & exercise guide	Class notes, participation in labs
3	Exception handling, File handling, String handling and manipulation	MS PowerPoints & exercise guide	Class notes, participation in labs
4	Abstraction, Polymorphism, Interfaces, Introduction to FP, Introduction to streams	MS PowerPoints & exercise guide	Class notes, participation in labs
5	JavaDocs, Introduction to Maven, Packaging applications, Introduction to TDD	MS PowerPoints & exercise guide	Class notes, participation in labs

⋆ Ordering and inclusion of topics may differ dependent on course-specific requirements and instructors delivery and lesson plans.

6 Student requirements

As the student, there is an onus of responsibility for your learning. This section highlights these responsibilities.

6.1 Attendance requirements

It is important for your instructor to be notified at the earliest convenience that you will not be able to attend any session(s).

It is understood that you may have existing pre-booked appointments or emergencies, but repeated episodes of tardiness outside of this realm will be escalated to the relevant team members at QA.

⋆ Additional requirements for attendance in terms of reporting may be in place for your course!

6.2 Catching-up with missed work

In the case that you miss some of the course, it is expected that you (the student) inquire with the instructor about work for catching up with classroom and exercise sessions.

The instructor will provide support where possible in helping you catch-up with the classroom material, but there can be **no expectations** that the instructor will be able to *re-teach* the material.

7 Instructor responsibilities

1. At the beginning of the course, the instructor will provide the course syllabus and materials required to each student.
2. The instructor will evaluate each student's participation and technical skills throughout the course.
3. Accurately record each student's attendance throughout the course.
4. At the instructor's discretion, additional work and learning may be implemented where plausible.

Part II

Course notes

The following pages contain notes to accompany your study of Java and Maven.

8 Introduction to Java

The objectives for this section include:

- Be able to describe what a **program** is and why we use them
- Be able to describe what an **algorithm** is and why we use them
- Be able to describe what Java is and understand how the JVM, JRE and JDK are intertwined
- Be able to apply your Java knowledge to create, compile and execute the canonical introductory program, *Hello World!*

In this section, we aim to cover:

8.1 What is a program?

A **program**, in this case a contraction of *computer program*, is defined by Wikipedia as:

> A **computer program** is a sequence or set of instructions in a programming language for a computer to execute. Computer programs are one component of software, which also includes documentation and other intangible components.
>
> Wikipedia 2023, Computer Program

The key thing here is that a program is a series of instructions which are executed, in order and one-by-one, by the computer. Take for example, navigating to a website in your browser. The web browser is a program designed to *surf* the World Wide Web (WWW), when you enter the URL of a website your browser will then execute a series of instructions in response:

1. Verify if search input is a valid URL or search term
2. If search input is a valid URL:
   2.1 Navigate to URL
3. Else:
   3.1 Navigate to search engine results

These instructions are of course simplified, but do represent a sequence of instructions executed by a program in response to input. A small sequence of steps like this could be referred to as an *algorithm* (discussed later)

8.2 What is programming?

Programming is the process through which you create a *program*, said program being created through writing source code. **Source code** is a human-readable version of the instructions that will eventually be executed by the computer.

Another popular term is **coding**, which represents the process of writing *code*. At first glance, they don't seem very different... Programming keeps the whole project in mind when creating a program whereas coding doesn't require you to have a project/program in mind. Coding is also the process of writing source code when programming.

8.3 What are algorithms and how do they apply to programming?

An **algorithm** is a sequence of instructions carried out to solve a problem. For this course, this is all we need to know but read on if you wish for some more information...

In the realms of Computer Science, algorithms involve asymptotic analysis - a practice from mathematics in which we calculate the general behaviour of an algorithm given a variety of increasing input sizes. The behaviour we discover represents the runtime boundaries of an algorithm. To explain further, lets use an example.

When a computer executes an algorithm, each algorithm has a *time complexity* - how many steps the algorithm will take, and how the amount of those steps grow, given ever-larger input sizes. Adding two numbers is a very simple algorithm:

```
1 // adding two small numbers
2 4 + 4
3 // takes the same amount of time in hardware as two bigger
     numbers
4 3000 + 45000
5 // subtraction also takes the same amount of time
6 65_555 - 1000
```

It doesn't matter how big the numbers are, the process of adding two numbers in the hardware remains the same in the general amount of time it takes. Instead of representing this as time though, we say it is an operation with **constant time** complexity. It is often represented as $O(1)$, which is read as *big-oh of 1* or *constant-time complexity*. It is simply a way of saying: Regardless of the input size, the algorithm will take the same amount of time.

> Due to how the basic mathematical operations are implemented at a hardware level, in binary, the addition, subtraction, multiplication, division and exponentiation operations are all considered to take constant time.

For our next and final example on this topic, let's consider adding a list of numbers together - a `sum` operation expressed using prefix notation:

```
1 (sum 20 30 40 50 60)
```

This operation will add each of the numbers together. If we consider that each ad-

dition is a constant time operation, it will take 4 additions to sum the above numbers and get the result: `20 + 30 + 40 + 50 + 60`. Now lets consider `(sum 20 30)`, this would only take 1 addition. How about `(sum 20 30 40)`, this would take 2 additions. What we can conclude from this is that the amount of steps (additions) the algorithm (`sum`) takes will vary dependent on its *input size*, the input size being the amount of numbers to be summed. This reveals something else, hidden beneath the numbers. Every time we increase the input size by 1 (the amount of numbers), the amount of steps the algorithm takes also increases by a constant time - of 1 in this case, but the steps could vary dependent on the algorithm, the key here is that the increase is of a constant time. We call this **linear time complexity**, often represented as $O(n)$ and read as *big-oh of n*. In this case though, as we are using a list of numbers which may vary in size, we need to represent that it depends on its size. So n can be seen as the value of something, and $|n|$ can be seen as representing the size rather than value (the amount of numbers in the list in this case). Now, we can say that the `sum` algorithm has a linear time complexity or `sum` has a runtime complexity $O(|n|)$ - the amount of steps this algorithm grows linear in proportion to its inputs size rather than value.

Here is a simple implementation of the `sum` algorithm in Java:

```
1 int sum(int[] nums) {
2     int sum = 0;
3     for (int num : nums) sum = sum + 1;
4     return sum;
5 }
```

8.4 Why do we use steps instead of time for measuring an algorithms performance?

It's simple really... Consider the oldest computer you have ever seen and/or used and then consider the newest and best computer you have ever seen. The old computer will likely have a slower central processing unit (CPU), amongst other components, compared to the newer one. This would be an unfair way to measure the general performance of an algorithm as the older computer could run it slowly, but the newer computer much faster. The idea is to abstract away this complexity, as even just running a simple computation such as adding the same two numbers twice in a row on the same computer could drastically vary in the amount of time they take - this isn't because addition is slow, but because your computer has additional overheads such as the kernel and operating system which manage your running programs. This is why we say addition takes 1 step

regardless of the computer it is on (in general), rather than something like the following conversation:

> **Bob**: "Adding two numbers took 0.00003455 seconds on my computer"
>
> **Fred**: "I've got the same computer as you Bob, it took 0.000008934 seconds for me"
>
> **Bob**: "That's crazy, how was yours so much faster. There must be something wrong with my computer"
>
> **Sarah**: "Mine took 0.0454343 seconds, maybe there is something wrong with mine too"
>
> **Ayisha**: "My addition operation took 0.000004777 seconds, maybe having more things open affects it?"
>
> ...

If we had to argue about why something took **x** amount of seconds for every piece of hardware, algorithmic analysis would be a snail-pace of a chore. Instead, we use the abstraction of steps...

8.5 Is there an intermediate representation for algorithms for notes?

Yes, we often represent algorithms using psuedo-code, a fake code which highlights the desired computations, and then implement them in a variety of languages. The `sum` algorithm in Java could be represented in **pseudo-code** as follows.

Assume *nums* is the input list of numbers:

1. set *result* to 0
2. for each *num* in *nums*:
   2.1 set *result* to *result* + *num*

The "set *x* to *value*" is used to set a variables value, a variable just being a label for a value. The first instruction, *1*, sets a variable with the label *result* to the value of 0. The second instruction represents an iterative statement, "for each *value* in

iterable", any instructions indented under this (such as 2.1) will be repeated as many times as there are values in the the iterable.

8.6 Programming paradigms

A **programming paradigm** is a classification for a set of features which a programming language incorporates. There are two main categories of classification:

- **Declarative**: A declarative paradigm indicates that you declare the properties of the results you want, rather than specifically how to get the results.
- **Imperative**: An imperative paradigm indicates that you are instruction the computer exactly how to get a desired result.

To highlight the difference, we will represent two ways of getting only the even numbers in a list of numbers using Java. First, the imperative way:

```
1 List<Integer> nums = List.of(1, 2, 3, 4);
2 List<Integer> evens = new ArrayList<>();
3
4 for (Integer num : nums) {
5     if (num % 2 == 0) evens.add(num);
6 }
```

This is quite verbose, as in there is a lot of code. Now let's look at a declarative way using Java:

```
1 List<Integer> nums = List.of(1, 2, 3, 4);
2 List<Integer> evens = nums.filter(num -> num % 2 == 0);
```

Don't worry about the code itself, but just consider for now how much shorter the declarative code is. The key difference comes down to how we are getting the even numbers. In the first example (imperative), we use a `for` loop to explicitly check every number in the `nums` list to see if it is even, before then adding it to the `evens` list if it is actually even. The second example skips all that, we instead just apply a `filter` to `nums` and get back the list of even numbers without having to do all the checking and adding to a new list ourselves.

As you might have figured out then, Java is a multi-paradigm language which has aspects of both imperative and declarative programming. Java's predominant and primary programming paradigm is **object-oriented programming** (imperative paradigm), a way of writing programs where its data and behaviours are represented as objects which can be *passed* around the program. The declarative

functional programming paradigm has been implemented in Java (from Java 8+) on top of Java's object system, with *lambda* functions as the representation - these are implemented through functional interfaces (explored later).

8.7 What is Java?

Java is a general-purpose, object oriented, platform independent, simple, and robust programming language. You may have heard the motto write once, run anywhere, or something similar, meaning that a Java program can be written once and compiled once to run on all platforms that support Java.

8.8 Compilers and interpreters

To understand how Java works, we need to understand what compilation and interpretation mean in the context of software development. Compilation is a process in which the work of converting source code, the code written in the programming language, to the machine language which can be understood by a digital processor – the name for this machine language is binary, a simple language consisting of 0s and 1s that is notoriously complex to work with directly.

- Compilation models do often have an intermediary language of some form to act as an abstraction between the source code and the machine code, this can be a language called assembly which works directly with the hardware to produce the required sequences of 0s and 1s or a bytecode style language which operates similarly to assembly, bytecode languages are then usually translated into machine-level assembly code.
- Compiling source code to intermediary languages often benefits the development of the language itself as developers can use pre-existing tools that are simpler to use than raw binary code
- We can say that source code is fed into a compiler which outputs the compiled form of the source, which is usually in the form of an executable file

A simple abstracted compilation model which translates instructions into some intermediary language can be seen in figure 8.1. On the left, the start of the compilation model, is the source code, which is compiled into an intermediary language that can then be interpreted into machine code by some interpretation software. An interpreter is different from a compiler, interpretation models dictate that source code is executed and converted to machine code as and when it is needed rather than being converted all at once.

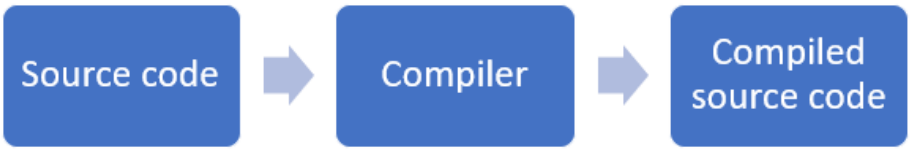


Figure 8.1: Simplified compilation model

> Interpreters may work with both compiled source code or direct source code, it depends on the implementation and the language.

What a simple abstracted interpretation model might look is represented in figure 8.2. Here, the source code is converted into binary machine code, 0's and 1's.

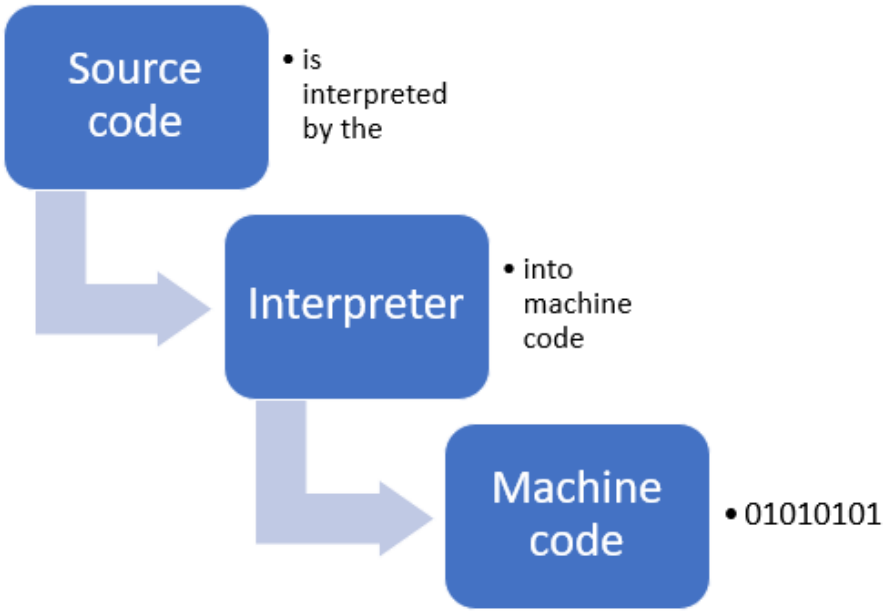


Figure 8.2: Simplified interpretation model

8.9 How do compilers and interpreters fit into Java?

Java is a language which goes through both compilation and interpretation, that is Java is traditionally known as a compiled language but does also include interpretation at a lower level.

Java applications run using a special piece of software called the Java Virtual

Machine (JVM), this software contains a bytecode interpreter and a just-in-time compiler (JIT compiler) which enables platform independence as all we need to run the code is a JVM on our machine. To run a Java app, we first must compile its source code into an intermediary language called bytecode; we can compile Java source code into bytecode using the Java Compiler (javac), a tool included in the Java Development Kit – a set of software required for developing Java applications (figure 8.3).

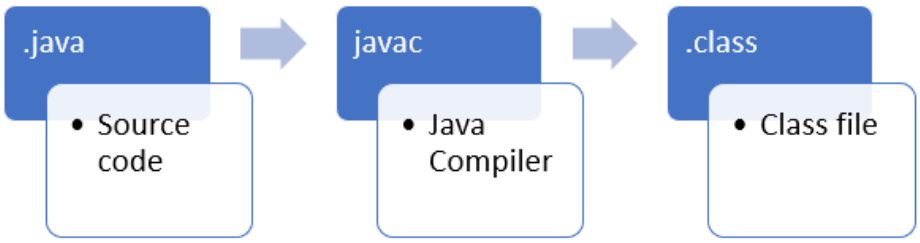


Figure 8.3: Compiling Java source code

The Java compilation model is demonstrated in figure 8.3, a .java file contains our source code which is fed into the Java compiler. The compiler then outputs a compiled .class file containing the compiled bytecode. This illustrates how we create compiled files, but the bytecode inside them has not yet been executed.

To run compiled bytecode, we must feed the class files into the JVM and launch it using the java launcher included in the JDK (figure 8.4). The Java bytecode (.class) files are fed into the Java Virtual Machine, the JVM is then capable of interpreting the bytecode into the correct machine code dependent on the platform it is running on. The full compilation and interpretation model is demonstrated in figure 8.5.

Just-in-time compilation is a process the JVM carries out to optimise code at run-time after the code has been compiled. The JVM is designed to look for frequently executed chunks of code when running, if a frequently executed block of code is found it will be compiled into native code to reduce performance deficits caused by frequently interpreting the same bytecode – essentially, compilation is almost always faster than interpretation for frequently used areas of code.

8.10 The benefits of Java

- Encapsulation: Java includes keywords for protecting data from unintended access or modification.

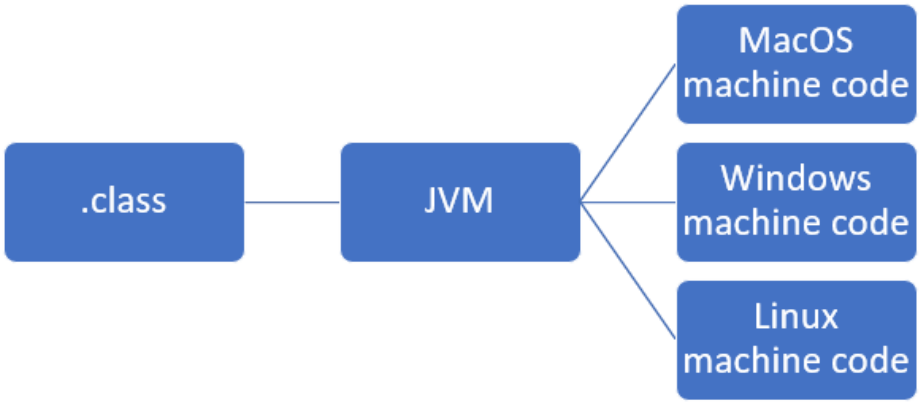


Figure 8.4: A model demonstrating cross-platform behaviour

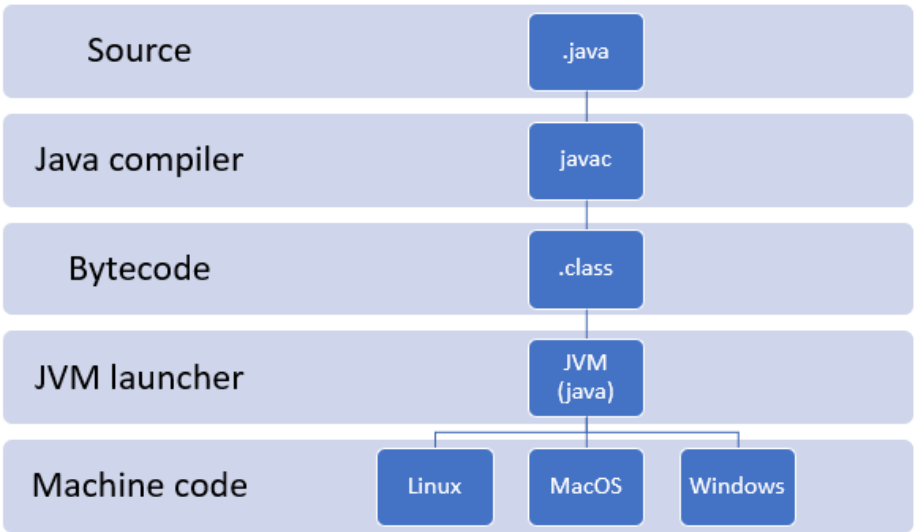


Figure 8.5: Full Java compilation and interpretation model

- Platform independence: Java's compilation and interpretation models allow for it to run on a wide array of platforms; essentially any platform that a Java Virtual Machine (JVM) exists for.
- Robust: The JVM automatically manages system memory and garbage collection, a process which prevents many mistakes common in languages like C from being made.
- Object oriented: Java uses the object-oriented programming paradigm, with the inclusion of functional behaviours from Java 8 onwards, to organise code.
- Secure: The JVM is a sandboxed environment which makes it more difficult for malicious code to be executed on a system.
- Backwards compatibility: The Java language has been designed with backwards compatibility in mind, meaning that each new version of Java is very likely to work with software written in older versions of Java.
- Multithreading: Java allows for multiple instructions to be executed at the same time using threads.

8.11 What is the JRE and JDK?

The Java Development Kit is a downloadable collection of software tools for developing Java applications including:

- Java compiler (javac)
- JVM launcher (java)
- Archiver (jar)
- API documentation (Javadoc)

Prior to Java 11, there was two parts to Java – the Java Runtime Environment (JRE) and the JDK. This allowed for people to download the JRE and run applications without requiring the tools included in the JDK to be downloaded on their system. From Java 11 onwards, the JRE is part of the JDK and thus everyone gets the access to the tools whether they use them or not.

> Commonly, developers use an **Integrated Development Environment** (IDE) to develop software rather than using the JDK tools directly as the IDE abstracts the process of using these tools. Two popular choices of IDE are IntelliJ and Eclipse.

8.12 Installing the JDK

Download the JDK from Oracle or OpenJDK (open-source equivalent). Once it is downloaded, move the downloaded folder to a secure place on your file system. A `JAVA_HOME` environment variable needs to be set so that we can use Java from the command-line. On Windows, type the following into CMD:

```
setx JAVA_HOME "C:\Program Files\Java\jdk"
setx PATH "%PATH%;%JAVA_HOME%\bin";
```

Change the `JAVA_HOME` path to match the root directory of the JDK.

`setx` is the only Windows terminal command capable of permanently setting user and system environment variables. The above usage sets the users environment variables by default. To make it system wide, pass the `/m` flag to `setx`:

```
setx /m JAVA_HOME "..."
```

8.13 IntelliJ introduction & installation

Your course instructor will walk you through installing and setting up IntelliJ. You can also download and set it up from JetBrains | IDEA.

8.14 Your first program in Java, Hello World!

A commonality amongst programmers is that in any new language we meet, we write a `"Hello World!"` program – a simple application which will output those exact words to the screen. To follow along, open a terminal application and then create a new file called `App.java`. In most Unix-like environments (such as Linux and macOS), the following commands will create a new folder/directory (`mkdir`) in your user `home` directory, change into the new directory (`cd`), create a new a file (`touch`) in that directory called `App.java` and then list out the contents of the current directory (`ls`):

```
mkdir ~/java_programs/hello_world
cd ~/java_programs/hello_world
touch App.java
ls
```

Listing 8.1: Creating the App.java file | Linux/GNU Shell commands

If you are using Windows, use the following commands instead:

```
mkdir %USERPROFILE%\java_programs\hello_world
cd %USERPROFILE%\java_programs\hello_world
fsutil file createnew App.java 0
dir
```

- `mkdir [<drive>:]<path>`: mkdir documentation
- `fsutil file create <filename> <size_in_bytes>`: fsutil documentation
- `%USERPROFILE%`: Represents the path of the users home directory, including drive name. For example: `C:\Users\morgan` would be my value of the `%USERPROFILE%` variable.
- `dir`: Used here to display the contents of the current working directory: dir documentation

See this post for additional information on user paths in Windows. Is there a shortcut command in windows command prompt to get to the current-user

> If you created the wrong file by accident, you can use the `rm` command on Linux systems (Documentation) or the `del` command on Windows (Documentation). If you would like to view the contents of a file on Linux, use the `cat` command (cat Documentation) or the `type` command on Windows (type Documentation).

Once we have created the file, type `notepad Runner.java` to open the file in the Notepad application on Windows or use `nano Runner.java` to use the nano text editor in the console (Git Bash on Windows only, usually preinstalled on Linux/GNU based systems). Then type in the following code:

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

```
Hello world
```

Once you have entered the code, save, and exit back to the console. Then type the following commands (except comments starting with a hash symbol) to compile and run our code:

```
# compile the Runner.java file as a class file using the javac
    tool
javac App.java
```

```
# run the compiled class file (which is in the same directory)
    using the java tool
# - do not specify .class or the compiled file will not be
    executed
java App
```

Adding `.class` to the java Runner command will produce a class loader error, where the JVM was not able to load the specified class file. If done correctly, you will see the text `Hello world` printed to the terminals display via standard output.

8.15 A breakdown of Hello World

A lot is going on in the background to make our program run, we are already aware of this; now it is time to investigate the structure of how we write our programs.

Java is an object-oriented language, it uses the construct of a class to represent a component in a software system – a component could be the representation of physical things like a customer, an animal, a tree or even non-tangible logical things like mathematical formula or graphs. A **class** is a blueprint which defines the data and behaviours that describe some object. In the Hello World example, we create a class called `App`:

```
public class Runner {

}
```

- The first line declaring the class, `public class Runner`, is known as the **class header**.
- `public` is a special Java keyword known as an **access modifier**, just know that for now it means that anyone can access and use the class called `App`.
- `class` is another Java keyword used to signify that we are creating a **class**, a blueprint.
- `App` is the name of the class, we can choose a name other than `App` if we like but it must match the file name (excluding file type).
- The curly braces signify the **body** of the class, this contains code that *belongs* to the class.
- code outside of a class is an **error** in Java, Java code must live inside a class aside from package declarations, imports, and type declarations.

Inside the class body, there is a special unit of code known as a method:

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

A **method** is a self-contained, reusable block of code that can be called/invoked by another method. The `main` method shown above is a special type of method that indicates to the JVM that the program should start here.

- `public static void main(String[] args)` is the **method header**, this identifies the methods accessibility, output, name, and inputs.
- All of the code between the curly braces after the method header is in the **method body**, this is the code that is executed when a method is invoked and belongs only to that method.

Complete instructions of code in Java, aside from blocks of curly brackets, must end with a semi-colon to be valid. The semi-colon used to terminate a programming statement is known as a delimiter. A block of code is represented lexically (we can see it) using pairs of curly braces, our program must always have a balanced set of braces in each file. `{}` is balanced, `{{}` is not balanced, `{{}}` is balanced, `}{}{` is not balanced.

To **invoke** (run/call) a method, we specify the name of the method followed by a set of parentheses. In the previous example, we use a special method called `println()`, accessible via a field on the System class, to print output to the console. If we called that method without any input, it would look like: `System.out.println();` This line of code would print a new line character to the console (an empty line) as we have not passed it any input inside its parenthesis.

9 A nod to JShell

JShell is a REPL tool, Read-Eval-Print-Loop, for experimenting with Java code to get immediate results.

To open the JShell tool, ensure you have Java on your system path and enter jshell into the console. You should see something like the following if JShell opened correctly:

```
$ jshell
| Welcome to JShell -- Version 17.0.1
| For an introduction type: /help intro

jshell>
```

The following commands will be useful for interacting with JShell:

Command	Description	Example
`/exit`	Exits the JShell tool.	`/exit`
`/list`	The following commands will be useful for interacting with JShell:	`/list`
`/save`	The following commands will be useful for interacting with JShell:	`/save snippet.txt`
`/open`	Opens and loads all code snippets from the specified file.	`/open snippet.txt`
`/reset`	Opens and loads all code snippets from the specified file.	`/reset`

Continued on next page

Continued from previous page

Command	Description	Example
`/vars`	Opens and loads all code snippets from the specified file.	`/vars`
`/help`	Opens and loads all code snippets from the specified file.	`/help`

9.1 Exploratory programming

In Java, we can perform the standard Arithmetic operations like addition, subtraction, multiplication, and division. If I enter `3 + 3;` into JShell for example, I will get the following result:

```
1 jshell> 3 + 3;
2 $1 ==> 6
```

What has happened here is that the Java code has been executed and a temporary variable has been created to store the result of the addition, a **variable** stores data for us and has an associated name. If I enter the name of the new variable, $1, into the console it will return $1 ==> 6 – this is informing us that the variable named $1 points to the value of 6. We can also pass the variable as input to a method, such as the `println()` method:

```
1 jshell> System.out.println($1);
2 6
```

Try entering the following operations into JShell and noting their results:

- `3 * 3`
- `3 / 3`
- `3 / 0`
- `3 - 3`
- `(3 + 3) * 3`
- `3 + 3 * 3`

10 The basics of Java

This section will cover the basic syntax of Java alongside topics like variables and data types.

10.1 Variables

A **variable** can be pictured as a storage bucket with a name that may contain a value. For example, in figure 10.1 we have a variable called x which refers to the value of 3.14. Variables may be **statically** or **dynamically typed**, that is they may only ever contain one type of data, or they may morph between data types respectively. For example, a static type could be an integer where our variable can only be a whole number and never a decimal. A dynamic variable could initially store an integer, and then store a float (decimal number) next, and then possibly a string of text – dynamic variables don't care about the type of data they store.

10.2 Data types

In programming, **data types** are a way of representing data and the operations that are applied to that data. The Java programming language is a **strongly typed** language, also known as *static typing* in which we must specify the data type when we create a variable. The typical syntax for declaring and initialising variables in Java are as follows:

```
1 // declare and initialise at the same time
2 DataType variableName = value;
3
4 // declare first
5 DataType2 variableName2;
6
7  // initialise later
8 variableName2 = value;
```

Listing 10.1: Declaring and initialising variables

> Comments in Java begin with two forward slashes, //, these are ignored by the compiler.

The above example demonstrates the expected grammar rules, known as the **syntax** for declaring and initialising variables. First, we must declare the type of data that we are storing in the variable. Next, the name is declared – usually in **camelCase** where the first letter is lowercase, and every new word afterwards is in uppercase. We finally use the **assignment operator**, =, to assign a value to the variable. The whole statement ends in a semi-colon to indicate to Java that this is a complete instruction in Java.

- Assigning a value to a variable is done from right to left (right-associative),

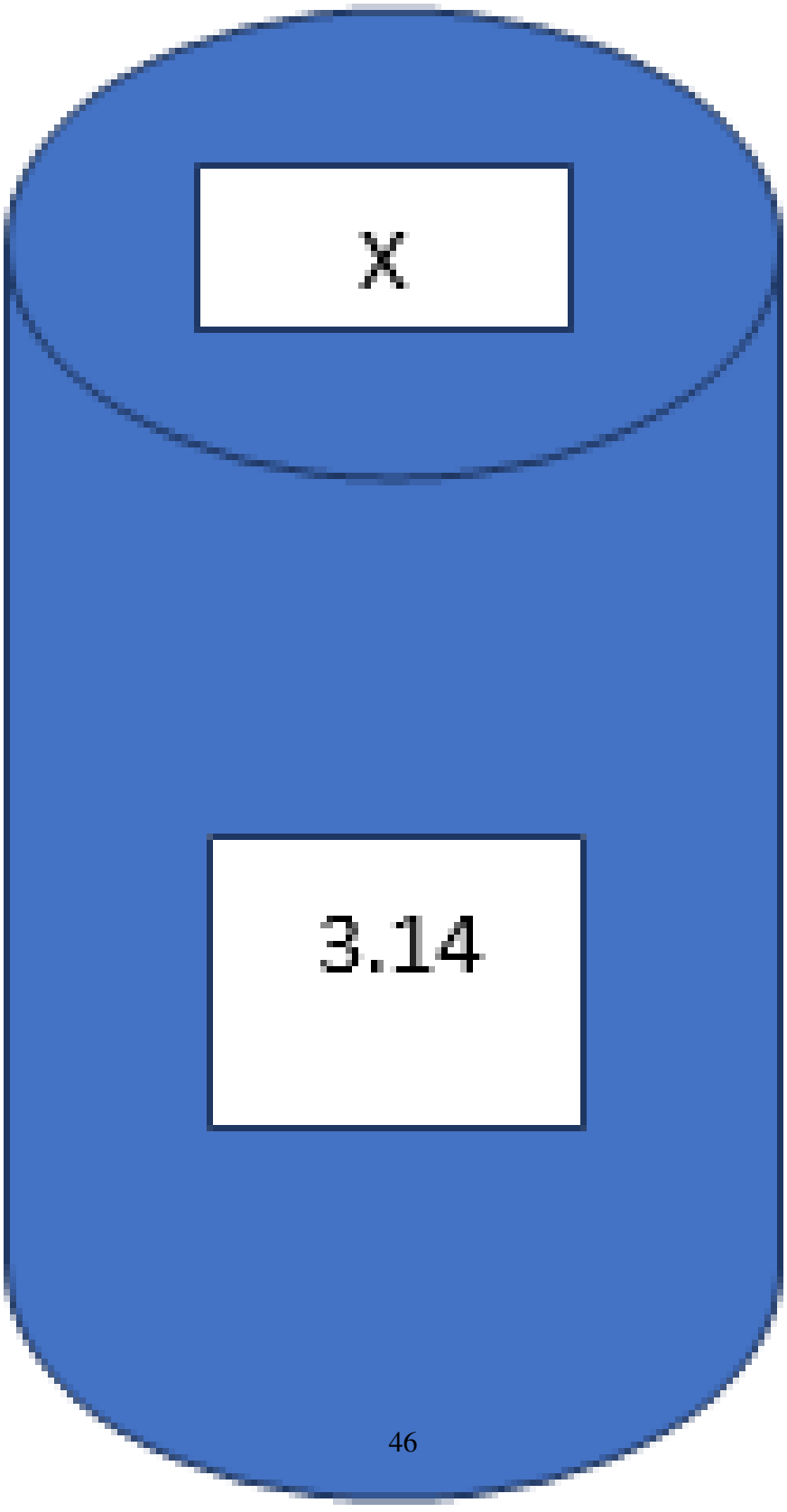


Figure 10.1: A variable represented visually

this means the value or expression on the right is evaluated by the computer before it is then assigned to the variable on the left.

There are two different categories of data type in Java:

- **Reference types**: Variables that store objects created from a class
- **Primitive types**: The smallest values in the language that can be used to create complex reference types

10.3 The primitive data types

Java has eight primitive data types which can be used anywhere in a Java application, a **primitive type** being the smallest value that can be represented in Java.

It is important to note that primitive types do not have methods, this means we cannot add additional behaviours beyond what has been designed in the language for the primitive types. Java's eight primitive types include numbers, characters, and logical values:

Data type	Representation	Range	Default value
`boolean`	Logical	`true` or `false`	`false`
`byte`	8-bit	`-128` to `127`	`0`
`short`	16-bit	`-32768` to `32767`	`0`
`int`	32-bit	`-2147483648` to `2147483647`	`0`
`long`	64-bit	`-9223372036854775808` to `9223372036854775807`	`0L`
`float`	32-bit	`3.4e +/- 38` (7 digits precision)	`0.0F`
`double`	64-bits	`1.7e +/- 308` (15 digits)	`0.0`
`char`	Unicode	`\u0000` to `\uFFFF`	`\u0000`

1. **Exploratory exercise**
   For this exercise, open the JShell tool and follow along with the code examples.
   Integers and floating-point numbers
   An **integer** is a whole number whereas a **floating-point** is a decimal number.
   As briefly described, we have a syntax for declaring variables of a specific data type in Java. To declare a variable of type `float`, we type the following (except comments):

```
// declare a variable of type float called pi and store 3.14 in it
float pi = 3.14;

We can then further declare more variables and use them in operations:
int multiplier = 2;

// create a variable called result and store the product of pi and multiplier in it float
product = pi * mulitiplier;
```

Running the above code in JShell (except lines beginning with //) will result in the following output:

```
jshell> float pi = 3.14f;
pi ==> 3.14

jshell> int multiplier = 2;
multiplier ==> 2

jshell> float product = pi * multiplier;
product ==> 6.28
```

Characters

In Java, we can represent individual characters in code, this can be done in a few different ways.

Create char using single-quotes and actual character:

```
jshell> char c1 = 'A';
c1 ==> 'A'
```

Create char using decimal ASCII code:

```
jshell> char c2 = 65;
c2 ==> 'A'
```

Create char using Unicode character:

```
jshell> char c3 = '\u0041';
c3 ==> 'A'
```

In these examples, we set three different char variables to the letter ‘A’.

Logical

Booleans are what are known as the **logic type**, they allow us to perform boolean logic to state conditions in our program. A boolean can be declared and initialised as follows:

```
jshell> boolean isRaining = true;
isRaining ==> true
```

An important thing to remember when declaring boolean variables is to turn the name into a question, boolean variables may only have one of two states like a yes or no question; shaping the names like a question helps us infer what the boolean is tracking. For example, look at the following snippet:

```
jshell> boolean raining = true;
raining ==> true
```

What does true represent here, that it is raining or that it is not... It is not a question we can answer, and then would likely have to make inferences based on how the variable has been used in the program – a complex and time-consuming process.

2. **Exercise**
   Using JShell:
   2.1 Create at least 1 variable of each data type
   2.2 Print each variable using System.out.println(varNameHere);
   2.3 The code `System.out.println('H');` outputs `H` to the console. Try the following code and note why you think the code outputted what it did: `System.out.println('H' + 'e');` (HINT: *Characters can be represented as alphanumeric characters, a Unicode sequence, or a number.*)

10.4 Reference types

A **reference type** is a data type declared as a class, these data types are different from primitive types in a few ways:

- Classes can have **fields** which hold data
- Classes can have **methods** defined on them that perform desired behaviours either independently of or on the data
- Reference types are stored in a place called the **heap**, a special area of memory in the JVM reserved for storing objects; primitive types are generally stored in a place called the **stack**. These will be explored further later.
- There can be a potentially infinite number of classes (bounded by hardware and memory limits) whereas there are only eight primitive types

We have previously seen that a primitive type called `char` exists in Java, that is great that we can represent characters but how do we represent a sequence of characters... There does in-fact exist a reference data type in Java known as a `String` which can be used to represent sequences of characters, this allows us to store whole words, sentences or even paragraphs of text in variables rather than just a single character.

When talking about strings of characters, we use the capitalised form to refer to the class – `String` – and the lowercase version when talking about strings in general. This text will highlight usages of the class version as a code snippet to help this stand out in the text.

To create a variable that stores a string, we just declare the data type like a primitive, then the name and then assign it a value:

```
jshell> String forename = "Alvin";
forename ==> "Alvin"
```

We have declared a variable called `forename` which references a string with the value of `"Alvin"`, this variable can now be used in further operations. For example, we may ask for Alvin's surname so we can put them together:

```
jshell> String surname = "Beck";
surname ==> "Beck";

String fullName = surname + "." + forename;
fullName ==> "Beck.Alvin";
```

In this example, we created a second variable called `surname` of type `String`. We then used the *additive operator* to perform string concatenation upon three different strings, to then store the result in the `fullName` variable.

String concatenation is an operation in which we join two strings together to form one new whole string. When you see a string directly in code it is known as a **string literal**. For example, the string `"."` In the above example is a string literal.

1. **Exercise**: Create and initialise a string variable
   Create a string called `name` that will hold your name
2. **Exercise**: Create and initialise another variable
   Create a string called `favouriteColour` that will hold your favourite colour
3. **Exercise**: Use string concatenation to create a greeting variable
   Create a string called `greeting` that when printed to the console would output a string like the following: `Hello Alvin Beck, hope you are well today! If I remember correctly, your favourite colour is red right?` (This will require string concatenation).

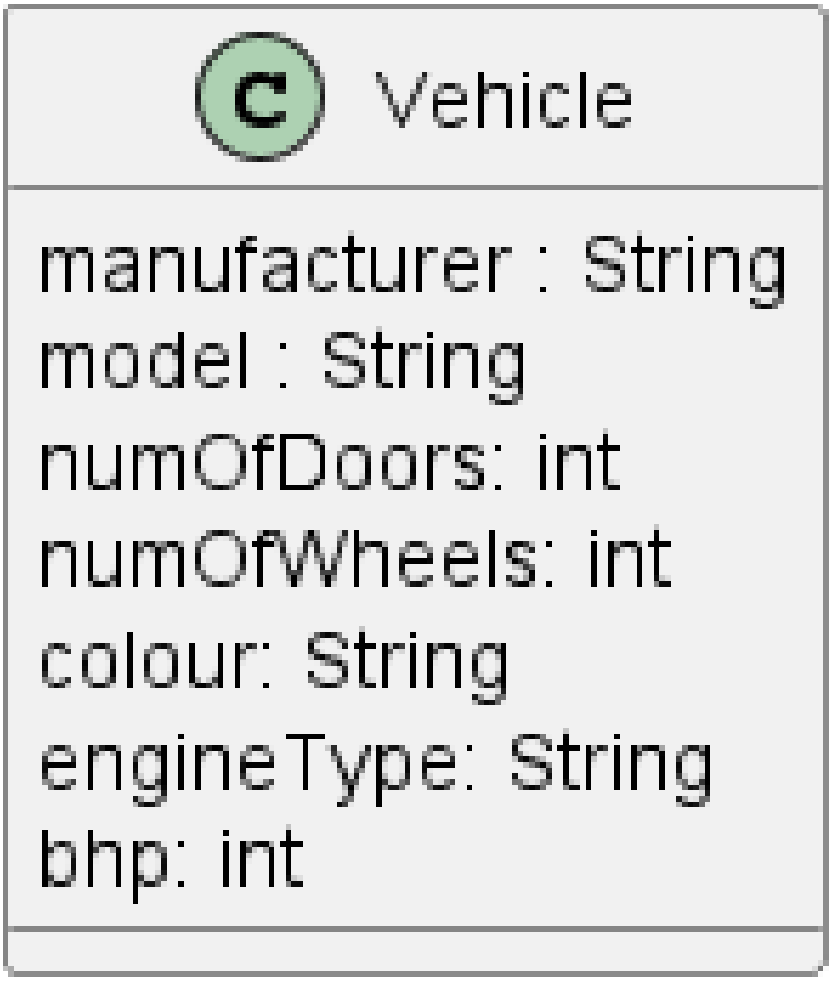
10.5 User-defined reference types

A **user-defined type** is a reference type that has been declared using type syntax, this could be a class, enumerated types or even interfaces – enumerated types and interfaces will be explored in a later chapter.

As we already know, a reference type is a class. Java provides some default classes, like `String`, to give us extra functionality in the applications we produce – generally, only the most common and useful data types are provided by programming languages. This means for extra functionality; we must define new reference types (classes) ourselves. Before we can write our own reference types, we must introduce the term **object**. Java is an object-oriented language, but what does this mean... If we think for a moment about something we would like to represent in software, maybe a car; what characteristics of cars exist that are common across all of them. When we create a class, we are creating a blueprint on how to construct new objects from that class – this is like real life where vehicle manufacturers implement the blueprint of a car to create new objects in the real world. When thinking about common traits of cars, what comes to mind for me is:

- The manufacturer and model
- Number of doors (sports cars usually only have two doors, family cars often have five, hatchbacks three, etc...)
- Number of wheels (think Reliant Robin, this could vary)
- Colour
- Engine type (electric, rotary, standard combustion, etc...)
- Horsepower (hp, common measurement that can be applied to measure engine power)

As a diagram represented using the Unified Modelling Language (UML), this might look like:



We start to see the blueprint forming from which we could fill in many different variations of data, giving us many unique **instances** (another term for objects) of the car class. For example, we could create two different instances of the `Car` class:

vehicle1
manufacturer = "Mazda" model = "787B" numOfDoors: 0 numOfWheels: 4 colour: "GREEN RED" engineType: "ROTARY" bhp: 650

vehicle2
manufacturer = "Reliant Motor Company" model = "Reliant Robin" numOfDoors: 3 numOfWheels: 3 colour: "LIGHT BLUE" engineType: "INLINE 4" bhp: 40

1. **Exercise**: Create a class in JShell
   As we have already discovered, a class can easily be declared using the `class` keyword in Java; follow along with the next few steps with JShell to learn how to create reference types yourself. You may want to type `/reset` in JShell if you have not closed and reopened it as we will be saving the code we enter to a file.
   In the JShell editor, enter the following code as it appears:

```
1 public class Car {
2     String manufacturer;
3     String model;
4     int numOfDoors;
5     int numOfWheels;
6 }
```

   The Car class above has four *fields*, these are known as **instance variables** as the data inside them will differ depending on the object created from Car that stores them. When typing this in JShell, you will see arrows, `...>`, which indicate you are currently writing code nested inside something, i.e., the class `Car` in this case:

```
1 jshell> public class Car {
2 ...> String manufacturer;
3 ...> String model;
4 ...> int numOfDoors;
5 ...> int numOfWheels;
6 ...> }
7 | created class Car
```

   You will also receive a message indicating the successful creation of the Car class. At this point, I would recommend running `/save Car.txt` to save your class to a file.
   The question now becomes, how do we create unique objects from the blueprint we have created? If we try creating a `Car` object like we would a primitive variable, we will encounter an error:

```
jshell> Car myCar = Car;
| Error:
| cannot find symbol
| symbol: variable Car
| Car myCar = Car;
|
```

Java thinks that we are trying to find a variable of some kind called `Car` to copy its value into the new `myCar` variable. This is an example of a **syntactical error** that is presented to us at compile time, i.e., when we try to convert the source code into byte code using `javac`. To create an instance of the `Car` class, we must use the `new` keyword to indicate we want to create an instance of a class:

```
jshell> Car myCar = new Car;
| Error:
| '(' or '[' expected
| Car myCar = new Car;
|
```

After running this in JShell, we get another error stating that we are missing either a parenthesis or a square bracket. It turns out, there is a very specific format to how we must create new objects of a certain type:

```
DataType variableName = new DataType();
```

First, we declare the data type of the variable and next its name. We then assign the value on the right to the variable on the left, the value is an instance of the specified `DataType` which is returned by the expression `new DataType()`. We must include the `new` keyword before the data type we want to create, we must also put a set of parentheses after the data type:

```
jshell> Car myCar = new Car();
myCar ==> Car@443b7951
```

The result of this assignment shows that `myCar` holds the value `Car@443b7951` (a reference) rather than our data, just know for now that this is the value stored in the variable and that it is a way of uniquely identifying different objects – objects are stored in the heap which will be explored later.
It is important to note though, we cannot store a value of a different type in the `Car` variable. As mentioned previously, Java is statically typed:

```
jshell> Car myCar = "hi";
| Error:
| incompatible types: java.lang.String cannot be converted
    to Car
| Car myCar = "hi";
|             ^--^
```

The JVM will throw an error at compile-time if we tried to compile a program with a line of code like this, we must ensure that the values we place into a variable are of the same type. It is also important to note that reference types declared as a field on a class (instance variables) take on a default value of `null` if you do not initialise them with a value when creating an object.
Once we have an instance of `Car`, we can use **dot notation** to access the instance variables to retrieve or set the data:

```
jshell> myCar.manufacturer = "Mazda";
$4 ==> "Mazda"

jshell> myCar.model = "787B";
$5 ==> "787B"
```

To use dot notation, we specify the name of an object followed by a period/-dot and then the *instance member* that we want to access – **instance members** are either instance fields/variables or instance methods which will be explored further later. When accessing a field, we can assign a value using the standard assignment operator as seen above. To retrieve a value from a field and store it in a variable, we declare a variable of the correct type and then assign it the value of the field:

```
jshell> String manufacturer = myCar.manufacturer;
manufacturer ==> "Mazda"
```

2. **Exercise**: Creating instances of a class
   For this exercise, ensure you followed along with the first exercise. You tasks are as follows:
   - Create a new instance of `Car` called `myCar`
   - Initialise the `numOfDoors` and `numOfWheels` instances variables on the `myCar` variable
   - Create a second instance of the class `Car` and initialise all its fields
   - For both instances of `Car`, print a string that looks like: `The Mazda 787B is in disrepair, it should have 4 wheels but doesn't have one!`. You will want to use string concatenation to join together multiple pieces of data into one string, you will also need to use dot notation to access the variables inside the `Car` objects.

10.6 Methods

In mathematics, we have the concept of a function. A **function** is some calculation which takes an input and produces an output. Commonly, we see them in linear

algebra in the form: $y = x+1$. The `x` and `y` points can be plotted on a graph, which takes us into the realm of the *cartesian plane* - a two-dimensional coordinate plane formed by the intersection of the x and y axis. Anyway, mathematicians commonly represent function using the notation $f(x) = x + 1$ where $f(x)$ just means y. We can read $f(x)$ as the *function of x* or the *function applied to x*.

You might be wondering, how does this connect to programming... Well, functions are available in almost all available high-level programming languages for executing a block of code without having to type out all of the contained code multiple times. Let's use JavaScript as an example, we can define a function like so:

```
function foo(x) {
    console.log(x * 2);
}

foo(1); // 1
foo(2); // 2
foo(3); // 3
```

Instead of writing out `console.log(x)` multiple times, we just make a call to this thing called `foo()`. We pass it an input of `x` and it returns us an output of `y`. Java has its own concept of a function, the humble method. A **method** is just a function which belongs to a class or instance of a class, dependent on whether it is static or not. The method can work on the data inside the class.

Let's first familiarise ourselves with the layout of a method:

```
AccessModifier ReturnType methodName(ParameterType paramName,
    ParameterType paramName2, ...) {
    // body
}
```

- `AccessModifier`: First, there is an optional access modifier which states where the method can be accessed. `public` means everywhere for example.
- `ReturnType`: Second is the return type, what type of data will the method return when called.
- `methodName`: The method name is its identifier, it is what we use to call it on an object
- `ParameterType paramName`: Inside the parenthesis can be upto 2^{16} parameters, each consisting of a data type and name. These represent the inputs to the method.

Everything before the curly braces, `{}`, represents the **method header**. The

`methodName` and its parameters make up the **method signature**, a way for the Java compiler and JVM to identify methods and allows for *method overriding*. The **method body** is everything inside the curly braces, this is the code that the method will execute when called. One other important thing to know is that parameters are local variables, local to the method. A method creates a new **scope** when called, a lexical area of code in which variables can be accessed.

Now let's look at an example:

```
class User {
    // field
    private String username;

    // method
    public String getUsername() {
        return this.username;
    }

    // method
    public void setUsername(String username) {
        this.username = name;
    }
}
```

We have declared a `User` class with one field, a `username` of type `String`. We have declared two methods, `getUsername()` and `setUsername()`. We will first look at their usage before explaining them:

```
User user = new User();
user.setUsername("Bob");

String username = user.getUsername();
System.out.println(username); // Bob
```

We first create a new `User` object, we then use **dot notation** to access the method called `setUsername` which is defined on that object. We pass it the string `"Bob"` as input. After that, we then use dot notation again to call (also known as invoke) the method `getUsername()` on the object to retrieve the value stored in the object and put it in the `username` variable. You may have noticed that there is also a `this` keyword, it is used to refer to the object on which you have called the method. In `setUsername`, we have two `username` variables in scope - the one defined as a field on the class, and the local variable in the methods parameter list. To differentiate them, we use the `this` keyword followed by a dot and then the name of the field to access it. The power comes from being able to have many unique `User` objects:

```
User user = new User();
user.setUsername("Bob");
User user2 = new User();
user.setUsername("Sarah");
```

10.7 Constructors

A **constructor** is a special feature of an object responsible for its initialisation. When we use the `new` keyword to create a new object, we are actually calling a constructor:

```
Object o = new Object();
```

When we create a custom class, a default constructor will be inserted by the compiler which looks like:

```
class Foo {

    // constructor starts here
    public Foo() {
        super();
    }
    // constructor ends here
}
```

The constructor looks very much like a method, it just does not have a return type; it must also have the same name as its class. Constructors have the following structure:

```
AccessModifier ClassName(ParameterType paramName, ...) {

}
```

Let's look at a class with a variety of constructors:

```
class User {
    private String username;

    public User() {
        this.username = "Unknown";
    }

    public User(String username) {
        this.username = username;
```

```
    }

    public String getUsername() { return username; }
}
```

Here, we have two constructors. A no-argument constructor which will initialise `username` to a default value and a constructor which takes a `username` as input. Usage would look like:

```
User user1 = new User();
User user2 = new User("Bob");

user1.getUsername(); // Unknown
user2.getUsername(); // Bob
```

Another aspect of constructors is that we can chain calls between them using the `this` keyword as if it was a method itself, the only rule is that the `this()` call must be the first statement in the constructor:

```
class User {
    private String username;

    public User() {
        this("Unknown");
    }

    public User(String username) {
        this.username = username;
    }

    public String getUsername() { return username; }
}
```

This time, in the no-argument constructor `User()`, we call the one-argument constructor `User(String username)` using `this("Unknown")` to reuse the code defined in that constructor and make initialising our objects in a variety of ways easier.

10.8 Identifiers

So far, we have learned to create variables of both reference and primitive types – a very important part of creating variables is naming them appropriately. Java does have some rules surrounding variable declaration:

- Identifiers cannot be reserved keywords such as `class` or `char`.
- Identifiers are case sensitive, `mycar` and `myCar` are different identifiers for example.
- Identifiers cannot be literal values like `""`, `true`, `false`, `0`, etc...
- Identifiers can contain alphanumeric characters `[0-9,A-Z,a-z]`, etc... but may not start with a number
- Identifiers may start with a letter, dollar symbol (`$`) or underscore (`_`)
- Identifiers may not contain whitespace
- Identifiers may not contain special symbols like `#`, `@`, `~`, etc...

10.9 Java core libraries

The Java JDK includes many different **application programming interfaces** (APIs) in the form of software libraries which we can use to develop software applications. A selection of which is visible in figure 10.2.

Some of the most important core libraries are shown in figure 10.2, `java.lang` is imported by default - implicitly, meaning we don't have to do it manually/explicitly do it - and is what allows us to use the `String` data type without any extra work.

If we wanted to use a random number generator, we would need to import it from the `java.util` package using its fully qualified name, a **fully qualified name** being made up of simple names separated by periods. A **simple name** is just underlying name of what is being accessed, such as the `math` package or the `Random` class. The default import for `java.lang` uses a **wildcard character** to import all classes in the package:

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

Import classes using the `import` keyword, when writing imports in `.java` files (during standard development rather than when using JShell) they must appear above any class names but below any package names. A package is just the name for a folder on your computer which logically groups together certain data types. `lang`, `util`, `math`, `io`, etc... are all simple package names. For example, to import the `Random` class:

```
1 import java.util.Random;
```

Once we have imported the `Random` class, we can create an instance of it to use just like any other class we have encountered:

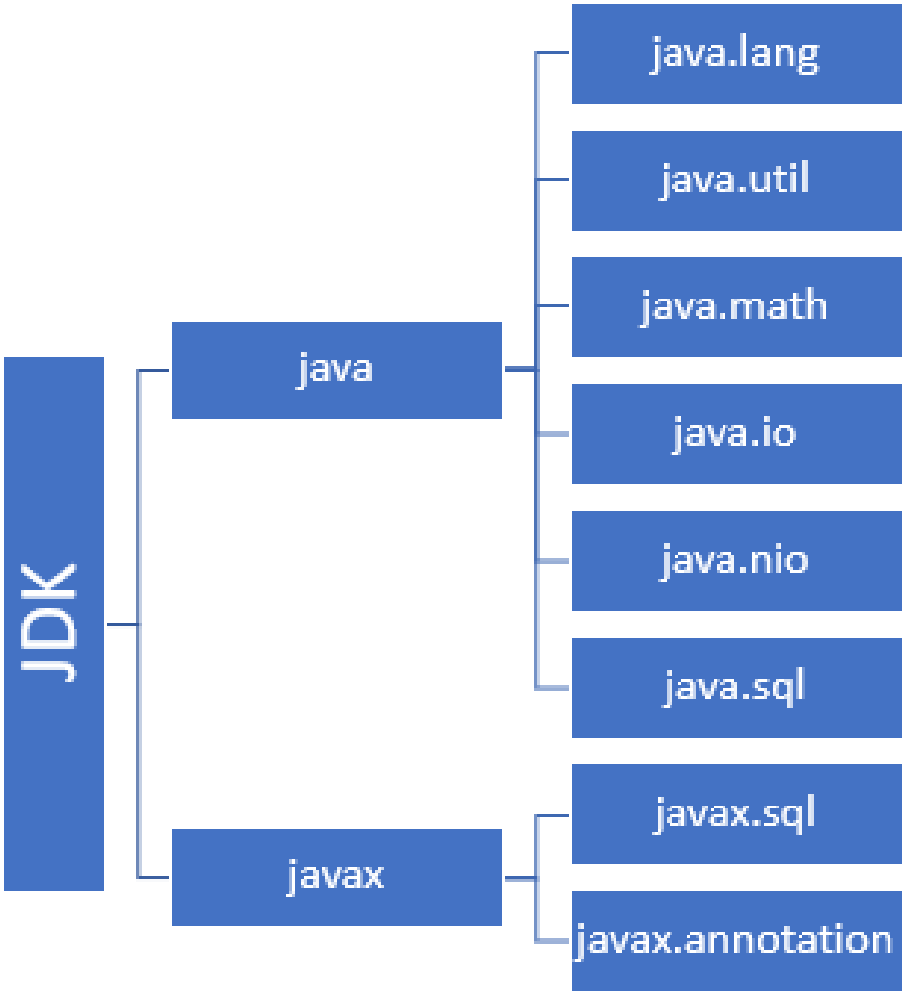


Figure 10.2: A selection of packages inside the JDK

```
jshell> Random randomGenerator = new Random();
randomGenerator ==> java.util.Random@4d76f3f8
```

If we look into the API documentation for the `Random` class, we can learn how to use it: Random (Java Platform SE 8)(oracle.com).

Try the following code if you would like to explore the `Random` class further:

```
int randomNumber = randomGenerator.nextInt(10);
int randomNumberTwo = randomGenerator.nextInt(10);
int sum = randomNumber + randomNumberTwo;

double smallNumber = randomGenerator.nextDouble();
System.out.println("Random number one: " + randomNumber);
System.out.println("Random number two: " + randomNumberTwo);
System.out.println(randomNumber + " + " randomNumberTwo + " = "
    + sum);
```

We have now discovered that Java includes **built-in packages** which contain classes with other behaviours that we can try. We will explore the Java core libraries further and how we define our own packages later on.

10.10 Operators

The Java language contain **operators** which can be applied to variables, values, or literals. The values used with an operator are known as its **operands**. For example, in 3 – 4, 3 is the left operand, - is the operator and 4 is the right operand.

Java has three types of operator:

- **Unary**: Unary operators have one operand
- **Binary**: Binary operators have two operands
- **Ternary**: Ternary operators have three operands

Operators, like in Maths, have a **precedence** – that is, we perform multiplication before addition for example. Java follows the **BIDMAS/BODMAS/PEMDAS** rules for arithmetic operators but does also include additional operators for additional functionality.

Associativity is the way in which an expression is evaluated, an **expression** is made up of variables, operators and method invocations and returns a single value,

whether that value is nothing or something. In the following examples, the parts highlighted in bold are expressions:

int **age = 30**;

String **name = "Bob"**;

System.out.println(**name + " is " + age + " years old."**);

if (**age == 30**) {

System.out.println(**"You are 30"**);

}

System.out.println(**name.toUpperCase()**);

int **mathExpression = 3 + 3**;

What we can see is that assignments are expressions, the value `30` is assigned to the age variable for example. **Arithmetic operations** are also expressions, the result of the right is assigned to the variable on the left.

Onto **associativity** though, all it means is that an operation is carried out in a certain direction: left-to-right or vice-versa. If two operators have the same precedence, they are carried out according to their associativity. The following table lists the operators from highest to lowest precedence with their associated associativity's:

Operator	Symbols and examples	Associativity
Miscellaneous	`[]`, `()`, `.`	LEFT → RIGHT
Post-unary operators	`expression++`, `expression--`	RIGHT → LEFT
Pre-unary operators	`++expression`, `--expression`	RIGHT → LEFT
Other unary operators	`-`, `!`, ~, `+`, `(type)`, `new`	RIGHT → LEFT
Multiplication, division and modulus	`*`, `/`, `%`	LEFT → RIGHT
Addition and subtraction	`+`, `-`	LEFT → RIGHT
Shift operators	«, », »>	LEFT → RIGHT
Relational operators	<, >, <=, >=, `instanceof`	LEFT → RIGHT

Continued on next page

Continued from previous page

Operator	Symbols and examples	Associativity
Equality operators	==, !=	LEFT → RIGHT
Bitwise logical operators	&, ^, | (AND before XOR, XOR before OR)	LEFT → RIGHT
Short-circuit logical operators	&&, || (AND before OR)	LEFT → RIGHT
Ternary operators	`boolean expression ? expression1 : expression2`	RIGHT → LEFT
Assignment operators	=, +=, -=, *=, /=, %=, &=, ^=, |=, «=, »=, »>=	RIGHT → LEFT

10.11 Unary operators

Unary operators can only be applied to a single value, the following table lists the unary operators:

Operator	Description	Example
`!`	The logical complement operator flips the value of a boolean expression.	`boolean b1 = !true;`
`+`	The additive operator indicates that a number is positive.	`int num = +3;`
`-`	The negation operator indicates a number is negative or negates an expression.	`int num = -3;`
`++`	The increment operator increments a value by 1.	`i++; ++i;`
`--`	The decrement operator decrements a value by 1.	`i--; --i;`
`(type)`	The cast operator casts a value to the specified type.	`int x = (int) 32.45;`

You may have noticed that the increment and decrement operators can be prefixed or suffixed to a variable:

- **Postfix-increment** and **postfix-decrement** will first return the current value

of its operand, and then it will increment or decrement its value by 1.
- **Prefix-increment** and **prefix-decrement** will first increment or decrement the current value of the its operand by 1, and then return modified value.

The `(type)` operator allows us to convert variables of a certain type to a variable of a related type, such as converting a double to an int in the above table, this will be explored further in numeric promotion.

10.12 Arithmetic operators

Arithmetic operators are binary operators, this means an operator is applied to two operands. The following table lists operators:

Operator	Description	Example
`+`	The additive addition operator returns the sum of two numeric values.	`3 + 3`
`-`	The subtraction operator returns the difference of two numeric values.	`3 - 3`
`*`	The multiplication operator returns the product of two numeric values.	`3 * 3`
`/`	The division operator returns the quotient of the dividend and the divisor (numerator and denominator in fractions)	`3 / 3`
`%`	The modulus operator returns the remainder after dividing the dividend by the divisor. The returned value is always a whole number, this means it will truncate any values after the decimal point.	`9 % 2`

The default order of operations for arithmetic operators follows that of BIDMAS/-BODMAS:

- Brackets
- Indices/Ordinals
- Division, Multiplication
- Addition, Subtraction

This means we can give certain parts of our expression's higher precedence:

```
int num1 = (3 + 3) * 10 - 50; // -17
int num2 = 3 + 3 * 10 - 50; // 10
```

Adding a set of parentheses around the addition changed the way in which the expression was evaluated, which changed the result.

10.13 Assignment operators

Assignment operators are used to assign the result of an expression to a variable. The simplest is the standard assignment operator represented by the equal's sign:

```
jshell> int bigNum = 3234254 + 1;
bigNum ==> 3234254
```

Assignment operators have a right-to-left associativity, this means that the right operand is evaluated before the left. If I was reading and executing the above assignment like the computer, it would go as follows:

1. Statement encountered: int bigNum = 3234254 + 1;
2. Evaluate right operand: 3234254 + 1;
3. Expression evaluated: 3234255,
4. Declare variable of type int called bigNum
5. Assign evaluated right operand to declared variable bigNum

An assignment is an expression, and thus also can return a result to be used in another assignment:

```
jshell> int a = 3;
a ==> 3

jshell> int b = (a *= 3);
b ==> 9
```

Compound assignment operators extend the ability of the standard assignment operator and give a short-hand way of writing out simple expressions, the most common compound assignment operators being:

Operator	Description	Example

Continued on next page

Continued from previous page

Operator	Description	Example
+=	The additive compound assignment operator returns the sum of two numeric values.	`int i = 3; i += 10;`
-=	The subtraction compound assignment operator returns the difference of two numeric values.	`int i = 3; i -= 10;`
*=	The multiplication compound assignment operator returns the product of two numeric values.	`int i = 3; i *= 10;`
/=	The division compound assignment operator returns the quotient of the dividend and the divisor (numerator and denominator in fractions).	`int i = 3; i /= 3;`
%=	The modulus compound assignment operator returns the remainder after dividing the dividend by the divisor. The returned value is a whole number for integers, this means it will **truncate** any values after the decimal point.	`int i = 3; i %= 3;`

To understand compound assignment operators, you just need to understand how they expand out to a full expression. For example, `num1 += num2;` is the same as writing `num1 = num1 + num2;` and `num1 -= num2;` is the same as writing `num1 = num1 - num2;`.

10.14 Comparison operators (Equality, relational, boolean logic)

Java contains many different comparison operators which are used in the creation of Boolean logic expressions, that is an expression using the logic rules originally set out by George Boole in the 19th century. The equality and relational operators are used to produce true or false values from the comparison of numbers whereas boolean logic operators introduce the core Boolean logic.

There are two **equality operators** in Java, one for comparing if two values or objects are equal and another for comparing if they are not equal:

Operator	Applied to primitives	Applied to objects
==	The equality operator returns true if the left operand is equal to the right operand, otherwise false.	Returns true if the left and right operands refer to the same object reference, otherwise false.
!=	The inequality operator returns true if the left operand is equal to the right operand, otherwise false.	Returns true if the left and right operands do not refer to the same object reference, otherwise false.

Examples:

```
int num1 = 30;
int num2 = 60;
boolean isEqual = (num1 == num2); // false
isNotEqual = (num1 != num2); // true
boolean isNotEqualAlt = !(num1 == num2); // true
```

Relational operators also compare two numeric operands, aside from the instanceof operator:

Operator	Description	Example
<	The less than operator returns true if the left operand is smaller than the right operand, otherwise it returns false.	`4 < 5`
<=	The less than or equal to operator returns true if the left operand is smaller than or equal to the right operand, otherwise it returns false.	`4 <= 5`
>	The greater than operator returns true if the left operand is larger than the right operand, otherwise it returns false.	`4 > 5`
>=	The greater than or equal to operator returns true if the left operand is larger than or equal to the right operand, otherwise it returns false.	`4 >= 5`

Continued on next page

Continued from previous page

Operator	Description	Example
`a instanceof b`	The instanceof operator returns true if the left operand is an instance of a class or subclass, or interface as specified in the right operand.	`"Hello" instanceof String`

Boolean logic operators are applied to the boolean data type, these are useful for creating conditions in our software and creating logic. There are three boolean operators in Java, but these three operators may be applied in a variety of ways – even to numbers in when using bitwise operators. The three common operations are AND, OR and XOR.

Logical operators may be applied to both numeric and boolean data types, they are known as a bitwise operator when applied to numbers and perform different logic then next described. The logical operators are as follows:

Operator	Description	Example
`&`	The logical AND operator returns true if both the left and right operands are true, otherwise it returns false.	`true & true`
`^`	The logical XOR operator returns true if only one of the left and right operands is true, otherwise it returns false.	`true ^ false`
|	The logical OR operator returns true if either the left or right operands, or both, are true, otherwise it returns false.	true | false

In a logical expression, both operands will always be checked unlike short-circuit operators… **Short-circuit** operators are a special kind of boolean operator that can only be applied to the boolean data type, the following table describes the short-circuit operators:

Operator	Description	Example

Continued on next page

Continued from previous page

Operator	Description	Example
&&	The short-circuit AND operator returns true if both the left and right operands are true, otherwise it returns false. The right operand is only checked if the left is true.	`true && false`
||	The logical OR operator returns true if either the left or right operands, or both, are true, otherwise it returns false. The right is only checked if the left is false.	false || true

10.15 Ternary operator

The **ternary operator** is a special operator that is used for short conditional statements that return one of two specified values, the ternary operator has three operands and takes the form:

```
boolean result = (boolean_expression) ? result_if_true :
    result_if_false;
```

A special rule about ternary expressions is that they always return a value, hence why there is an assignment occurring to the boolean result. The `(boolean_expression)` is exactly that, a set of values combined with the boolean logic operators, comparison operators, relational operators, or a mix. A boolean expression that checks if the temperature is above 30 is demonstrated below:

```
jshell> int temp = 28;
temp ==> 28

jshell> boolean isRaining = (temp > 30); // false
isRaining ==> false
```

If we wanted to get a string that states, `"It is too hot"` when the temperature is above 30 or `"It is just right for me"` otherwise, we can plug the expression `(temp > 30)` into the first operand of the ternary expression. This gives us:

```
String statement = (temp > 30) ? result_if_true :
    result_if_false;
```

We now have a semi-complete statement; the next two operands are what values should be returned if the boolean expression is true or false. All we must do is put in the string literals containing the values we want in this case:

```
jshell> int temp = 28;
temp ==> 28

jshell> String statement = temp > 30 ? "It is too hot" : "It is
    just right for me";
s ==> "It is just right for me"
```

We could then print the `statement` string to the console or use it somewhere else.

10.16 Operator exercises

1. Use 1 of each of the arithmetic operators and assign the result of the expression you make to a variable.
2. Use 1 of each of the arithmetic operators and assign the result of the expression you make to a variable.
3. Boolean expressions can be compounded, that is they can have multiple conditions in them:

```
boolean isRaining = true;
boolean isSunny = false;
boolean isCloudy = true;
boolean isRainingAndSunnyOrCloudy = isRaining && isSunny ||
    isCloudy;
```

   In the above expression, the `isRaining && isSunny` expression is executed first, which results in `false`. The expression then becomes `false || isCloudy`. This then returns true, which is the correct result, but the logic of the expression is wrong. If we change `isRaining` to false, we will still get `true` back. We want to know if it is both raining and whether it is sunny or cloudy. If it is not raining, then the expression should evaluate to false. Insert parenthesis or re-arrange the expression to make it evaluate as expected.
4. Given the integer `age` and double `hourlyRate`, create a boolean expression that checks if the `age` is greater than 18 but less than 21 and their hourly rate is greater than or equal to £6.83, i.e., that they are earning at least minimum wage for their age range:

```
int age = 18;
double hourlyRate = 7.00;
double minWageEighteenToTwenty = 6.83;
```

5. Boolean expressions can be created to check if a number is even or odd by using the modulus operator, for example:

```
jshell> boolean isOdd = (31 % 2) == 1;
isOdd ==> true
```

   The above expression can be used to check if a number is odd, replace 31 with other values to check if they are odd and then create a boolean expression to check if a number is even.

10.17 Numeric promotion

When performing arithmetic operations upon numeric types, we may mix types like an integer and a double in the same expression:

```
double num = 3.0 + 3;
```

Each primitive data type has a size – numerical types included. This means technically that an 8-bit number (`byte`) is also a 16-bit number (`short`), but a 16-bit number is not an 8-bit number as it is too large. For example, if we declare and initialise a variable of type `byte`, we can assign it to a variable of a primitive data type that holds a larger numeric value:

```
byte smallNum = 64;
short convertedNum = smallNum; // 64
```

   This can occur because a byte (8-bits) can fit in a short's (16-bit) memory space on your hardware.

If we tried to assign a `short` to a `byte`, or an `int` to a short we would get an error as a `short` is too large to fit in a `byte` and an `integer` is too large to fit into a `short`:

```
jshell> int bigNum = 3234254;
bigNum ==> 3234254

jshell> byte uhOh = bigNum;
|  Error:
|  incompatible types: possible lossy conversion from int to byte
|  byte uhOh = bigNum;
|              ^-----^
```

JShell will give us a warning about **lossy conversion**, it is indicating that an integer value is too large to fit in a byte and that doing so would cause a loss of accuracy

(numbers/data). To get around this, we can use typecasting to widen or narrow the type:

- **Widening** the type means making a type larger, i.e., assigning a byte to an int variable
- **Narrowing** the type means making a type smaller, i.e., assigning an int to a byte variable

If we wanted to put the large integer `bigNum` variable into a byte variable, we can by explicitly casting `bigNum` to a `byte`, but we will lose data if the value is too large for the desired narrower type:

```
jshell> byte lostMaData = (byte) bigNum;
lostMaData ==> -50
To fit into a byte, we need a number smaller than 256:
jshell> byte didNotLoseData = (byte) smallNum;
didNotLoseData ==> 32
```

The rules of numeric promotion are as follows:

1. If the left and right operand are of different types, the smallest type will be promoted to the largest. For example, `78938 + 92374893L` contains an integer and a long, the integer `78938` will be promoted to a `long` of the form `78938L`.
2. If the left or right operand is an integral type and the other a floating-point type, the integral type would be promoted to a floating-point type. For example, `3 + 4.0` contains an integer and a double, the integer `3` will be promoted to a `double` of the form `3.0`.
3. The small data types of `byte`, `short`, and `char` are first promoted to an `int` when used with any arithmetic operator, even if neither operator is an `int`.
4. After all promotions have been made, the result will be of the same type as the expressions operands.

10.18 Array theory

Arrays are a construct in programming that allow us to store statically sized collections of values in variables rather than just a single value.

The **array** data type is a special reference data type designed to store values *contiguously* in memory, meaning one after the other. If we represent the memory of a computer as having an address for its location (memory address) and a element stored in that address, we can create a simple tabular visualisation of how an array

would look at a hardware level:

```
| Memory address | @255 | @256 | @257 | @258 | @259 | @260 |
| Element        | 'H'  | 'E'  | 'L'  | 'L'  | 'O'  | '!'  |
```

Each memory address in the above model (`@255`, `@263`, `@271`, etc...) represents a byte, 8-bits. Each byte is storing a character to build a string of text - this is how the `String` data type is actually represented, by creating an array of characters.

> Technically, a `char` in Java is 16-bit (two bytes of data).

10.19 Declaring and initialising arrays

To declare an array, we must specify the data type that will be stored in the array (primitive or reference types are both allowed to be specified) and a set of square brackets after the data type:

```
1 DataType[] variableName;
```

All of the following declarations of an array are valid:

```
1 int[] numbers;
2 String []names;
3 boolean responses[];
```

To initialise an array, a size must be specified as arrays are **static** in size (this relates to how they store values contiguously in memory). Array initialisation takes the following form:

```
1 DataType[] variableName = new DataType[size];
```

All of the following are valid declarations and initialisations of an array:

```
1 int[] numbers = new int[10]; // holds 10 integer elements
2 String[] names = new String[5]; // holds 5 string elements
3 boolean[] responses = new boolean[3000]; // holds 3000 boolean
    elements
```

As arrays directly store values, whether primitive or reference, their **capacity/length** (the amount they can store) and its **size** (the amount of elements it contains) are always the same - this differs from how other data types that store collections of values are represented in Java, such as with the `ArrayList` and `LinkedList` data types which can dynamically grow and shrink in size.

To get the length of an array, and thus the number of elements it contains, use the array types length attribute:

```
int[] numbers = new int[10];
String[] names = new String[5];

int numbersLength = numbers.length; // 10
int namesLength = names.length; // 5
```

10.20 Initialising arrays at declaration

The initialisation of an array at declaration can be done in two ways, the first way (demonstrated in the previous section) will intialise the values in array to their data types default value (0 for primitive int, null for reference types such as `Integer` or `String` for example). This method requires us to specify a size.

We can also directly initialise an array with its intended values using an **array initialiser**:

```
// long form, size is inferred from input data, size = 5
int[] numbers = new int[] { 1,2,3,4,5 };

// short form, size = 2
String[] names = { "Fred", "Sarah" };
```

10.21 Initialising arrays after declaration

When initialising an array after declaration, it can only be initialised with the long form - whether we specify an initialiser or not:

```
int[] numbers;
String[] names;

numbers = new int[] { 1,2,3,4,5 }; // good

names = { "Fred", "Sarah" }; // Illegal start of expression
```

10.22 Accessing and setting data in an array

When data is stored in an array, each piece of data - known as an **element** - is given a position known as an **index**. Unlike the normal counting system for hu-

mans, computers start counting from 0; this is known as a zero-based index. If we represent an array of integers as a table, this then becomes quite simple:

```
int[] numbers = new int[] { 1,2,3,4,5 };
```

Index	0	1	2	3	4
Element	1	2	3	4	5

The first element in the array, 1, has the index of 0 while the index of the last element of the array is always the arrays length minus one. To access an element in an array, whether for retrieving or setting a value, we use the following syntax:

```
arrayName[index]
```

The following example demonstrates setting and retrieving values in an array:

```
int[] numbers = new int[10];

System.out.println(numbers[0]); // 0

numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;

System.out.println(numbers[0]); // 10

numbers[numbers.length - 1] = 100;
System.out.println(numbers[9]); // 100
```

After running this code, we end up with an array whose internal structure looks like:

Index	0	1	2	3	4	5	6	7	8	9
Element	10	20	30	0	0	0	0	0	0	100

> If the index specified is larger than an arrays `length - 1` or smaller than `0`, an error known as an `IndexOutOfBoundsException` will occur which tells us that the index specified is too large or too small for the specified array.

10.23 Iterating over an array

Arrays can easily be iterated over with a `for` statement by creating a counter variable which starts from 0, the same as the first index in an array, and is incremented while its value is less than the length of the array.

The following example iterates over an array of 5 elements and prints each one out to the console on a separate line:

```
1 int[] numbers = { 1,2,3,4,5 };
2
3 for (int i = 0; i < numbers.length; i++) {
4     System.out.println(numbers[i]);
5 }
```

The key part here is that we initialised the counter as zero, `int i = 0`, and set the expression to allow iteration to occur until we reach the final index of the array, `i < numbers.length`.

11 Control flow

11.1 Conditional statements

Conditional statements are used to provide different paths for our application to follow, they act just like the ternary operator except that we can usually provide more than just two different paths to follow. Conditional statements are used for selection, they allow the selection of some block of code to run based on some condition being met.

- **Decision problems** are those that have only two possible decisions, and one of them must be selected
- **Classification problems** are those that have more than two possible decisions, of which one must be selected

11.2 If statements

An `if` statement is a simple logic statement which accepts a boolean expression, if that boolean expression evaluates to `true` then the `if` statement will execute some region of code:

```
1 if (boolean_expression) {
2     // execute this code
3 }
4 // execution resumes here after the if statement
```

11.3 If-else statements

The **if-else** construct is a simple statement that accepts a boolean expression like a ternary statement, but instead of returning a value it will run a block of code based on the result of that expression:

- A block is a section of code, contained within curly braces {} that defines a scope, a scope being some area where variables are valid

– this will be explored further in the class member's chapter.

To create an if-else statement, we use the following syntax structure:

```
1 if (booleanExpressionIsTrue) {
2     // then do this
3 } else {
4     // do this instead
5 }
```

Let's say we want to set a string with a value that depends on the temperature of a person, we are checking for a fever. We could use a simple if-else statement to check if they are too hot:

```
int temp = 36.5; // degrees c
int feverTemp = 38; // degrees c
int minSafeTemp = 36.3; // degrees c
String output = "";

if (temp > feverTemp) {
    output = "Patient too hot! Start emergency cooling
        procedures";
} else {
    output = "Patient is fine";
}
```

11.4 Else-if statements

The simple if-else statement can also be expanded to handle multiple different conditions, leading to different branches that the code can take – `else-if` statements are used for selecting an appropriate path for the program to take so-to-speak.

The previous example demonstrated how we might check if a patient is too hot or not, we haven't been able to consider whether they are too cold or not though. To do this, we can use an else-if in addition to the existing statements to check for this:

```
if (temp > feverTemp) {
    output = "Patient too hot! Start emergency cooling
        procedures";
} else if (temp < minSafeTemp) {
    Output = "Patient too cold! Start emergency warming
        procedures";
} else {
    output = "Patient is fine";
}
```

11.5 Switch statements

A **switch statement** is like the previous conditional statements, except that we instead pass it a value for comparison against some values to select the correct

path of code to follow. Switch statements only work with the following data types:

- `byte`, `short`, `char` and `int`
- `Byte`, `Character`, `Short`, `Integer`
- Enumerated types (Enums)

Consider the following problem:

Given an integer between 1 and 12, representing the month of the year, output the appropriate month as a string to the console. For example, the input value of '3' would result in 'March' being printed to the console.

We could write a set of if-else-if-else statements to cover all the possible decisions, but the solution will be very verbose. Instead, we can use a switch statement to reduce the necessary amount of code:

```
1  int month = 3;
2
3  switch (month) {
4      case 1: System.out.println("January");
5              break;
6      case 2: System.out.println("February");
7              break;
8      case 3: System.out.println("March");
9              break; // etc...
10     default: System.out.println("Incorrect input");
11             break;
12 }
```

In the above example, we declare a switch statement that accepts a `month` variable as input, `switch (month)`. Following this is the body, known as the **switch block**, of the switch statement, the switch block contains the **cases** which are checked against the input value. The first case, `case 1:`, will check if the input variable `month` is equal to the value `1`. If it is equal, January will be printed to the console and then the `break` statement would be executed; a **break statement** terminates the enclosing switch statement and prevents it from continuing execution.

A break statement is necessary to prevent **falling-through**, a behaviour where once a case has been matched every statement in the cases after will run until a break statement is encountered. Try switching the value of month to 1 and removing the break statement from case 1 to see this behaviour.

The final part of a switch statement is the **default case**, which is optional to include. The default case specifies some code to run should none of the previous cases have matched the input.

11.6 Iteration

Iteration is a powerful concept in programming that allows for the repetition of a block of programming instructions. The noun iteration is generally used in one of two ways:

- To represent the process of repeating some mathematical or computational process, or set of instructions, such as incrementing a counter by 1 over and over.
- To represent a single repetition of some iterative process, such as a single incrementation of a counter by 1.

11.7 While statements

A **while statement** is a programming construct where an instruction is repeated only while some boolean expression evaluates to `true`. Generally, while loops are used when we don't know how many times we should iterate.

A while statement takes the following format:

```
while (boolean_expression_is_true) {
    // execute the code in here
}
```

> IMPORTANT: A while loop will only work with a boolean value as input, or an expression that evaluates to a boolean value.

Any variables declared inside the code block of the while loop are, like conditionals, **locally scoped** to that block - the variable cannot be accessed outside of the block:

```
while (true) {
    int i = 3;
}
System.out.println(i); // Error: cannot find symbol
```

In the example above, an error would be thrown at compile-time as the variable `i` is not accessible outside of the while loops block.

A while loop will also work with a boolean expression, the following example demonstrates creating a counter that will print the values 1 through 10 to the console:

```
int counter = 1;

while (counter <= 10) {
    System.out.println(counter);
    // increment counter by one before the next iteration of the
        loop
    counter++;
}
```

11.8 Do-while statements

A **do-while** statement is similar to a while loop, the major difference is that it will always execute at least once; a standard while loop will not execute even once if the boolean input is false.

A do-while loop takes the following format:

```
do {
    // code to execute
} while (boolean_expression_is_true);
```

Variables declared inside the `do { } block`, like a standard while loop, are **locally scoped**. The following example demonstrates an interactive loop, a way of getting input from the console and performing some action in response to that:

```
int counter = 1;
boolean isRunning = true;
// Create a scanner to get input from the console
Scanner sc = new Scanner();
String input;

do {
    System.out.println("\nWelcome to Counter\n");
    System.out.println("Enter 'R' to reset the counter,\n'P' to
       print and increment by 1,\nand 'Q' to quit");
    System.out.print("\n> ");

    // get input
    input = sc.nextLine();

    switch (input.toUpperCase()) {
        case "R":
```

```
17              counter = 1;
18              break;
19          case "P":
20              System.out.println("Counter: " + counter++);
21              break;
22          case "Q":
23              System.out.println("Goodbye");
24              isRunning = false;
25              break;
26          default:
27              System.out.println("Invalid input, please try again"
                    );
28              break;
29      }
30  } (isRunning);
```

11.9 For statements

A **for statement** is used to repeat some block of code when we know when to stop repeating that code. For example, the original counter example from the while statement section can be written more concisely with a for statement.

A for statement takes the following format:

```
1  for (initializiation; termination; increment) {
2      // code to repeat here
3  }
```

> The `initialization` section of the for statement will only run once, at the start of the loop.

Like while statements, variables declared inside the block are locally scoped, this also includes any variables declared in the `initialization` section. Each section must be separated by a semi-colon and have the following meaning:

- Initialization: Variables are declared here for use inside the loop, usually as an index variable for collections or a counter until some condition is met.
- Termination: A boolean expression under which the loop should continue executing.
- Increment: The expression which increments the loops counter, if any.

When a for statement runs, it runs in the following order:

1. Initialization runs once
2. Termination condition is checked (the code inside the loop will not be executed once if this evaluates to false on the first pass)
3. Execute code inside block
4. Increment section runs
5. Repeats from step 2 until step 2 evaluates to false

A simple for statement for counting from 1 to 10 would look like:

```
for (int i = 1; i <= 10; i++) {
    System.out.println(i);
}
```

> It is common for the variable which tracks the iteration of a loop to be named `i` for index.

11.10 Nested iterative statements

As iterative statements declare a block of code, an iterative statement can also be nested inside that. This most commonly occurs with for statements for iterating over nested arrays, but also has over use cases.

The following example demonstrates a nested for statement which prints the times tables out to the console:

```
for (int i = 1; i <= 12; i++) {
    System.out.println(i + " times table");

    for (int j = 1; j <= 12; j++) {
        System.out.print((i * j) + " ");
    }
    System.out.println("\n");
}
```

This produces output similar to the following:

```
1 times table
1 2 3 4 5 6 7 8 9 10 11 12

2 times table
2 4 6 8 10 12 14 16 18 20 22 24

etc...
```

11.11 A note on infinite loops

Should a loops termination expression never evaluate to false, an infinite loop will occur in which the same instructions are executed repeatedly until the computer runs out of memory, crashes, powers off, someone intervenes and force closes the program or some other outer condition which could cause the program to stop running. For example, the following code would print 1 to the console over and over unless manually stopped:

```
1 int counter = 1;
2
3 while (counter <= 10) {
4     System.out.println(i);
5 }
```

As counter is never incremented by 1, the expression counter <= 10 never evaluates to false and thus the loop never stops executing.

We can also create infinite loops with `for` statements by forgetting the increment. Here is a not so obvious, but also obvious, example of an infinitely looping for statement:

```
1 for (;;) {
2     System.out.println("INFNITE LOOP");
3 }
```

12 Object-oriented programming principles

Object-oriented programming has four key principles which enable us to create extensable software, these are:

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

12.1 Encapsulation

The principle of **encapsulation** states that a module (a class in Java) is encapsulated if either of the following conditions are met:

- The data of a module is bundled with the behaviours (methods) that act upon that data.
- A modules components accessibility from other modules is restricted

The simplest way to represent encapsulation in Java is with a POJO (Plain Old Java Object) class representing a domain entity, such as a user. The first thing we should do is secure the fields with the private access modifier:

```
1  public class User {
2      private String forename;
3      private String surname;
4      private int age;
5  }
```

To then allow access to the data, and a way of setting the data, provide accessor and mutator methods known as getters and setters respectively:

```
1  public class User {
2      private String forename;
3      private int age;
4
5      public String getForename() {
6          return this.forename;
7      }
8
9      public void setForename(String forename) {
10         this.forename = forename;
11     }
12
13     public int getAge() {
14         return this.age;
15     }
16
17     public void setAge(int age) {
18         if (age < 18) {
19             throw new Exception("Age must be 18 or over");
20         } else {
21             this.age = age;
22         }
23     }
24 }
```

The encapsulated User class can now be used as follows:

```
User fred = new User();
fred.setForename("Fred");
fred.setAge(47);

System.out.println(fred.getForename()); // Fred

User bob = new User();
bob.setForename("bob");
bob.setAge(17); // Exception thrown: Age must be 18 or over
```

> The line throw new `Exception("Age must be 18 or over");` will cause the program to crash unless handled. Exception handling is explored in a later section.

12.2 Inheritance

The pillar of **inheritance** states that a module A will inherit attributes and behaviours from another module B. In Java, a class can inherit from only one other class - this is known as **single inheritance**. This allows for complex hierarchies of objects to be created, to understand how to build these we must understand some terminology:

- **Superclass**: A superclass, also known as a *base class* or *parent class*, is the class in which attributes and behaviours are inherited from.
- **Subclass**: A subclass, also known as a *derived class* or *child class*, is the class which inherits attributes and behaviours from a superclass.

Class inheritance in Java uses extends keyword to indicate that inheritance is taking place, the syntax looks as follows:

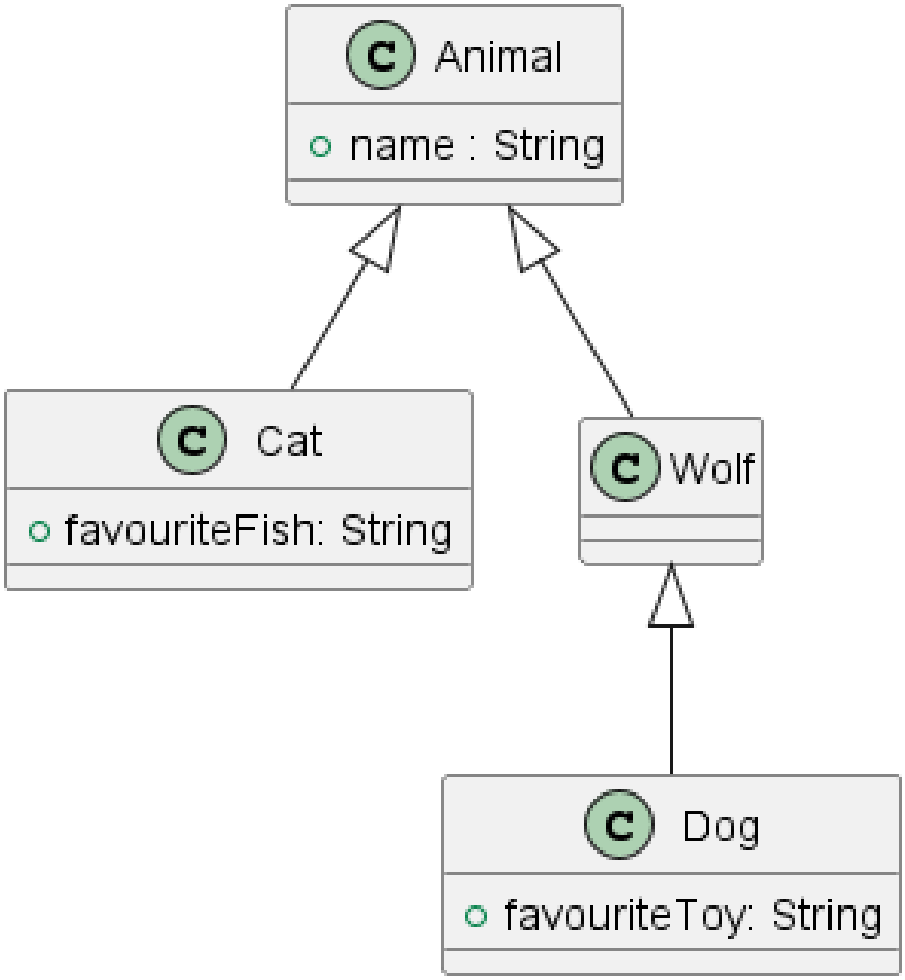
```
class A extends B { }
```

In Java, only non-private class members (except constructors) are inherited. The rules are as follows according to their access modifiers:

- Private fields and methods are not inherited.
- Package-private (default) fields and methods will only be inherited by subclasses in the same package as their superclass.
- Protected fields and methods will be inherited by subclasses and accessible regardless of the subclasses package.

- Public fields and methods will be inherited by subclasses and accessible regardless of the subclasses package.

The following example demonstrates a simple inheritance hierarchy between animals:



It can be said in inheritance that if a class B inherits from class A, class B is-a instance of class A but not the other way around.

In the above diagram:

- `Animal` is the supertype of all animal types
- `Cat` is-a `Animal`
- `Dog` is-a `Wolf`

- `Wolf` is-a `Animal`
- As a `Wolf` is-a `Animal`, a `Dog` is-a `Animal` but a `Dog` is-not-a `Wolf`
- A `Cat` is-not-a `Wolf`

The following code demonstrates creating a simple inheritance hierarchy:

```
1  class Animal {
2
3      private String name;
4
5      public Animal(String name) {
6          this.name = name;
7      }
8
9      public void setName(String name) {
10         this.name = name;
11     }
12
13     public String getName() {
14         return this.name;
15     }
16 }
17
18 class Cat extends Animal {
19
20     public Cat() {
21         super("Unknown");
22     }
23
24     public Cat(String name) {
25         super(name);
26     }
27
28     public void meow() {
29         System.out.println(getName() + " meowed");
30     }
31 }
```

The above example demonstrates a `Cat` class inheriting from an `Animal` class, the `Cat` class specifically inherits the `setName()` and `getName()` methods as they are `public`, nothing else is inherited from `Animal`.

> As constructors are not inherited and the `Animal` class did not have an empty 0-parameter constructor, the `Cat` class had to specify a constructor or else an error would be thrown.

The `Cat` classes constructor that takes a string as an argument uses the `super`

keyword to call the superclasses constructor that takes a single string as input - when doing this, a constructor must exist in the superclass which accepts the same number and types of parameters specified. In the above example, the call `super(name)` calls the constructor `public Animal(String name) {}` to initialise the `name` field in the `Animal` class.

The `Cat` class also defines its own custom `meow()` method which makes use of the inherited getter method to access its names value and print out a message to the console. These classes can be used as follows:

```
Animal animal = new Animal("Fred");
System.out.println(animal.getName()); // Fred

Cat cat = new Cat("Bob");
cat.meow(); // Bob meowed

animal.meow(); // Error: cannot find symbol: method meow()
```

12.3 Overriding methods

Any class which inherits non-private methods becomes a candidate for **dynamic method dispatch**, or what is more commonly called *method overriding* in Java. This allows a subclass to override a method from its superclass, and when that method is called it uses the overriden one.

Let's use the `Animal` example. Let's assume all animals could make some noise (a single noise to keep it simple for now). Instead of each animal type having a custom method such as `woof()` for `Dog` and `meow()` for `Cat`, we can create a default `noise()` method which can be overriden by a subclass:

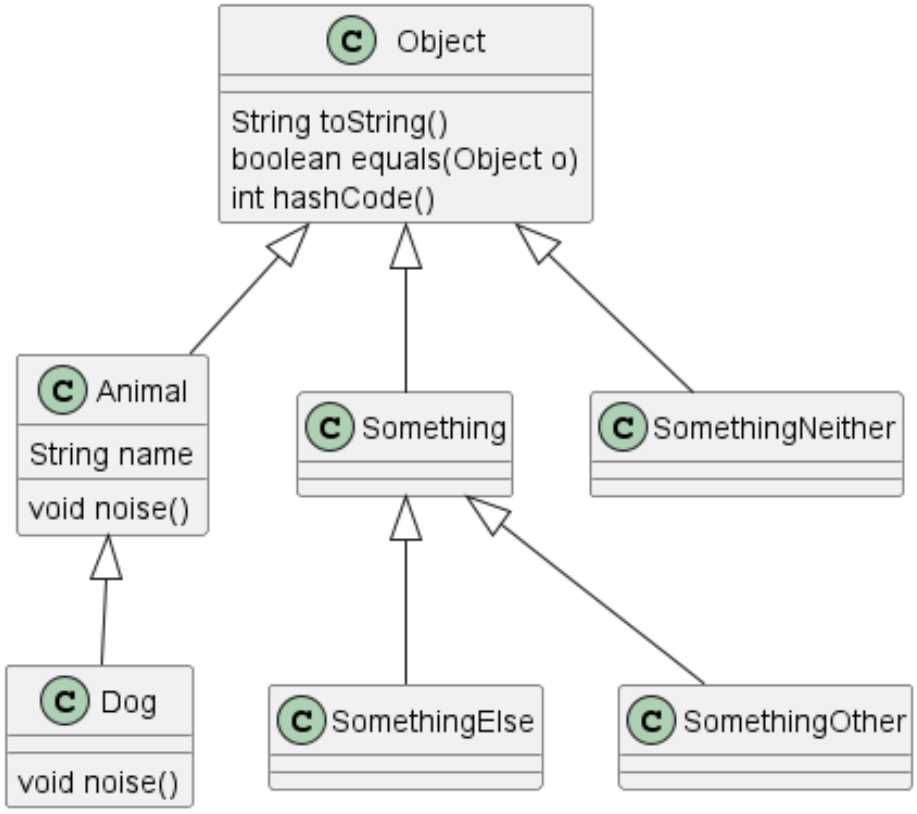
```
class Animal {
    public void noise() {
    }
}

class Dog extends Animal {
    private int hungerLevel = 100;

    @Override
    public void noise() {
        if (hungerLevel >= 50) {
            System.out.println("WOOF WOOF WOOF WOOF WOOF WOOF!!!
                ");
        } else {
            System.out.println("Woof woof!");
```

```
15            }
16        }
17    }
```

Here, we create a `Dog` class which inherits from `Animal` and overrides the `noise()` method. We also use the optional `@Override` annotation to indicate to the compiler and reader that this method has been overriden. You may have noticed in the above example that the `noise()` method in the `Animal` class has a method body but isn't used... We have a solution for that, using abstraction and polymorphism - more on that soon.

12.4 Everything in Java is an `Object`

In Java, every value that we use, aside from the 8 primitive types, is an object - they all inherit from a class called `Object`, whether directly or implicitly, we can visualise this using UML:



Our types start to form hierarchies as we create them and extend them through inheritance.

The Object class documentation shows that all objects also have a basic set of methods, ones of interest for beginners to Java include:

- `toString()`: Returns a String representation of an object
- `equals(Object o)`: Returns a boolean indicating whether the objects are equal (`true`) or not (`false`)
- `hashCode()`: Returns a hash code value for the object, used for built-in data types backed by hash tables for efficiency purposes

Here is an example of overriding the `toString()` method to return a custom string representation of the object:

```
class Animal {
    String type;
    int age;

    @Override
    public String toString() {
        return "ANIMAL: type(" + type + "), age(" + age + ")";
    }
}
```

12.5 Polymorphism

Polymorphism is used in programming and type theory to represent multiple different types under a single *interface*. An **interface** is just some contract which defines how something behaves. For example, we could have an `Animal` class which defines the base behaviours for all animals:

```
class Animal {
    public void move() {}
    public void eat() {}
    public void sleep() {}
    public void noise() {}
}
```

Each class inheriting from `Animal` could then override these methods (dynamic method dispatch is a form of polymorphic behaviour as well):

```
class Cat extends Animal {
    @Override
```

```
    public void noise() {
        System.out.println("Meow");
    }

    public void catSpecific() {
        System.out.println("I am cat");
    }
}
```

The interesting part is how we can use the `noise()` method:

```
Animal animal = new Cat();
animal.noise(); // Meow
```

Here, we have created an instance of `Cat` and stored the reference to it in an `Animal` variable. There are two forms of polymorphism here, runtime method overriding being one and *subtype polymorphism*. **Subtype polymorphism** is the ability for a parent type to hold objects of its child types - using the `Animal` example, we can store a `Cat` object in an `Animal` variable because a `Cat` is a `Animal`.

Another kind of polymorphism is **ad-hoc polymorphism**, or more commonly called *method overloading* in Java. This is where we can multiple methods with the same name, but different input type signatures:

```
class Calculator {
    int add(int a, int b) { return a + b; }
    double add(double a, double b) { return a + b; }

    int truncate(float f) { return (int) f; }
    int truncate(double d) { return (int) d; }
}
```

It is perfectly valid to have two methods with the same name, in the same class, as long as their parameters have different types.

12.6 Abstraction

Abstraction is a pervasive concept in computer science and mathematics, holding various meanings and having many different forms of representation. Put simply though, abstraction can be seen as **modelling** something in a *generalised* way. For example, when a new car is designed, it will have a steering wheel in front of the driver and pedals on the floor of the car in front of them; the steering wheel is

used to turn the wheels of the car and the pedals to control its speed and gears. The idea here is that each of these is an abstraction, hiding a tonne of implementation complexities. When we get in a car to drive it, we only care that it has a steering wheel; we don't generally care about how it is actually built and wired into the car and such. Abstractions are everywhere, the `Animal` class from before is one - well, almost...

Java has the `abstract` keyword for representing abstractions, let's see how it is used:

```
abstract class Animal {
    abstract public void move();
    abstract public void eat();
    abstract public void noise();

    public void sleep() {
        System.out.println("zzz...");
    }
}
```

First of all, we have applied the `abstract` modifier to the class declaration. This means we cannot create instances of the class directly any more, like this: `Animal a = new Animal();`. There are anonymous classes, but they are beyond the scope of this section. We also have applied the `abstract` modifier to 3 methods, each only having a method header - the body is omitted, this is mandatory when `abstract` is specified. The `sleep()` method is not absract, and provides a default implementation. An implementation class might then look like:

```
class Dog extends Animal {
    @Override
    public void move() {
        System.out.println("Moving");
    }

    @Override
    public void eat() {
        System.out.println("Eating");
    }

    @Override
    public void noise() {
        woof(2);
    }

    public void woof(int times) {
        for (int i = 0; i < times; i++) {
```

```
19                 System.out.println("Woof");
20             }
21         }
22 }
```

We have overriden the three abstract methods in the first **concrete-class** which implements the abstract class, either directly or indirectly. A concrete-class is just a none-abstract class, and **must** implement any abstract methods defined in a superclass. We can use this `Dog` class as follows:

```
1  Dog dog = new Dog();
2  dog.move(); // moving
3  dog.sleep(); // zzz...
4  dog.noise(); // Woof Woof
5  dog.woof(2); // Woof Woof
6
7  Animal animal = dog;
8  animal.sleep(); // zzz...
9  animal.noise(); // Woof Woof
10 animal.woof(2); // No such method compile-time exception
```

The main issue with abstraction is that we lose implementation detail, but abstraction helps make our system more maintainable, extendable and easier to reason about. The `Animal` class has no awareness of the `Dog` classes `woof(int times)` method, and thus when we try to compile the program an exception occurs. If we look though, we could call the `noise()` method on an object in an `Animal` variable which thanks to method overriding, calls the `noise()` method of the `Dog` object referenced by the `animal` variable. This `noise()` method in the `Dog` class is still capable of calling the `woof(int times)` method as it is in scope for that overriden method.

12.7 Interfaces

An **interface** in Java is a construct for declaring abstract types. We declare an interface with the `interface` keyword:

```
1 interface DataType {}
```

The rules for declaring and defining an interface are as follows:

1. All methods are implicitly `public` and `abstract` (pre Java 8), from Java 8 we can also specify `default` methods in interfaces.

2. Instance variables are not allowed, only public, static constant variables are which use the modifiers `static`, `public` and `final`.
3. Interface methods can not be static (pre Java 8), from Java 8 onwards we can specify static methods in an interface.
4. Interface methods can not be `final`, `strictfp` or `native`.
5. Interfaces cannot extend classes, but they can extend other interfaces. Interfaces support multiple inheritance whereas classes do not.

An interface is quite similar to an abstract class, the main difference is that we cannot have instance variables and constructors. Like abstract classes though, interfaces cannot be instantiated unless providing an anonymous implementation.

Here is an example interface defining a `KeyPair` data type and the basic operations we might want to support:

```
1 interface KeyPair {
2     String getKey();
3     String getValue();
4
5     default String getKeyValuePair() {
6         return getKey() + ":" + getValue();
7     }
8 }
```

We have three methods in this data type:

- `getKey()` and `getValue()` are both public and abstract, so we don't specify a body. The inheriting type will override these methods and provide an implementation.
- `getKeyValuePair()` is a `default` method, meaning it has a default implementation that all inheriting types will inherit. This method is also implicitly public.

An implementing class might look like:

```
1  class ConcreteKeyPair implements KeyPair {
2      private String key;
3      private String value;
4
5      public ConcreteKeyPair(String key, String value) {
6          this.key = key;
7          this.value = value;
8      }
9
10     @Override
```

```
11      public String getKey() {
12          return key;
13      }
14
15      @Override
16      public String getValue() {
17          return value;
18      }
19  }
```

Take note of how we inherit from an interface, we use the `implements` keyword followed by a comma-separated list of interface types:

```
1  class Foo implements Bar, FooBar {}
```

Using our new implementation of the interface would look like:

```
1  KeyPair pair = new KeyPair("morgan", "j98sf9adj899j0j090");
2  System.out.println(pair.getKeyValuePair()); // default method
       inherited from KeyPair
3  System.out.println(pair.getKey()); // overridden version of
       method, in ConcreteKeyPair, inherited from KeyPair
```

If we want a class to inherit from another class and implement some interfaces, our type signature would look something like:

```
1  class Foo extends AbstractFoo implements Bar, FooBar {}
```

We can also make an interface inherit from another interface using the `extends` keyword, similar to how a class might inherit from another class:

```
1  interface Foo extends Bar, FooBar {}
```

13 Exception handling

13.1 What is an exception?

As Oracle states in their guide, an **exception** is:

> An event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
>
> (Oracle, What is an Exception?)

More specifically, this means an exception is something that can cause our program to stop executing - this is known as an exceptional event.

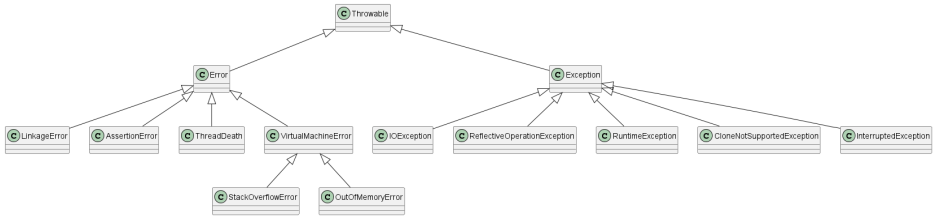
Exceptions occur when an error happens within a method, the method will then create an **exception object** containing:

- information about the error
- the type of the error
- the programs state when the error occurred

The exception object will then be handed to the Java runtime system in a process known as throwing an exception.

13.2 The Exception hierarchy

The Java exception hierarchy looks as follows:



At the root of the hiearchy is the `Throwable` superclass, which represents all exceptions and errors in our program. An **exception** is a recoverable state where some erroneous condition happened, but was not serious enough to crash the JVM. An **error** is a serious, non-recoverable state which causes the JVM to stop execution.

13.3 Checked vs Unchecked exceptions

Exceptions are split into two general groups, *checked* and *unchecked*. A **checked** exception is one which inherits from `Exception` but is not a `RuntimeException`. If our program contains a checked exception, we must handle it or the program will not compile. An **unchecked** exception is any `RuntimeException` that the app may throw, and does not have to be handled for the application to run.

- Checked exceptions generally represent error conditions which we know will occur, and thus must be handled.
- Unchecked exceptions generally represent error conditions that may or may not occur dependent on programming practice, i.e., whether you have misused the API or not.

To handle exceptions, we use a `try` block with some continuation block following it:

```
1 int[] arr = new int[10];
2
3 try {
4     // The code which might throw an exception
5     arr[10] = 20;
6 }
```

When we expect that an exception may occur, we wrap the code which could cause it in the `try` block. A `try` block cannot exist by itself though, so we also specify a `catch` block (or many) for catching the exceptions. In this case, we will get an `ArrayIndexOutOfBoundsException`; an unchecked `RuntimeException`:

```
1 int[] arr = new int[10];
2
3 try {
4     // The code which might throw an exception
5     arr[10] = 20;
6 } catch (ArrayIndexOutOfBoundException e) {
7     e.printStackTrace();
8 }
```

Here, we get an exception object as input to the catch block if that specific exception type was thrown. In this case, we ask for the method stack trace to be printed to standard out; a stack trace being all the method calls that lead upto the exceptional event.

We can also chain `catch` blocks to handle multiple different exceptions, from

most to least specific:

```
int[] arr = new int[10];

try {
    // The code which might throw an exception
    arr[10] = 20;
} catch (ArrayIndexOutOfBoundsException e) {
    e.printStackTrace();
} catch (Exception e) {
    // default handling, catches all checked and unchecked
        exceptions
}
```

We have added a `catch` block below the existing one, which catches a less specific `Exception` instance; thanks to inheritance, this means we could catch the `ArrayIndexOutOfBoundsException` using just the last `catch` block:

```
int[] arr = new int[10];

try {
    // The code which might throw an exception
    arr[10] = 20;
} catch (Exception e) {
    // default handling, catches all checked and unchecked
        exceptions
}
```

13.4 Throwing exceptions

When writing our code, it will often be the case that we need to consider bad program states; For example, what happens if the user enters invalid data, how do we handle that? We can throw an exception to indicate a bad program state (exceptional condition) using the `throws` keyword and passing it an `Exception` object instance:

```
int numerator = 30;
int dividend = 0;

if (dividend == 0) throw new ArithmeticException("Cannot divide
    by 0");
```

In this example, we throw an `ArithmeticException` (a type of `RuntimeException`) if the `dividend` of a division is 0; this is because we cannot divide by 0 using the normal laws of mathematics.

13.5 The handle-or-declare rule

The handle or declare rule is a way of thinking about checked exceptions specifically. It is the case that checked exceptions must be handled using a `try` block, but what if we don't want to **handle** a specific exception where it occurs; maybe we want to let the calling method handle it instead. For example:

```
class UserRepository {
    User getUserById(int id) {
        throw new Exception("User with id " + id + " not found!"
            );
    }
}

class App {
    public static void main(String[] args) {
        UserRepository repo = new UserRepository();

        try {
            repo.getUserById(30);
        } catch (Exception e) {
            // do something to handle it
        }
    }
}
```

The issue here is that we have used a checked exception but not handled it in `UserRepository.getUserById()`, so this program would never compile. We must **declare** that the method throws a checked exception if it does not handle it itself:

```
class UserRepository {
    User getUserById(int id) throws Exception {
        throw new Exception("User with id " + id + " not found!"
            );
    }
}

class App {
    public static void main(String[] args) {
        UserRepository repo = new UserRepository();

        try {
            repo.getUserById(30);
        } catch (Exception e) {
            // do something to handle it
        }
```

```
    }
}
```

13.6 Defining a custom exception

To define a custom exception, extend an existing `Exception` type in the hierarchy. For example, let's create a checked `UserNotFoundException`:

```
class UserNotFoundException extends Exception {
    public UserNotFoundException(int id) {
        super("User not found with id " + id ".");
    }
}
```

In the constructor, we accept an ID and then call the parent classes constructor with the exceptions message. Using this in our code will then look like:

```
class UserRepository {
    User getUserById(int id) throws UserNotFoundException {
        throw new UserNotFoundException(id);
    }
}

class App {
    public static void main(String[] args) {
        UserRepository repo = new UserRepository();

        try {
            repo.getUserById(30);
        } catch (UserNotFoundException e) {
            // do something to handle it
        }
    }
}
```

13.7 The `finally` block

The `finally` block can be attached to a `try` block, and can replace the `catch` block although it cannot catch an exception. We use the `finally` block for cleaning up resources after we are done using them, this might look like:

```
Scanner sc = null;

try {
```

```
4       sc = new Scanner(new File("/home/morgan/non_existant_file.
            txt"));
5       // use scanner
6   } catch (Exception e) {
7       // some exception that might be thrown when using the
            scanner
8   } finally {
9       // close the scanner
10      if (sc != null) {
11          try {
12              sc.close();
13          } catch (Exception e) {
14              // there may be an exception from closing the
                    scanner
15          }
16      }
17  }
```

Here, we create a `Scanner` object and use it to connect to a file. In the `finally` block, which always runs after all `try` and `catch` blocks have executed, we then check if the `Scanner` object is not null. If it isn't null, we should attempt to close it. It is important to remember that the `finally` block will always be executed, unless a fatal error crashes the JVM that is...

13.8 Try-with-resources

The `try-with-resources` block, introduced in Java 7, allows for us to use objects which implement the `AutoCloseable` interface with the guarantee that they will be closed without us manually having to trigger the process.

To demonstrate why we need such a block, let's first consider how we might typically close a `Scanner`:

```
1   Scanner sc = null;
2
3   try {
4       sc = new Scanner(new File("/home/morgan/non_existant_file.
            txt"));
5       // use scanner
6   } catch (Exception e) {
7       // some exception that might be thrown when using the
            scanner
8   } finally {
9       // close the scanner
10      if (sc != null) {
```

```
11          try {
12              sc.close();
13          } catch (Exception e) {
14              // there may be an exception from closing the
                    scanner
15          }
16      }
17  }
```

It's a lot of work to do this. The try-with-resources block takes some `AutoCloseable` implementations as arguments, somewhat like a method call (it is not a method, do not confuse it):

```
1  try (Scanner sc = new Scanner(new File("/home/morgan/nef.txt")))
       {
2      // use scanner
3  } catch (Exception e) {}
```

We no longer need to manually close `close()` on the scanner, another bonus is that the `Scanner` is scoped to the try block rather than whatever method it is contained in.

14 Collections and Maps

14.1 Generic type parameters

The built-in collection types in Java make use of *generics*, which is basically a way of making a method or data type reusable across many different data types. The array data type introduced previously is a good example of this. It didn't matter what we stored in the array, we specified a type and the computer stored it as we asked.

Going forward, we will see a new syntax when working with Java:

```
1 List<String> strings = List.of("Hello", "World");
```

If we look at the variable declaration, we will notice some angle brackets after the `List` keyword. The `List` interface in Java is actually defined as `interface List<E>`, the `E` is a type parameter. We pass the type parameter a value when declaring the data type of the variable to indicate what the `List` is dealing with. So we could declare a list of integers in the same way and then use all the same common operations without worrying about the implementation:

```
1 List<Integer> numbers = List.of(10, 20, 30, 40);
```

14.2 The Collection interface

The `Collection` interface is the abstract data type representing some group of elements, it is the default interface for most built-in collection data structures in the JDK (Collection interface | Oracle).

The collection interface is defined as:

```
1 interface Collection<E> extends Iterable<E> { /* operations */ }
```

The `Iterable<E>` type is what allows us to use an enhanced for-loop over an `Array`, as well as many other collection types such as `List` or `Set` implementations. The `Collection<E>` interfaces declares a few key methods which all collection types will implement, aside from operations which modify state on immutable collections. Here are some of them:

Method	Description

Continued on next page

Continued from previous page

Method	Description
`boolean add(E e)`	Adds an element to the collection, returning true if successfully added or false otherwise. This is an optional operation for implementors due to immutable collections.
`boolean addAll(Collection<? extends E> c)`	As long as the collection `c` is of the same type as the collection this method is called on, it will add all items of `c` to the collection.
`boolean contains(Object o)`	Returns `true` if the element `o` is in the collection.
`void clear()`	Removes all elements from the collection, optional due to immutable collections.
`boolean isEmpty()`	Returns `true` if the collection is empty.
`boolean remove(Object o)`	Removes a single instance of an object `o` from the collection, returns `true` if successful. Optional operation due to immutable collections.
`int size()`	Returns the number of items in the collections
`Iterator<E> iterator()`	Returns an iterator object which allows for iterating over the elements in the collection.

We could use the collection type to store lists of data and then iterate over it:

```
1 import java.util.Collection;
2 import java.util.List;
3
4 Collection<Integer> numbers = List.of(1, 2, 3, 4, 5);
5
6 for (Integer number : numbers) {
7     System.out.println(number);
8 }
```

Which will output:

```
1
2
```

```
3
4
5
```

You can find out more about the `Collection` data types operations, including the ones we haven't used, at the Oracle documentation.

14.3 The List interface

The `List<E>` interface implements the `Collection<E>` interface, which also means it implements the `Iterable<E>` interface. As you may have saw in the previous section, we stored a `List` instance in a `Collection` variable and then iterated over it using an enhanced for loop. The `List` interface defines an ordered collection data type where the user can control the position of each element in the list. You can learn more about the `List` interface using the Oracle documentation.

> Some implementations of the `List` data type will be immutable, which will usually cause an exception to be thrown if you try to mutate the data.

In addition to those methods defined in the `Collection<E>` interface, we also now have:

Method	Description
`void add(int index, E element)`	Inserts the specified `element` at given `index`. Optional for implementations due to possibility of immutability.
`E get(int index)`	Returns the element at the specified `index`.
`int indexOf(Object o)`	Returns the index of the specified element `o` if it is in the list, otherwise returns `-1` to indicate the element is not present.
`lastIndexOf(Object o)`	Returns the index of the last occurrence of the element `o` if present in the list, otherwise returns `-1` if the list does not contain the specified element.
`E remove(Object o)`	Removes the first occurrence of the element `o` if present, returning it. Optional due to possible immutability.

Continued on next page

Continued from previous page

Method	Description
`E set(int index, E element)`	Replaces the element in the list at the given `index` with the specified `element`.

We could use the `List` interface like so:

```
import java.util.List;
import java.util.ArrayList;

List<Integer> numbers = new ArrayList<>(List.of(1, 2, 3, 4, 5));

int firstElement = numbers.get(0);
int lastElement = numbers.get(numbers.size() - 1);

System.out.println("INDEX 0: " + firstElement);
System.out.println("INDEX " + (numbers.size() - 1) + ": " +
    lastElement);

numbers.add(6); // append
numbers.add(0, 0); // shift
numbers.add(32); // oops
numbers.set(numbers.size() - 1, 7); // better, back in order
numbers.remove(0);

System.out.println("NUMBERS:");
for (int i = 0; i < numbers.size(); i++) {
    System.out.println("INDEX " + i + ": " + numbers.get(i));
}
```

This will print the following the output to standard out:

```
INDEX 0: 1
INDEX 4: 5
NUMBERS:
INDEX 0: 1
INDEX 1: 2
INDEX 2: 3
INDEX 3: 4
INDEX 4: 5
INDEX 5: 6
INDEX 6: 7
```

14.4 The ArrayList class

The `ArrayList<E>` class implements the `List<E>` interface, it is the first concrete data structure we will use which implements the `List` abstract data type. The key to an `ArrayList` is that its data is stored in an `Array` behind the scenes, this array is then resized as needed. Do not worry about efficiency, the algorithms behind the scenes making it work are well optimised.

> If you really want to know, the `ArrayList` has a constant time complexity for getting and settings items - it is fast at this. Adding and removing elements is not so fast depending on the circumstance. Appending to the end of an `ArrayList` is practically constant time due to an **amortised** constant time complexity, even though an append may cause the array to resize, as long as the resize uses a constant factor (typically 1.5 - 2.0) the resizes generally won't affect the runtime of the operation by any worrying amount. Adding and removing elements is less clear cut... If we add an element to the start of the array, we have to iterate over every element and shift it one to the right. This is the same deal for removing an element from the start, except that we are shifting each element to the left instead. This is the worst scenario, which represents a linear time complexity for inserting and removing elements at an arbitrary position. If you desire constant time complexity for inserting new elements and removing existing elements, use the `LinkedList` data structure instead.

The `ArrayList` was demonstrated in the `List` interface section, refer back to see.

14.5 The Set interface

In mathematics, a **set** is a collection of unique items known as its **elements** or **members**. Sets are a ubiquitous concept in modern mathematics, especially in the field of discrete mathematics when it comes to finite and thus, countable sets. The key tenant of discrete mathematics is dealing with a discrete range of numbers. Discrete mathematics is also the required background knowledge for the study of data structures and algorithms.

Anyway, a set in mathematics is typically denoted by an italic, capital letter. We may represent a set using **roster notation**:

$\phi = \{\}$

$\mathbb{Z} = \{..., -3, -2, -1, 0, 1, 2, 3, ...\}$

$\mathbb{N} = \{1, 2, 3, 4, 5, 6, ...\}$

$\mathbb{N} = \{0, 1, 2, 3, 4, 5, ...\}$

- The greek letter ϕ (phi) represents the null/empty set
- The letter $\mathbb{Z}$ represents the set of all integers (whole numbers)
- $\mathbb{N}$ is the set of all natural numbers, a positive integer. Some fields of mathematics also include 0 as a member of the set all natural numbers.

The **membership** symbol for sets represents whether an element *is a member of* a set or not. The $\in$ symbol represents an element being a member of a set, the $\notin$ represents when an element is not a member of a set. For the following example, we represent whether a number is a member of one of the prior mentioned sets or not:

$4 \in \mathbb{Z}$

$-1 \in \mathbb{Z}$

$-1 \notin \mathbb{N}$

$1 \in \mathbb{N}$

> A set does not contain duplicates.

Now, enough about the mathematical side - should you choose to research sets, you have a starting point!

A set in Java is represented by the `Set` interface, and is also a collection which does not contain duplicate elements (documentation). It has a type declaration as follows:

```
1 interface Set<E> extends Collection<E> {}
```

A `Set` is a collection in Java, so it has all of the usual collection methods. We should take care when choosing an impelementation of this abstract data type, some implementations do not retain the insertion order like a `List` does; the `HashSet` is an example of a set which does not retain insertion order. If we wanted to retain insertion order, a `LinkedHashSet` would be the appropriate choice. A `HashSet` is backed by a `HashMap` data structure in Java, which is backed by arrays. A `LinkedHashSet` is backed by a linked list implementation.

Beyond the original method descriptions from the `Collection` interface, there is not much to tell. The `add(E e)` method will only add the element `e` to the set if it is not already present and is not an immutable set, the same can be said for the `addAll(Collection<? extends E> c)` method inherited from the `Collection` abstract data type.

> The most notable thing about the `Set` interface is what it implies for the `add`, `equals` and `hashCode` methods and its constructors. The stipulations for the `add` method and the constructors is that they must not create a set with duplicate elements, that is it. The `equals` and `hashCode` methods are used for efficient storage and access of a sets members internally.

The following example demonstrates using a `HashSet` by adding and removing a variety of objects, then iterating over it and printing its contents:

```
1  import java.util.HashSet; // import goes above the class
2
3  HashSet<String> pcCompanies = new HashSet<>();
4  pcCompanies.add("Dell");
5  pcCompanies.add("HP");
6  pcCompanies.add("Lenovo");
7  pcCompanies.add("oops");
8  pcCompanies.remove("oops");
9
10 for (String company : pcCompanies) {
11     System.out.println(company);
12 }
```

14.6 The Map interface

In the field of mathematics, there doesn't exist a map data structure like in computer science; a structure of key value pairs all cobbled together as one unit. Instead, we have mappings (known also as a **morphism**) represented by functions in the form:

$$f : X \to Y \tag{14.1}$$

What this represents is a function f which maps values between the two sets X and Y where X is called the **domain** (the source of the mapping) and Y is called the **codomain** (the target values). We depict a mapping/morphism using an arrow, as seen above. Let's take a quick look at an example of its usage:

$X = \{a, b, c, d\}$

$Y = \{1, 2, 3, 4\}$

$f(a) \rightarrow 1$

$f(b) \rightarrow 2$

$f(c) \rightarrow 3$

$f(d) \rightarrow 4$

What the above shows is that the function f applied to some input x, a value in the set X, has a corresponding mapping in Y. So $f(a)$ is mapping from a to 1 where $a \in X$ and $1 \in Y$. This only briefly touches upon the idea of mappings, but now, onto the code.

A `Map` in Java is a data type which represents a mapping between keys and values (Oracle documentation). It provides three **collection views** through specific methods which allow the contents of the `Map` to be viewed as a:

- `Set` of keys
- `Collection` of values
- `Set` of key-value mappings

The `Map`'s type declaration looks as follows:

```
1 interface Map<K,V> {}
```

- `K` is the type of the keys
- `V` is the type of the values

As with the `Set` data type, the behaviour of a `Map` is implementation dependent; the `LinkedHashMap` class will retain insertion order whereas the `HashMap` will not. The `Map` interface in Java has a variety of useful methods, here are some of them:

Method	Description
`clear()`	Removes all mappings from the a map, optional if the map is immutable.
`containsKey(Object key)`	Returns `true` if a map contains the given `key`.

Continued on next page

Continued from previous page

Method	Description
`containsValue(Object value)`	Returns `true` if a map contains the given `value`.
`entrySet()`	Returns a `Set` collection view of the mappings in a map.
`get(Object key)`	Returns the associated value of the specified `key`, or null if the given `key` does not exist.
`isEmpty()`	Returns `true` if the map is empty.
`put(K key, V value)`	This operation will only fail on immutable maps. If the `key` does not exist in the map, it will be created first. Then the `given` key will be associated with the given `value`.
`remove(Object key)`	This operation removes the key:value mapping from a map if the given `key` is present. Optional as the map may be immutable.
`size()`	This operation returns the amount of key:value mappings in a map.
`values()`	Returns a `Collection` collection view of all values in a map.
`keySet()`	Returns a `Set` collection view of all keys in a map.

Let's now use a `HashMap` implementation of the `Map` interface to map questions to answers:

```
1  import java.util.Scanner;
2  import java.util.Map;
3  import java.util.HashMap; // above class
4
5  Scanner sc = new Scanner(System.in); // for console input
6
7  Map<String, String> questionBank = new HashMap<>();
8  questionBank.put("What is the capital of England?", "London");
9  questionBank.put("Who created the relational model for database
       management?", "Edgar Codd");
10 questionBank.put("Who created the SQL (Structured English Query
       Language)?", "Donald Chamberlin and Raymond Boyce");
11
12 Set<String> questions = questionBank.keySet();
```

```
for (String question : questions) {
    System.out.println(question)
    if (sc.nextLine().equals(questionBank.get(question))) {
        System.out.println("Correct!");
    } else System.out.println("Incorrect");
    System.out.println("---");
}
```

This simple program utilises a good chunk of the `Map` interfaces methods, and hopefully demonstrates how we might consider using them in our programs.

14.7 What's the deal with `HashMap` and `HashSet`, what's the `Hash` about?

The `HashMap` and `HashSet` classes make use of the `hashCode()` and `equals()` methods of objects in Java. Let's explore what these two methods are really for...

When we want to compare two objects, we need to use the `equals()` method instead of the == operator:

```
class User {
    String username;

    public User(String username) { this.username = username; }
}

// main() method
User user1 = new User("Bobby123");
User user2 = new User("Bobby123");
boolean duplicate = user1.equals(user2);
```

Given this example, it would be reasonable to expect `duplicate` to be set to `true`... By default though, that is wrong in Java; `duplicate` will in fact be set to `false` as the default implementation of `equals()`, defined in the `Object` supertype, states that *equality is the same as object identity*. This means it will compare the objects identities by default and is the reason why `user1` and `user2` are not equal. We can override the `equals()` method to fix this:

```
class User {
    // omitted for brevity, as above

    @Override
    public boolean equals(Object o) {
        if (o == this) return true; // identity check
```

```
7           if (!(o instanceof User)) return false; // type check
8           // value check
9           User user = (User) o;
10          return user.username.equals(this.username);
11      }
12  }
```

The Java specification states the following criteria to fulfill when implementing the `equals()` method:

- **Reflexive**: An object must be equal to itself, i.e., `o.equals(o)` must return `true`
- **Symmetric**: Two objects must be equal to each other, regardless of which called `equals()` on the other, i.e., `o.equals(u)` is the same as `u.equals(o)`.
- **Transitive**: Transitive here implies an object can be equal to another if they are both equal to an intermediary object. That is, if `o.equals(u)` and `u.equals(y)`, then `o.equals(y)` as well.
- **Consistent**: The truth value returned by calls to `equals()` must not change unless a property the `equals()` method relies on changes.

There is also one more criteria... If we override the `equals()` method, we must also override the `hashCode()` method and vice versa.

> Be careful when using inheritance, it can be very easy to break the symmetry of the `equals()` method by overriding it more than once. Favour composition where possible.

The `hashCode()` method is also present on all `Object` types in Java, this returns an integer representation of an instance of a class. The `hashCode()` method calculates this in such a way that it is meant to be unique, so we must also aim for this generated hash code to be as unique as possible to help prevent collisions. A good way of helping achieve this is to use prime numbers in the calculations:

```
1   class User {
2       // omitted for brevity, same as above
3
4       @Override
5       public int hashCode(Object o) {
6           int code = 7; // starting on a prime
7           if (username != null) {
8               // multiplying by a prime
9               code = 37 * code + username.hashCode();
10          }
11          return code;
```

```
12      }
13  }
```

Now, this is a correct `hashCode` method according to the Java specification, but why is the question. Well, lets consider the criteria for the `hashCode()` method:

- **Internal consistency**: The hash code produced when `hashCode` is invoked must not change unless information used in `equals()` has also changed.
- **Equals consistency**: If two objects are equal to each other, they **must** return the same hash code.
- **Collisions**: It is not a requirement that collisions must be prevented, but it is a recommendation to aim for this as distinct integer results can improve a hash tables performance.

Now let's consider how this ties into the `HashMap` and `HashSet` data types. A `HashSet` is backed by a `HashMap`, so we will focus on that. The `HashMap` type will maintain a table of entries, each entry will have references to its associated keys and values. The entries in this table are organised by their hash code to achieve a constant time complexity ($O(1)$) for operations like getting and setting an elements value in the map.

> When using a HashMap, the data type of the keys should not be a mutable data type. The `HashMap` will produce unexpected results should the `key` mutate, not finding the data assigned to a key when using it is a common side-effect.

14.8 The `Collections` class

The `Collections` class from the `java.util` package is a utility class consisting of only static methods which operate on or return collections. The algorithms use **parametric polymorphism** (generics) to work successfully with the collection classes. Here are some of the methods you may wish to use:

Method	Description
`static <T> boolean addAll(Collection<? super T> c, T... elements)`	This method takes a `Collection` of elements of type `T` followed by a variable number of arguments (also of type `T`) to add to the collection.
`static <T> List<T> emptyList()`	Returns an empty immutable list.

Continued on next page

Continued from previous page

Method	Description
`static <T> void fill(List<? super T> list, T obj)`	This method takes a list of elements of type `T`, setting every element already in that list to the object of type `T`, `obj`.
`static <T extends Object & Comparable<? super T» max(Collection<? extends T> coll)`	This method returns the maximum element of the given collection, according to its *natural ordering*. All elements in the collection must implement the `Comparable` interface, and must also be *mutually comparable* which means that `e1.compareTo(e2)` must not throw a `ClassCastException` for any elements `e1` and `e2` in the collection.
`static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp)`	As with `max(Collection<? extends T> coll)`, but uses a custom `Comparator`.
`sort(List<T> list)`	Sorts the specified list into ascending order, according to the *natural ordering* of its elements. The natural ordering is defined by the `Comparable` interface being implemented by the elements contained in the list.
`sort(List<T> list, Comparator<? super T> c)`	Sorts the list according to the order specified by the given `Comparator`.
`static <T> void copy(<List<? super T> dest, List<? extends T> src)`	Copies the elements of list `src` into the list `dest`.

There also exists a `min` equivalent to the `Collections.max` methods, which as suggested by their name return the minimum value in the list. Let's look at some of these operations using numbers:

```
1 import java.util.List;
2 import java.util.Collections;
3 import java.util.ArrayList;
4 // place imports above class declaration
5
```

```
6  List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 20, 10, 14,
       18, 18);
7  List<Integer> numbersCopy1 = new ArrayList<>(Collections.nCopies
       (numbers.size(), 0));
8  Collections.copy(numbersCopy1, numbers);
9
10 // Original order
11 System.out.print("=== ORIGINAL ORDER OF 'numbers' : [");
12 for (var num : numbers) System.out.print(num + " ");
13 System.out.print("]\n");
14
15 // Sorted order:
16 Collections.sort(numbersCopy1);
17 System.out.print("=== SORTED ORDER OF 'numbers'   : [");
18 for (var num : numbersCopy1) System.out.print(num + " ");
19 System.out.print("]\n");
20
21 // Min item
22 System.out.println("=== MIN ITEM OF 'numbers'       : " +
       Collections.min(numbers));
23
24 // Max item
25 System.out.println("=== MAX ITEM OF 'numbers'       : " +
       Collections.max(numbers));
```

We would get the following output:

```
=== ORIGINAL ORDER OF 'numbers' : [1 2 3 4 5 6 20 10 14 18 18 ]
=== SORTED ORDER OF 'numbers'   : [1 2 3 4 5 6 10 14 18 18 20 ]
=== MIN ITEM OF 'numbers'       : 1
=== MAX ITEM OF 'numbers'       : 20
```

As we can see, the `Collections.sort` static method will mutate the order of the elements contained inside a list - this will fail if the list is immutable. We can also see that the minimum and maximum items have been retrieved according to their *natural order* defined by the data types `Comparable` implementation.

14.9 What is a `Comparable`?

The `Comparable<T>` functional interface is used to define the natural ordering of some data type, the **natural ordering** being the order we would *naturally* sort some set of items into - such as ascending (smallest to largest) or descending (largest to smallest) order for numbers. We can define the natural ordering of a data type by implementating the `Comparable` interface. The interface is designed

to work closely with the built-in Java collection API's. The `Comparable` interface would look as follows:

```
1 public interface Comparable<T> {
2     int compareTo(T o);
3 }
```

The `compareTo` method is the only method defined in this interface and at a minimum should abide by the following rules:

- Returns a positive integer if the current object (the object the method is called on) is greater than the specified object `o`
- Returns `0` if the current object is equal to `o`
- Returns a negative integer if the current object is less than the specified object `o`

It is also a requirement that the `equals()` method be consistent with the implementor of this interface, it should be indicated where it is not consistent. By consistent, that means that `(x.compareTo(y) \equal\equal 0) == (x.equals(y))` should return `true`. This essentially means that when two items have the same sorting order, they are the same item when compared for equality.

Let's look at an example of implementing the `Comparable` interface where we sort `Todo` items by their ID (their natural order for this example):

```
1  import java.util.*;
2
3  class Todo implements Comparable<Todo> {
4      private int id;
5      private String content;
6
7      public Todo(int id, String content) {
8          this.id = id;
9          this.content = content;
10     }
11
12     public int getId() { return id; }
13     public String getContent() { return content; }
14
15     @Override
16     public String toString() {
17         return "TODO \n"
18                 + "  ID      : " + id + "\n"
19                + "  CONTENT : " + content;
20     }
21
```

```
22      @Override
23      public boolean equals(Object o) {
24          if (o == this) return true;
25          if (!(o instanceof Todo)) return false;
26          Todo other = (Todo) o;
27          if (other.id != id) return false;
28          if (other.content != content) return false;
29          return true;
30      }
31
32      @Override
33      public int compareTo(Todo todo) {
34          if (id == todo.getId()) return 0;
35          else if (id < todo.getId()) return -1;
36          else return 1;
37      }
38  }
39
40  public class App {
41      public static void main(String[] args) {
42          List<Todo> todos = new ArrayList(List.of(
43                                      new Todo(3, "Take the trash
                                            out"),
44                                      new Todo(2, "Take the bins
                                            out"),
45                                      new Todo(1, "Plan lessons for
                                             tomorrow")
46          ));
47          System.out.println("=== STARTING ORDER ===");
48          for (Todo todo : todos) System.out.println(todo);
49
50          // sort the todos
51          Collections.sort(todos); // this is what iterates and
                calls compareTo on each todo in the list
52          System.out.println("=== SORTED ORDER ===");
53          for (Todo todo : todos) System.out.println(todo);
54      }
55  }
```

There is a lot going on in this code. Let's take a look at the type signature for `Todo`:

```
1  public class Todo implements Comparable<Todo> { /* ... */ }
```

We implement the `Comparable` interface and specify the type that `Comparable` should use for the argument to `compareTo`, this indicates the class is also a candidate for use in Java's built-in collection utility methods for sorting. Let's look at

the implementation for `compareTo`:

```
1  public class Todo implements Comparable<Todo> {
2
3      /* code omitted for brevity */
4
5      @Override
6      public int compareTo(Todo todo) {
7          if (id == todo.getId()) return 0;
8          else if (id < todo.getId()) return -1;
9          else return 1;
10     }
11 }
```

Here, we have implemented the `compareTo` method. It returns `0` when the current todo object has the same id as the passed in `todo`. `-1` is returned if this items id is less in value than the passed in `todo`. Otherwise, this todo objects id value is greater than the passed in `todo` and `1` is returned. When we then use the `Collections.sort` method in the `main` method example code to sort a `List<Todo>`, the `Todo` items are sorted by their natural order (the order of their ids). The output for the prior program is as follows:

```
=== STARTING ORDER ===
TODO
  ID      : 3
  CONTENT : Take the trash out
TODO
  ID      : 2
  CONTENT : Take the bins out
TODO
  ID      : 1
  CONTENT : Plan lessons for tomorrow
=== SORTED ORDER ===
TODO
  ID      : 1
  CONTENT : Plan lessons for tomorrow
TODO
  ID      : 2
  CONTENT : Take the bins out
TODO
  ID      : 3
  CONTENT : Take the trash out
```

14.10 What is a `Comparator`?

The `Comparator<T>` functional interface is used, as the Oracle JDK documentation states, to *impose a total ordering on some collection of objects*. By **total ordering**, it means we can override the natural ordering of a `Comparable` implementation such as the `Todo` class in the previous section - this allows us to provide a custom sort order. `Comparator` implementations can also be used for providing sort orders for object collections which don't have a natural sort order defined. The interface looks as follows:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
    // default and static methods
}
```

The rules for the `compare` method are similar to the `Comparable`'s requirements for `compareTo`:

- If `o1` is smaller than `o2`, return a negative number
- If `o1` is equal to `o2`, return `0`
- If `o1` is larger than `o2`, return a positive number

We will use the `Todo` example from the previous section, defined as follows:

```
class Todo implements Comparable<Todo> {

    public static enum Status { TODO, IN_PROGRESS, DONE }

    private int id;
    private String content;
    private Status status;

    public Todo(int id, String content, Status status) {
        this.id = id;
        this.content = content;
        this.status = status;
    }

    public int getId() { return id; }
    public String getContent() { return content; }
    public Status getStatus() { return status; }

    @Override
    public String toString() {
        return "TODO \n"
             + "  ID       : " + id + "\n"
```

```
            + "  CONTENT  : " + content + "\n"
            + "  STATUS   : " + status;
    }

    @Override
    public boolean equals(Object o) {
        if (o == this) return true;
        if (!(o instanceof Todo)) return false;
        Todo other = (Todo) o;
        if (other.id != id) return false;
        if (other.content != content) return false;
        if (other.status != status) return false;
        return true;
    }

    @Override
    public int compareTo(Todo todo) {
        if (id == todo.getId()) return 0;
        else if (id < todo.getId()) return -1;
        else return 1;
    }
}
```

We have added a new `Status` enumerated type and a field on the `Todo` class to represent a status for each todo object. The natural ordering is still by its ID, we will now create a custom `Comparator` to sort `Todo` instances by their priority:

```
class TodoStatusComparator implements Comparator<Todo> {
    // Assumes neither Todo has a null priority
    public int compare(Todo o1, Todo o2) {
        // return a negative integer if the first arg is less
           than the second
        // return 0 if they are equal
        if (o1.getStatus() == o2.getStatus()) return 0;
        // return a positive integer if the first argument is
           greater than the second
        switch (o1.getStatus()) {
        case TODO:
            return -1; // TODO (highest priority) is smaller
               than IN_PROGRESS and DONE
            // we already know it is not equal because of the
               first check for equality
        case IN_PROGRESS:
            // o2 is smaller if it is TODO
            if (o2.getStatus() == Todo.Status.TODO) return 1; //
                o1 is larger
            // otherwise o2 must be the largest as it must be
               DONE
```

```
            return -1;
        }

        // o1 must be larger as o2 is not equal to it
        return 1;
    }
}
```

Our comparitor is checking the order of the priorities, where `TODO` is the smallest value and thus highest priority. `DONE` is the largest value and thus smallest priority. We can use our custom comparator as follows:

```
import java.util.*;
// import for Todo and TodoPriorityComparator

public class App {
    public static void main(String[] args) {
        List<Todo> todos = new ArrayList(List.of(
                                    new Todo(3, "Take the trash
                                        out", Todo.Status.
                                        IN_PROGRESS),
                                    new Todo(2, "Take the bins
                                        out", Todo.Status.DONE),
                                    new Todo(1, "Plan lessons for
                                         tomorrow", Todo.Status.
                                        TODO)
        ));
        System.out.println("=== STARTING ORDER ===");
        for (Todo todo : todos) System.out.println(todo);
        System.out.println();

        // sort the todos
        Collections.sort(todos); // this is what iterates and
            calls compareTo on each todo in the list
        System.out.println("=== NATURAL SORTED ORDER ===");
        for (Todo todo : todos) System.out.println(todo);
        System.out.println();

        // sort todos using custom comparator in ascending
            priority order
        TodoStatusComparator todoCmp = new TodoStatusComparator
            ();
        Collections.sort(todos, todoCmp);
        System.out.println("=== STATUS ASCENDING ORDER ===");
        for (Todo todo : todos) System.out.println(todo);
        System.out.println();

        Collections.sort(todos, todoCmp.reversed());
```

```
29          System.out.println("=== STATUS DESCENDING ORDER ===");
30          for (Todo todo : todos) System.out.println(todo);
31          System.out.println();
32      }
33  }
```

The output will look as follows:

```
=== STARTING ORDER ===
TODO
  ID       : 3
  CONTENT  : Take the trash out
  STATUS   : IN_PROGRESS
TODO
  ID       : 2
  CONTENT  : Take the bins out
  STATUS   : DONE
TODO
  ID       : 1
  CONTENT  : Plan lessons for tomorrow
  STATUS   : TODO

=== NATURAL SORTED ORDER ===
TODO
  ID       : 1
  CONTENT  : Plan lessons for tomorrow
  STATUS   : TODO
TODO
  ID       : 2
  CONTENT  : Take the bins out
  STATUS   : DONE
TODO
  ID       : 3
  CONTENT  : Take the trash out
  STATUS   : IN_PROGRESS

=== STATUS ASCENDING ORDER ===
TODO
  ID       : 1
  CONTENT  : Plan lessons for tomorrow
  STATUS   : TODO
```

```
TODO
  ID       : 3
  CONTENT  : Take the trash out
  STATUS   : IN_PROGRESS
TODO
  ID       : 2
  CONTENT  : Take the bins out
  STATUS   : DONE

=== STATUS DESCENDING ORDER ===
TODO
  ID       : 2
  CONTENT  : Take the bins out
  STATUS   : DONE
TODO
  ID       : 3
  CONTENT  : Take the trash out
  STATUS   : IN_PROGRESS
TODO
  ID       : 1
  CONTENT  : Plan lessons for tomorrow
  STATUS   : TODO
```

As we can see, we can change the sort order from that defined by the `Comparable` interface which the `Todo` class implements to a custom order by defining a `Comparator`. As `Comparator` is a functional interface, it could also be defined using a lambda expression.

15 String handling and manipulation

The `String` class documentation can be found here.

In this section, we aim to cover:

15.1 String literals

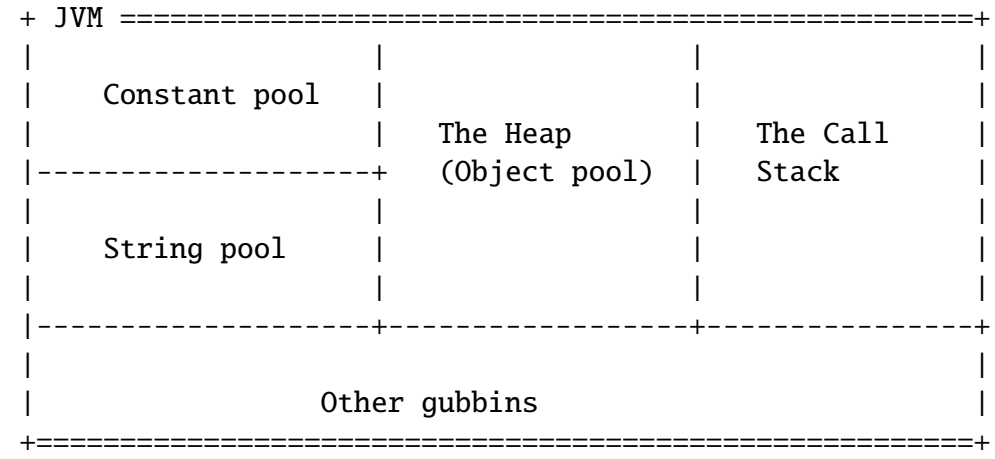
A **string literal** is that which we write in source code between double quotes, for example:

```
1 String name = "Bob";
```

The literal is `"Bob"`, these string literals are stored in the string pool for efficiency and performance reasons.

15.2 The String Pool

The **string pool** is an area of memory within the JVM that is optimised for storing textual data:



As we can see, the JVM is made up of many different components. The string pool is under discussion here, which exists thanks to the fact that strings are immutable. This allows for the JVM to optimise their storage by only storing one copy of any given string, and then using that string for future references. By default, all string literals are **interned** into the string pool - *interning* is the process of putting a copy of a string into the string pool. We can demonstrate this with some simple code:

```
1 String greeting1 = "Hello";
2 String greeting2 = "Hello";
3
4 if (greeting1 == greeting2) {
5     System.out.println("greeting1 and greeting2 both refer to
          the same\n object in the string pool with the value of '"
           + greeting1 + "'.");
```

```
}
```

This will produce the following result confirming that these two variables reference the same object in the string pool:

```
greeting1 and greeting2 both refer to the same
 object in the string pool with the value of 'Hello'.
```

We know this to be the case as the == operator applied to two objects compares their identities (references). If we create a string using the `String` constructor, the Java compiler will not store this string in the string pool by default - it will instead be stored in the heap with the rest of the objects. We can confirm this in the same way as before:

```
String greeting1 = "Hello";
String greeting2 = new String("Hello");

if (greeting1 != greeting2) {
    System.out.println("greeting1 and greeting2 do not refer to
        the\n same object in the string pool with the value of '"
         + greeting1 + "'.");
}
```

And here are the results:

```
greeting1 and greeting2 do not refer to the
 same object in the string pool with the value of 'Hello'.
```

The not-equal-to operator (`!=`) also compares two objects by their identities, just like the equality operator ==.

You might now be wondering, how can we put a string in the string pool then if it is created using a constructor call. Java provides a method called `intern()` on every string which can be called to tell the JVM to store its reference in the pool. Here is an example of this:

```
String greeting1 = "Hello";
String greeting2 = new String("Hello");

if (greeting1 != greeting2) {
    System.out.println("greeting1 and greeting2 do not refer to
        the same\n object in the string pool with the value of '"
         + greeting1 + "'.");
}

```

```
8  greeting2 = greeting2.intern();
9  if (greeting1 == greeting2) {
10     System.out.println("greeting1 and greeting2 both refer to
           the same\n object in the string pool with the value of '"
            + greeting1 + "' after interning.");
11 }
```

The `intern()` method returns a reference which points to the strings location in the string pool, the previous program will output:

```
greeting1 and greeting2 do not refer to the same
 object in the string pool with the value of 'Hello'.
greeting1 and greeting2 both refer to the same
 object in the string pool with the value of 'Hello' after interning.
```

15.3 Basic string manipulation

In Java, the `String` data type is immutable - this means the data stored inside it, the characters, cannot be modified. This also means that any operations or methods which attempt to manipulate a string will produce a new string object. The following example demonstrates this immutability:

```
1  String greeting = "Hello";
2  String name = "Bob";
3
4  String completeGreeting = greeting + " " + name;
5
6  if (greeting == completeGreeting) {
7      System.out.println("'greeting' was modified");
8  } else {
9      System.out.println("'greeting' was not modified, strings are
           of course immutable.");
10 }
```

This will produce the following output:

```
'greeting' was not modified, strings are of course immutable.
```

There also exists two methods on the `String` data type for retrieving a new string in upper or lower case:

```
1  String commandInput = "create table";
2  System.out.println("> " + commandInput);
3  commandInput = commandInput.toUpperCase();
```

```
4
5 String response = "Error: failed, missing name in command '" +
      commandInput + "'";
6 response = response.toUpperCase();
7
8 System.out.println(response);
```

This program is simulating some kind of command interpreter. The key detail is that when `toUpperCase()` is called, it returns a new string object which is then assigned to the given variables. The output is as follows:

```
> create table
ERROR: FAILED, MISSING NAME IN COMMAND 'CREATE TABLE'
```

There also exists a `trim()` method for removing leading and trailing whitespace from a string, this also returns a new string object with the original unmodified:

```
1 String input = "   Hi there!  ";
2 System.out.println("INPUT: \"" + input + "\"");
3 System.out.println("TRIMMED INPUT: \"" + input.trim() + "\"");
```

As we can see, the `trim()` method returns a new string which has removed any surrounding whitespace:

```
INPUT: "   Hi there!  "
TRIMMED INPUT: "Hi there!"
```

15.4 Substrings, splitting and retrieving characters

It is often the case when parsing textual data that we want to select specific sections of a string or iterate over each character. We will first demonstrate the use of `charAt()` in combination with `length()` to iterate over and retrieve each char in the string:

```
1 String name = "Bob";
2 for (int i = 0; i < name.length(); i++) {
3     char current = name.charAt(i);
4     System.out.print(current);
5 }
```

This example just prints the value of the string, but one character at a time. Dreadfully inefficient and pointless, but it does demonstrate how `charAt()` works. We would get the following output to the console:

```
Bob
```

We will now look at substrings, which are strings that represent part of another string. For example, `"lo"` is a substring of `"hello"` from index 3 onwards:

```
1 // substring(startIndex)
2 String sub = "hello".substring(3);
3 System.out.println("'hello' substring(3) = " + sub);
4
5 // substring(starrIndex, endIndex)
6 System.out.println("'hello' substring(1, 3) = " + "hello".
    substring(1, 3));
```

We have demonstrated both available `substring()` methods, one which retrieves the letters from the specified index onwards and another which substrings between a range (the end index of this range is exclusive, it does not include that character at that position):

```
'hello' substring(3) = lo
'hello' substring(1, 3) = el
```

We also have the `split()` method which accepts a regular expression as a string and splits a string around the match:

```
1 String source = "( + 3 3 )";
2 String[] tokens = source.split("\s");
3
4 for (String token : tokens) {
5     System.out.println("TOKEN: " + token);
6 }
```

```
TOKEN: (
TOKEN: +
TOKEN: 3
TOKEN: 3
TOKEN: )
```

Here, we have a split a string using a whitespace character as the regular expression. You can limit the amount of times that the split occurs by calling the overloaded version `split(String regex, int limit)`.

15.5 String comparisons

When comparing strings, we can generally compare them in one of three ways:

1. Using the `equals()` method inherited from the `Object` supertype.
2. Using the equality operator (==) to compare identity.
3. Using the `compareTo()` method which internally uses the `Comparable<T>` interface where `T` is `String`.

There are also additional methods available for comparing the contents of a string:

Method	Description
`int compareTo(String anotherString)`	Compares two strings using their lexicographic order (alphabetic order).
`int compareToIgnoreCase(String str)`	As with `compareTo`, but ignores case.
`boolean endsWith(String suffix)`	Returns `true` if the string this method is called on ends with the specified `suffix`, otherwise returns `false`.
`boolean equals(Object anObject)`	Compares a string to the given object using an internal, custom implementation rather than identity like is default. Returns `true` if equal, otherwise `false`.
`boolean equalsIgnoreCase(String anotherString)`	As with `equals`, but only accepts a string as an argument and ignores case.
`int indexOf(int ch)`	Returns the index of the first occurrence of a given character in a string, otherwise returns `-1`. There is also an overload which uses a string.
`int indexOf(int ch, int fromIndex)`	As with `indexOf`, but starts searching from the specified index (`fromIndex`). There is also an overloaded version which uses a string.
`int lastIndexOf(int ch)`	Returns the index of the last occurrence of the specified character, otherwise returns `-1`. There is also an overload which uses a string.

Continued on next page

Continued from previous page

Method	Description
`boolean matches(String regex)`	Returns `true` if the string matches the given regular expression, otherwise `false`.
`boolean startsWith(String prefix)`	Returns `true` if the string begins with the specified `prefix`, otherwise false.
`boolean startsWIth(String prefix, int offset)`	Returns `true` if the string, from the given `offset` into the string, begins with the specified `prefix`, otherwise returns `false`.
`boolean contains(CharSequence s)`	Returns `true` if and only if this string contains the specified character sequence of values, otherwise `false`.

Here is an example of using a variety of the string comparison methods:

```
1  String name = "Dave";
2  String greeting = "Hello there " + name;
3
4  System.out.println("greeting                              : " +
       greeting);
5  System.out.println("name                                  : " +
       name);
6
7  System.out.println("greeting contains(name)               : " +
       greeting.contains(name));
8  System.out.println("greeting startsWith(\"Hello\")          : " +
       greeting.startsWith("Hello"));
9  System.out.println("greeting indexOf('l')                 : " +
       greeting.indexOf('l'));
10 System.out.println("greeting lastIndexOf('l')             : " +
       greeting.lastIndexOf('l'));
11 System.out.println("greeting compareTo(\"Hello there \") : " +
       greeting.compareTo("Hello there "));
12 System.out.println("greeting compareTo(greeting)          : " +
       greeting.compareTo(greeting));
13 System.out.println("greeting compareTo(greeting + \"!\") : " +
       greeting.compareTo(greeting + "!"));
```

Following would be the output of the above program:

```
greeting                              : Hello there Dave
name                                  : Dave
```

```
greeting contains(name)             : true
greeting startsWith("Hello")        : true
greeting indexOf('l')               : 2
greeting lastIndexOf('l')           : 3
greeting compareTo("Hello there ") : 4
greeting compareTo(greeting)        : 0
greeting compareTo(greeting + "!") : -1
```

The `compareTo` method is of interest here, when called, it will return one of the following values:

- `-1` if the string is lexiographically less (has less characters) than the given string
- `0` if the strings are equal
- A value greater than `0` if the string the method is called on is lexicographically larger than the string it is being compared to. The value returned is equivalent to `str1.length() - str2.length()` for the operation `str1.compareTo(str2)` where `str1` has a greater length than `str2`.

15.6 Regular expressions

Put simply, a **regular expression** is a pattern used against a string to look for specific sequences. The term **regular** comes from where regular expressions lie in the *Chomsky Hierarchy*, regular expressions represent a type 3 (regular) language in this hierarchy. The **Chomsky Hierarchy** is a way of classifying the formal grammars of a language - a languages syntax rules and thus their capabilities.

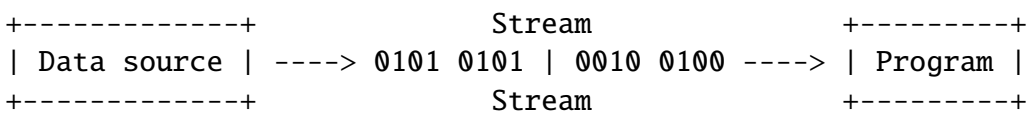
Regular expressions are beyond the scope of this document. Oracle have documentation summarising the constructs of regular expressions here, constructs meaning the different patterns we can apply at its simplest. You may also want to use an online, in-browser, regular expression playground such as Regex101 or RegExr.

16 File handling

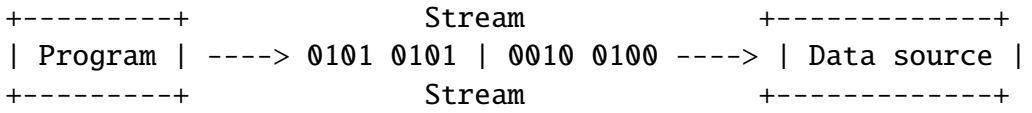
16.1 What is an I/O stream?

An **I/O Stream** is a general term for a variety of different input sources and output destinations, such as files, devices, etc... Streams can be used with many different data types in Java, from bytes to other primitives up-to and including objects. Streams are generally used for reading data and manipulating data to transform it into desired formats.

A stream at its core is a sequence of data, an **input stream** is used to read data from a source (one piece of data at a time):

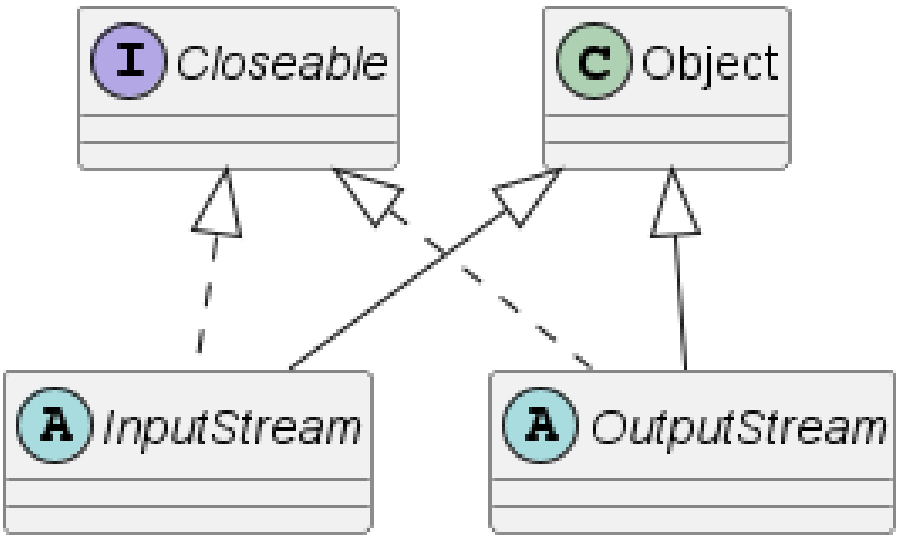


An **output stream** writes data to a destination:

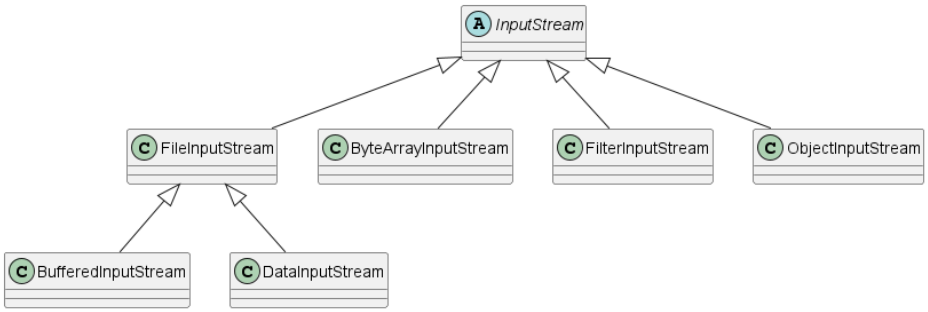


16.2 The stream data type hierarchy

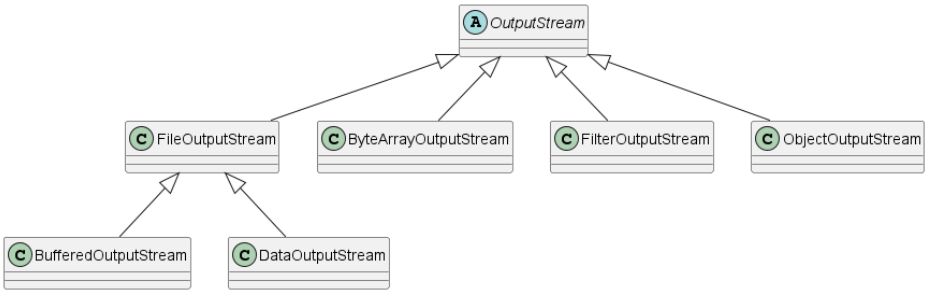
The input and output streams are not represented by one unified supertype, but instead by their own types. We have the InputStream and OutputStream abstract classes available, which create their own hierarchies. We can visualise the start of this hierarchy as follows:



The `InputStream` and `OutputStream` classes are abstract, representing the behaviours that a stream should implement. The `InputStream` hierarchy look as follows:



The `OutputStream` hierarchy is as follows:



All we will explore is how to read from and write to a file, but there are a vari-

ety of uses for the IO stream implementations; it is advisable that you do some research here. The Basic I/O Java 8 Oracle tutorials are a good place to start - the tutorial is rather plain, so it could be advisable to pair it along with some other kinds of interactive tutorial. From there, it would be good to update yourself with the changes made between JDK 8 and JDK 11 which saw the introduction of an improved file API.

16.3 Reading from a file

Reading from a file requires the use of an `InputStream`, luckily Java provides a few different implementations. A `BufferedInputStream` has been the goto choice from Java 1 through 8 as it allows us to read in the data as required, line by line or character by character without flooding memory. Here is an example of reading a files contents from the root of the classpath:

```
String filepath = "./resources/test.txt"; // ./ is from root of
    classpath
StringBuilder results = new StringBuilder();

try (BufferedReader br = new BufferedReader(new FileReader(
    filepath))) {
    // body of try
    String line;
    while ((line = br.readLine()) != null) {
        results.append(line).append('\n');
    }
} catch (IOException e) {
    e.printStackTrace(); // something went wrong closing the
        stream
}
System.out.println(results.toString());
```

We can also use a `Scanner` object to read from a file, the benefit being that we can specify a custom delimiter token:

```
String filepath = "./resources/test.txt";
StringBuilder results = new StringBuilder();

try (Scanner sc = new Scanner(new File(filepath))) {
    sc.useDelimiter("\n");
    String line;

    while ((line = scanner.next()) != null) {
        results.append(line);
    }
```

```
} catch (IOException e) {
    e.printStackTrace();
}
```

16.4 Writing to a file

Writing to a file is as simple as reading from a file, we are just using the `OutputStream` implementations instead. Here is an example using the `BufferedWriter` to create a new file and write a string to it:

```
String filepath = "./resources/newfile.txt"; // ./ is from root of classpath

try (BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(filepath))) {
    bufferedWriter.write("Hello world\n");

} catch (IOException e) {
    e.printStackTrace(); // something went wrong closing the stream
}
```

If we want to append to an existing file, we should pass a boolean indicating this to the `FileWriter` constructor:

```
String filepath = "./resources/newfile.txt"; // ./ is from root of classpath

try (BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(filepath, true))) {
    bufferedWriter.write("Hello world\n");

} catch (IOException e) {
    e.printStackTrace(); // something went wrong closing the stream
}
```

17 Functional programming

17.1 What is functional programming?

Wikipedia defines **functional programming** as follows:

> In computer science, functional programming is a programming paradigm where programs are constructed by applying and composing functions.
>
> (Wikipedia - Functional Programming, 2023)

The key part here is that it is a **programming paradigm**, a declarative one in-particular, and is represented by the application and composition of functions for computation. Functional programming aims to be free from *side-effects*, a **side-effect** being anything which modifies state; this also means that functional programming favours immutability. The Wikipedia page on functional programming goes on to say:

> In functional programming, functions are treated as first-class citizens, meaning that they can be bound to names (including local identifiers), passed as arguments, and returned from other functions, just as any other data type can. This allows programs to be written in a declarative and composable style, where small functions are combined in a modular manner.

What this reveals is that we treat functions as values that can also map a value to another value - a **value** being some first-class citizen in a language in this context. To use a simpler terminology, we can assign functions as values of a variable (bound to names), we can pass functions as arguments to another function and return a function from a function.

In the following example, we demonstrate a few different features of the functional programming paradigm using Java 8+ - the functional paradigm in Java is layered on top of the object system:

```
1  import java.util.function.Supplier;
2  import java.util.function.Consumer;
3
4  public class App {
5      public static void main(String[] args) {
6          var counter = getCounter();
7          var counter2 = getCounter();
8          Consumer<Object> println = o -> System.out.println(o);
9
10         println.accept("COUNTER 1");
```

```
11          println.accept(counter.get());
12          println.accept(counter.get());
13
14          println.accept("COUNTER 2");
15          println.accept(counter2.get());
16          println.accept(counter2.get());
17          println.accept(counter2.get());
18      }
19
20      /**
21         When called, this function returns a supplier function
               which can be used as a
22         counter. An array of 1 item is used to enable mutability
               as lambda expressions can
23         only use a local variable if it is effectively final or
               final.
24       */
25      public static Supplier<Integer> getCounter() {
26          final int[] counter = {0};
27          return () -> counter[0]++;
28      }
29  }
```

This example demonstrates a few key features of functional programming, within the boundaries that Java imposes - the type system implemented in Java, and the way that lambda expressions are built on top of it, limits how we can use the functional programming paradigm. Anyway, on line 8 we create a lambda expression and assign it to a variable; that is to say, we have treated the lambda as if it is a value. The `getCounter()` method demonstrates two concepts from functional programming, higher-order functions and closures. A higher-order function is any function that accepts a function as an argument to a parameter or returns a function, `getCounter()` returns a lambda expression (Java's representation of a function expression). The closure part refers to the fact that the returned function can hold onto its environment, meaning it can still reference the `counter` variable defined in `getCounter()` even though we called `counter.get()` in a different scope. Anyway, here are the results of executing the previous block of code:

```
COUNTER 1
0
1
COUNTER 2
0
1
2
```

17.2 The functional interface

The functional interface was introduced in Java 8 as a way to allow lambda expressions to be used in the language as first class members, there was no such thing as a lambda in Java before version 8. Functional interfaces replace the need to create a variety of small classes which each encapsulate a single behaviour, this will be demonstrated shortly with an example.

Before we apply the concept of a functional interface, we must understand the rules behind it. A functional interface is an `interface` type declaration which abides by the **Single Abstract Method** (**SAM**) rule, where the interface is only a functional interface if it has a single abstract method:

```
1 @FunctionalInterface
2 public interface Command {
3     Object execute(String[] args);
4 }
```

This is an example of a functional interface, the `@FunctionalInterface` annotation is optional - its usage is recommended to aid the compiler with recognising errors related to the SAM rule and for readability. You can have `default` methods inside a functional interface as these are not abstract, a useful little quirk.

Now, let's consider a use for this interface. Prior to Java 8, we would have to create a new class for every command:

```
1  public class PrintCommand implements Command {
2      @Override
3      public Object execute(String[] args) {
4          for (String arg : args) System.out.print(arg);
5          return null;
6      }
7  }
8
9  public class AddCommand implements Command {
10     @Override
11     public Object execute(String[] args) {
12         double result = 0;
13         try {
14             for (String arg : args) {
15                 result += Double.valueOf(arg);
16             }
17         } catch (Exception e) {
18             throw new RuntimeException("Oops, something went
                   wrong... Argument needs to be a number");
19         }
```

```
20          return result;
21      }
22  }
```

Using lambda expressions, we can define these commands inside a method as variables:

```
1  Command printCommand = (args) -> {
2      for (String arg : args) System.out.print(arg);
3      return null;
4  };
5
6  Command addCommand = (args) -> {
7      double result = 0;
8      try {
9          for (String arg : args) {
10             result += Double.valueOf(arg);
11         }
12     } catch (Exception e) {
13         throw new RuntimeException("Oops, something went wrong
               ... Argument needs to be a number");
14     }
15     return result;
16 };
```

This can reduce the amount of required boilerplate code when dealing with encapsulated behaviours which may only be used in a few places in our program.

17.3 The built-in functional interfaces

Java provides a package of built-in functional interfaces (`java.util.function`) for the most common function types used, these interfaces are used thoroughly throughout the JDK for collection classes and streams. Here is a table listing some of them:

Interface	Description
`Function<T,R>`	Represents a function which accepts an argument of type `T` and returns a value of type `R`.
`Consumer<T>`	Represents a function which accepts a single argument of type `T` and does not return a value.
`Supplier<T>`	Represents a function which accepts no arguments and returns a value of type `T`.

Continued on next page

Continued from previous page

Interface	Description
`Predicate<T>`	Represents a predicate function which accepts an argument of type `T` and returns a `boolean`.
`BiFunction<T,U,R>`	Represents a function which accepts two arguments of types `T` and `R`, returning a value of type `R`

You can view a whole list of the functional interfaces introduced in Java 8 using the documentation.

17.4 The syntax of a Lambda expression

In Java, lambda expressions follow a specific syntax. To declare a lambda expression, we specify a parameter list followed by an arrow and then the code to execute when the lambda is invoked:

```
(param1, param2, ...) -> code;
```

> When declaring parameters, we don't need to specify the data types as these are inferred from the interfaces single abstract method which represents the lambda.

To declare a lambda expression which takes no arguments, we specify an empty parameter list:

```
() -> System.out.println("Hello World");
```

A lambda expression which accepts a single argument can optionally omit the parameters:

```
(name) -> System.out.println("Hello " + name);
// is equivalent to
name -> System.out.println("Hello " + name);
```

A lambda expression which does not have a block body can return a value without use of the `return` keyword, it is in fact illegal to use the `return` keyword in this way:

```
BiFunction<Integer,Integer,Integer> add = (num1, num2) -> num1 +
    num2;
int result = add.apply(10, 20);
```

As a lambda expression actually expands to an anonymous class in the background, we don't call it like a method but instead call a method on the anonymous class it expanded to. That `BiFunction` called `add` that we defined will be expanded by the compiler to something like:

```
BiFunction<Integer,Integer,Integer> add = new BiFunction<>() {
        @Override
        public Integer apply(Integer num1, Integer num2) {
            return num1 + num2;
        }
}
```

That is how Java is able to implement Lambda expressions without breaking backwards compatibility, by compiling them down to anonymous classes. Anyway, if we have a block of code as the body of a lambda, we must use the `return` keyword to return a value:

```
Function<String, Integer> primeTheHashSomeMore = (str) -> {
    int hashCode = str.hashCode();
    return hashCode / 31 * 17;
};
int primedHash = primeTheHashSomeMore.apply("Hello world");
```

17.5 Applying Lambda expressions to collections

Lambda expressions, from Java 8 onwards, can be applied as arguments to many different methods defined on the data stream types. These lambda expressions could be used for something like sorting, a comparator implementation that is, or even for things like filtering out numbers in a list or aggregating data.

The following example demonstrates how we might apply lambda expressions to a list of data to filter or manipulate its contents into something new:

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

List<Integer> evenNumbers = numbers.stream()
                .filter(num -> num % 2 == 0)
                .collect(Collectors.toList());
evenNumbers.forEach(num -> System.out.println(num));

List<Integer> squaredNumbers = numbers.stream()
                .map(num -> num * num)
                .collect(Collectors.toList());
squaredNumbers.forEach(num -> System.out.println(num));
```

Let's take a closer look at the filtering operation:

```
List<Integer> evenNumbers = numbers.stream()
                .filter(num -> num % 2 == 0)
                .collect(Collectors.toList());
```

We first transform out list into a stream using the `stream()` instance method. We then call the *non-terminal* `filter(Predicate<? super T> p)` which accepts a predicate which operates on values of type `T`, the data type of the values in the stream. We then collect the filtered numbers, even in this case as that is what we are checking for, to a new `List` instance. Next, each element in the collection is printed to the console using the `forEach(Consumer<? super T> cons)` instance method defined on the `List` interface.

The lambda expression, `num -> num % 2 == 0`, is the same as:

```
class EvenNumberPredicate implements Predicate<Integer> {
    public boolean test(Integer num) {
        return num % 2 == 0;
    }
}
```

The lambda expression means we don't have to create this class, whose usage would be as follows:

```
List<Integer> evenNumbers = numbers.stream()
    .filter(new EvenNumberPredicate())
    .collect(Collectors.toList());
```

We can also do the same for the lambda expression which implements the `Consumer` interface, the one used by the `List.forEach()` instance method:

```
class PrintConsumer implements Consumer<Object> {
    public void accept(Object O) {
        System.out.println(o);
    }
}
```

17.6 Higher-order functions

A **higher-order function** is a function which either returns a function as output or accepts a function as input. The `forEach` instance method defined on the `List` interface is a higher-order function equivalent in Java, it is a method which

can take a lambda expression as input. The lambda expression, although it does compile to an anonymous class instance, is treated as a first-class member of the Java language. We can also define our own higher-order functions, such as our own `filter` implementation:

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1,2,3,4);
        var evenNumbers = filter(numbers, num -> num % 2 == 0);
        evenNumbers.forEach(num -> System.out.println(num));
    }

    public static List<Integer> filter(List<Integer> numbers,
        Predicate<Integer> filterFn) {
        List<Integer> results = new ArrayList<>();
        for (var num : numbers) {
            if (filterFn.test(num)) results.add(num);
        }
        return results;
    }
}
```

Our `filter()` method accepts a list of integers and a predicate lambda expression to use as the test on each number.

17.7 Closures

A **closure** is a special kind of behaviour present in functional programming where a function *keeps hold of* the environment it was defined in. Let's take our counter example from the start of this section:

```
public static Supplier<Integer> getCounter() {
    final int[] counter = {0};
    return () -> counter[0]++;
}
```

Each method has its own environment, the environment holds onto what variables are available inside the method. The method will also have a parent environment, which is the class it is defined in, which may also have its own parent environment, and so on.... This *holding onto* the environment allows for a function to be used

incrementally in code. Let's put the `getCounter()` method in a different class to help demonstrate the environments idea:

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

public class App {
    public static void main(String[] args) {
        Utils utils = new Utils();
        Utils utils2 = new Utils();
        var counter = utils.getCounter();
        var counter2 = utils2.getCounter();

        System.out.println("COUNTER 1: " + counter.get() + "\n");
        System.out.println("COUNTER 1: " + counter.get() + "\n");
        System.out.println("COUNTER 2: " + counter2.get() + "\n");
        System.out.println("COUNTER 2: " + counter2.get() + "\n");
    }
}

class Utils {
    private static int INTERNAL_ID = 1;
    private String message;

    public Utils() {
        message = "+= UTILS ENVIRONMENT " + INTERNAL_ID++ + " =+";
    }

    public Supplier<Integer> getCounter() {
        String methodMessage = "+= getCounter() ENVIRONMENT =+";

        final int[] counter = {0};
        return () -> {
            System.out.println("ACCESSED PARENT ENV OF getCounter() " + message);
            System.out.println("ACCESSED " + methodMessage);
            return counter[0]++;
        };
    }
}
```

The above code did not need to make two instances of `Utils` to get new counters, we could have kept the same ones. The key idea this shows is that each instance

has its own environment (numbered in the output), which indicates what is still in scope or not for a given method when called - in this case, what is still in scope for each lambda expression returned by `getCounter()`:

```
ACCESSED PARENT ENV OF getCounter() += UTILS ENVIRONMENT 1 =+
ACCESSED                            += getCounter() ENVIRONMENT =+
COUNTER 1: 0

ACCESSED PARENT ENV OF getCounter() += UTILS ENVIRONMENT 1 =+
ACCESSED                            += getCounter() ENVIRONMENT =+
COUNTER 1: 1

ACCESSED PARENT ENV OF getCounter() += UTILS ENVIRONMENT 2 =+
ACCESSED                            += getCounter() ENVIRONMENT =+
COUNTER 2: 0

ACCESSED PARENT ENV OF getCounter() += UTILS ENVIRONMENT 2 =+
ACCESSED                            += getCounter() ENVIRONMENT =+
COUNTER 2: 1
```

You might have also noticed that `getCounter()` uses an array instead of just an `int` variable. This is because Java's implementation of functional programming is limited to keep backwards compatibility with older programs. The rule when using a lambda expression is that any local variables used from outside of its scope must be final or effectively final, that is they must be a constant value and never changing. To get around this, we use an array of one item. The array itself is just a reference, we are not modifying that reference though; we are now just modifying the number stored inside the array to get around the limitation. Thus, allowing for a near-normal closure implementation.

17.8 Optionals

The `Optional` data type in Java is a *value-based class* used to represent whether a value is present or not while avoiding the infamous null pointer exception (documentation).

Lets start with a simple example of a **null-check**:

```
1 Vehicle v = garage.getVehicle(id);
2
3 if (v != null) {
```

```
    // do something
} else {
    // do something else
}
```

Null-checks can easily pollute code bases and cause hard to find errors, a simple way to work around this is to use the optional type which holds the state to represent whether a value is present or not (`null`).

To use the Optional data type, provide the following import in JShell or at the top of your of your class file and below the package statement:

```
import java.util.Optional;
```

Once imported, an optional can be declared and created using one of three static factory methods (there are no public constructors for the `Optional` class):

```
// Empty Optional that can store a single String
Optional<String> opt1 = Optional.empty();

// Create an Optional using a non-null object
String greeting = "Hello World";
Optional<String> opt2 = Optional.of(greeting);

// Create an Optional using a possibly null object
String neverInitialised; // null
var opt3 = Optional.ofNullable(neverInitialised);
// opt3 is of type Optional<String>
```

An optional acts like a container that wraps around some value, which may or may not be null. Optionals are **immutable**, we cannot change what object reference they store once an instance is created; if the object stored is a mutable type, we can still modify the data on that object though.

Once an instance of optional is created, we can call the instance method `.get()` to retrieve the value:

```
jshell> String greeting = "Hello World";
   ...> Optional<String> opt2 = Optional.of(greeting);
greeting ==> "Hello World"
opt2 ==> Optional[Hello World]

jshell> String copy = opt2.get();
copy ==> "Hello World"
```

Should we try to retrieve a value from a `null` optional, an optional storing the value of `null`, the `get()` method will throw a `NoSuchElementException`:

```
jshell> String neverInitialised;
   ...> var opt3 = Optional.ofNullable(neverInitialised);
neverInitialised ==> null
opt3 ==> Optional.empty

jshell> String copy = opt3.get();
|  Exception java.util.NoSuchElementException: No value present
|        at Optional.get (Optional.java:143)
|        at (#35:1)
```

What we can do instead is a simple `if` statement like before, but now we have no risk of a null pointer exception:

```
Optional<String> name = Optional.empty();
if (name.isPresent()) {
    // do this
} else {
    // do this
}
```

The `isPresent()` instance method returns `true` if there is an object inside the optional and `false` if it is empty/null. We can also use the `isEmpty()` instance method for the inverse. By default, the `Optional` type does not support being initialised with `null`:

```
Optional<String> name = Optional.of(null); // throws
   NullPointerException
```

So we can instead use the static `ofNullable` factory method which returns an empty optional:

```
Optional<String> name = Optional.ofNullable(null);
```

Here is where we can now explore the power of optionals in terms of functional programming. Let's consider the null-check if statement from before. What we can do instead is pass a lambda to the `ifPresent()` overloaded method to execute some block of code, for example:

```
Optional<String> name = Optional.of("Bob");
name.ifPresent(value -> {
        System.out.println(value);
});
```

Now, only if a value is present will the name `Bob` be printed to the console. The optional helps protect us from forgetting a null check and and can help to make the code more readable.

18 Maven basics

Apache Maven is an open-source build tool designed to make a programmers life easier and is the de-facto build system of choice for Java developers. A **build tool** is some method of automating the building of executable files, whether native or for a VM; build tools usually include plugin systems and ways to easily load external dependencies. The What is Maven? page holds a good description describing the Maven project from a high-level.

18.1 Installing Maven

For the installation, you should do the following:

1. Ensure Java is installed as Maven requires Java to work, it is also good practice to add the root directory of your chosen JDK to the system or users environment as a variable called `JAVA_HOME`.
2. Download the distribution for Maven from the downloads page.
3. Add Maven to your path on Unix-like environments (Linux and macOS) as specified on the install page.

18.2 Structure of a standard Maven project

A typical Maven project has the following structure:

```
app
|-- pom.xml
|-- src
    |-- main
    |   |-- java
    |   |-- resources
    |-- test
        |-- java
        |-- resources
```

The `pom.xml` file holds the project's configuration, the `src` folder contains both the main application code and the test code. The `resources` folders may be used to specify required files that do not form part of the main program or test codes Java source. The `java` folders are where your Java source code and test code live, in their respective folder hierarchies that is.

18.3 The POM file

The `pom.xml` file, called the **POM** file where POM is short for *Page Object Model*, is a configuration file which usually contains the majority of the required information to customise the build of a project to how you desire. When Maven is executing a goal or task, it looks for the POM in the current directory and uses it to get the required configuration info. The POM file uses the XML language for specifying configurations, which is syntactically similar to HTML. We have the concept of tags, of which each tag opener will have a corresponding tag closer:

```
<tag1>
  <tag2>value of tag2</tag2>
  <tag3>
    <tag4>11</tag4>
    <tag5>11</tag5>
  </tag3>
</tag1>
```

The above example demonstrates an example of using well-formed tags, the `<tag1>` is the container of all other tags in this example, it's end is marked by the tag `</tag1>`. The forward slash before a tag name signifies that the tag is a tag closer. `<tag2>` is the first child of `<tag1>` and has a string value, `<tag3>` is the second child of `<tag1>` and is also a container for its own child nodes.

All POM files inherit from Maven's default POM, the **Super POM**, unless you explicitly set it not to. You can view a copy of the default configuration for Maven 2.6.3 at its Super POM page, which should give you an idea of the general layout. To create our own POM file, there is a minimum configuration we must meet:

- Specify a `project` root element
- Specify a `modelVersion`, which should be at least 4.0.0
- The `groupId` and `artifactId` elements represent the project's group and artifact (project) respectively
- A `version` element

A minimal example might look like:

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.qa.example</groupId>
  <artifactId>my_project</artifactId>
  <version>1.0.0</version>
</project>
```

The `groupId`, `artifactId` and `version` together form the project's **fully qualified artifact name**, which takes the form:

`<groupId>:<artifactId>:<version>`

`com.qa.example:my_project:1.0.0`

18.4 Adding dependencies

Maven allows for dependencies to be included in your project when they are compiled, these are downloaded from a **repository** of *artifacts*. The default repository is specified in the Super POM, version 3.6.3 specifies `https://repo.maven.apache.org/maven2` as the default repository. You can search for dependencies using Maven Repository.

To add a dependency, you will need to add it inside a `<dependencies>` element which is also inside of the `<project>` element:

```
<project>
  <!-- omitted for brevity -->

  <dependencies>
    <!-- https://mvnrepository.com/artifact/com.googlecode.
       lanterna/lanterna -->
    <dependency>
      <groupId>com.googlecode.lanterna</groupId>
      <artifactId>lanterna</artifactId>
      <version>3.1.1</version>
    </dependency>
  </dependencies>
</project>
```

If we want additional dependencies, we should specify them between the `<dependencies>` tags as well.

18.5 Packaging an application as a distributable uber .jar file

An **uber jar** is a jar file which also contains any dependencies the project uses.

To package an application using Maven, we need to add the Maven Assembly Program to the `<build>` section of the POM file:

```
<build>
    <finalName>app</finalName>
    <plugins>
        <plugin>
            <artifactId>maven-assembly-plugin</artifactId>
            <configuration>
                <archive>
                    <manifest>
```

```
                        <mainClass>com.qa.example.App</mainClass
                            >
                    </manifest>
                </archive>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</
                        descriptorRef>
                </descriptorRefs>
            </configuration>
            <executions>
                <execution>
                    <id>make-assembly</id>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

The Maven Assembly Plugin is used to create an uber jar. To use the build-plugin, make sure to set the `<mainClass>` element to point at your entry point file as this will be used as the `Main-Class` property in the JAR file's *MANIFEST.MF* file. To use the plugin, building both a normal jar (without dependencies) and the uber jar, run `mvn clean package`. If you only want the uber jar, run `mvn clean compile assembly:single`.

19 TODO Testing your code

19.1 TODO Test-Driven Development (TDD)

19.2 TODO Unit Testing with JUnit 5

Part III

Additional content

20 Compiling and executing Java Applications

In this section, we aim to cover:

20.1 Single-File Source-Code Programs

A **Single-File Source-Code** program is one which includes a single file, with a single `Main` method. This program can then be executed directly without explicitly compiling it. More information is discussed in JEP 330, the proposal which saw its inclusion from JDK 11+.

Let's see an example, we could have a Hello World program as follows:

```
1 public class App {
2     public static void main(String[] args) {
3         System.out.println("Hello World");
4     }
5 }
```

We can the directly execute the program on any platform with a JRE and/or JDK available using:

```
1 java App.java
```

This is equivalent to running:

```
1 javac HelloWorld.java
2 java HelloWorld
```

We **must** be in the same directory, in the terminal application (CMD, Bash, etc...), as the projects `App.java` file for this example to work.

20.2 Compiling multiple files in the same directory

All examples seen in this document of compiling Java code have been for a single file, let's now start looking at how we handle multiple files...

For this example, we will have no packages - our code will reside in the *default package*. We will have two classes, `App` and `Calculator`. The `Calculator` class implements some basic mathematical operations as follows:

```
1 public class Calculator {
2     public <T extends Number> double add(T num1, T num2) {
3         return num1.doubleValue() + num2.doubleValue();
4     }
5
6     public <T extends Number> double minus(T num1, T num2) {
7         return num1.doubleValue() - num2.doubleValue();
```

```
    }
}
```

The `App` class will depend on the `Calculator` class, and also be the entrypoint to our application:

```
public class App {
    public static void main(String[] args) {
        Calculator c = new Calculator();
        System.out.println("int(5) + int(5) = " + c.add(5, 5));
    }
}
```

As all of the files in the program in one directory, we don't need to specify the location of the `Calculator` class when compiling. It could be as simple as executing the following commands to compile and run the app:

```
javac App.java
java App
```

As all of the code we use is contained within one directory **and** only uses code from within the JDK, we only had to specify the `App.java` file as a compilation target for the `Calculator.java` file to also be compiled. If we wanted to be more explicit, we could also do:

```
javac App.java Calculator.java
```

or even use a wildcard expansion to say all `.java` files in the current directory:

```
javac *.java
```

20.3 Compiling multiple files spread across packages

We will use the calculator example from the last example, **Compiling multiple files in the same directory**. This time though, the `App.java` file will be in the package `com.qa`, whereas the `Calculator.java` file will be in the `com.qa.util` package. We have also moved all the source files inside a `src` directory to keep everything tidy. Here is the `Calculator` class:

```
package com.qa.util;

```

```
public class Calculator {
    public <T extends Number> double add(T num1, T num2) {
        return num1.doubleValue() + num2.doubleValue();
    }

    public <T extends Number> double minus(T num1, T num2) {
        return num1.doubleValue() - num2.doubleValue();
    }
}
```

The `App` class will depend on the `Calculator` class, and also be the entrypoint to our application:

```
package com.qa;

import com.qa.util.Calculator;

public class App {
    public static void main(String[] args) {
        Calculator c = new Calculator();
        System.out.println("int(5) + int(5) = " + c.add(5, 5));
    }
}
```

We can then compile and run the application using the following commands from the root directory of the project:

```
javac -sourcepath src src\com\qa\App.java src\com\qa\util\*.java
java -classpath src .\src\com\qa\App
```

The `-sourcepath` flag is followed by a path to where our source files are, this helps the compiler when compiling our program. The `-classpath` option is followed by a path indicating where the compiled Java source code files are, the `.class` files.

20.4 Compiling to a target directory

We have so far learned how to compile basic Java programs, but the compiled code (`.class` files) ends up in the same directories as the source code (`.java` files). This isn't easy for us to maintain and reason about, so let's learn how to output the compiled Java files to a different directory.

We will use the calculator example from previous examples:

```java
package com.qa.util;

public class Calculator {
    public <T extends Number> double add(T num1, T num2) {
        return num1.doubleValue() + num2.doubleValue();
    }

    public <T extends Number> double minus(T num1, T num2) {
        return num1.doubleValue() - num2.doubleValue();
    }
}
```

The `App` class will depend on the `Calculator` class, and also be the entrypoint to our application as before:

```java
package com.qa;

import com.qa.util.Calculator;

public class App {
    public static void main(String[] args) {
        Calculator c = new Calculator();
        System.out.println("int(5) + int(5) = " + c.add(5, 5));
    }
}
```

The difference comes into how we use `javac` and `java` to compile and run our program respectively, make sure to run the following commands from the project root:

```
javac -d target -sourcepath src src\com\qa\App.java src\com\qa\
    util\*.java
java -classpath target com.qa.App
```

When we compile the app, we now include a `-d` flag which is followed by the location to place the compiled files. It will be created if it does not exist, the Java compiler will also mirror your package structure. Next, we run the application using the `java` command. First, we specify the location(s) of any required classes (`target` directory in this case) and then the **fully qualified name** of the class which represents the applications entry point, `com.qa.App` in this case.

> The `-classpath` argument also has a short form, `-cp`. We can specify multiple locations for the compiler to search for required class files by separating each path by a colon:

```
javac -classpath <CLASSPATH1>:<CLASSPATH2>:<ETC...>
```

This is also how we would include `.jar` files.

20.5 Including external dependencies (.jar files)

A **JAR**, Java Archive, is a special filetype which acts almost like a ZIP folder. It acts as a wrapper around some compiled Java source code which any JVM implementation can then run on any platform using a single command. These JAR files can be used to represent two things, programs and libraries. A program is exactly as it sounds, a **library** is some existing set of classes which we can use in our program. Third-party dependencies normally come in the form of a JAR file, so we must learn how to link these during compilation.

First, create a directory for your project and then lets run some commands inside that directory to get everything set up:

```
mkdir lib src target
curl https://repo1.maven.org/maven2/com/googlecode/lanterna/
    lanterna/3.1.1/lanterna-3.1.1.jar -o lib/lanterna-3.1.1.jar
```

This will create the initial project structure and download the dependency for Lanterna from Maven Repository. Lanterna is a text UI library for Java. Here is the code for the application:

```
package com.qa;

import com.googlecode.lanterna.terminal.DefaultTerminalFactory;
import com.googlecode.lanterna.terminal.Terminal;

import java.io.IOException;

public class App {
    public static void main(String[] args) throws
        InterruptedException {
        DefaultTerminalFactory factory = new
            DefaultTerminalFactory();
        Terminal terminal = null;

        try {
            terminal = factory.createTerminal();

            terminal.putCharacter('H');
            terminal.putCharacter('i');
```

```
            terminal.flush();

            Thread.sleep(5000);

            terminal.bell();
            terminal.flush();
            Thread.sleep(200);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (terminal != null) {
                try { terminal.close(); }
                catch (IOException e) { e.printStackTrace(); }
            }
        }
    }
}
```

This will open a new Lanterna terminal, then print `Hi` to it, waits 5 seconds, rings a bell and then closes. We can compile and run this application as follows:

```
javac -d target -sourcepath src -classpath "lib\lanterna-3.1.1.
    jar" src\com\qa\*.java
java -classpath "target;lib\lanterna-3.1.1.jar" com.qa.App
```

- For the Windows CMD terminal, when we need to specify multiple items in the classpath we should use a semi-colon (`;`) instead of a colon to separate each item.

Let's break down the commands step by step. First, the compilation with `javac`:

- `-d target`: The location to output the compiled class files to
- `-sourcepath src`: The location of our source code (`.java` files)
- `-classpath "lib\lanterna-3.1.1.jar"` : The location of any required dependencies, `.class` files that is.
- `src\com\qa\*.java`: Following our options, we list the files to compile using a wildcard (there is just one, `App.java` in this case)

Now let's consider running the compiled code:

- `-classpath "target;lib\lanterna-3.1.1.jar"` : We tell the JVM where it can find the classes for the application. Here, we are saying look in the `target` directory for class files and also look in the `lib\lanterna-3.1.1.jar` file as well.

- `com.qa.App`: The entrypoint of our application

20.6 Creating a JAR file

A JAR file, as prior mentioned, is a Java Archive which contains resources for and related to a Java project. They can be used to represent libraries or programs, and can be executed using a simple command: `java -jar`.

We will use a simple *Hello World* application for the example:

```
1 package com.qa;
2
3 public class App {
4     public static void main(String[] args) {
5         System.out.println("Hello World");
6     }
7 }
```

To convert this code into a JAR file, we need to:

1. Create a `manifest.txt` file
2. Compile the source files
3. Convert the compiled class files and manifest into a JAR file

First, the manifest... The **manifest** contains important information on how the JAR file and its content should be interpreted by the JVM, it is metadata describing the JAR's contents in other words. Typical content for this file would include the applications entry point, classpath and version information.

> The manifest file is located inside the JAR, in a special file called `MANIFEST.MF` inside the `META-INF` directory.

Our manifest is in the root of the project and will look as follows:

```
1 Main-Class: com.qa.App
2 Created-By: <Your/software name here...>
3 Class-Path: .
```

The file must end with a newline character to be correctly interpreted. The `Main-Class` is the entry point to the application and `Class-Path` is exactly that, where the class files are located inside the JAR.

Now, lets compile the application to the `target` directory and then create the JAR

file:

```
rmdir /S /Q target
javac -d target\classes -sourcepath src src\com\qa\*.java
cd target\classes
jar cfmv ..\HelloWorld.jar ..\..\manifest.txt .
cd ..\..
```

We can then run the JAR file like so:

```
java -jar target\HelloWorld.jar
```

20.7 Creating a fat/uber JAR

A **fat JAR** is a special kind of JAR, it is one which includes program dependencies. The regular JAR file we previously created can only use libraries defined within the JDK itself when using just the `java -jar` command. If we wanted to also use external libraries, we would need to specify the classpaths, which is tedious. A fat JAR gets rid of this need by including a copy of the dependencies compiled class files inside the JAR with your code.

Let's start by configuring a project. First, we create the project structure and download our dependency into the `lib` folder:

```
mkdir lib src target
curl https://repo1.maven.org/maven2/com/googlecode/lanterna/
    lanterna/3.1.1/lanterna-3.1.1.jar -o lib/lanterna-3.1.1.jar
```

Here is the code for the application:

```
package com.qa;

import com.googlecode.lanterna.terminal.DefaultTerminalFactory;
import com.googlecode.lanterna.terminal.Terminal;

import java.io.IOException;

public class App {
    public static void main(String[] args) throws
        InterruptedException {
        DefaultTerminalFactory factory = new
            DefaultTerminalFactory();
        Terminal terminal = null;

        try {
```

```
            terminal = factory.createTerminal();

            terminal.putCharacter('H');
            terminal.putCharacter('i');
            terminal.flush();

            Thread.sleep(5000);

            terminal.bell();
            terminal.flush();
            Thread.sleep(200);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (terminal != null) {
                try { terminal.close(); }
                catch (IOException e) { e.printStackTrace(); }
            }
        }
    }
}
```

As before, this will open a new Lanterna terminal, then print `Hi` to it, wait 5 seconds, ring a bell and then close.

As before, we need to compile our application:

```
javac -d target\classes -sourcepath src -classpath "lib\lanterna
    -3.1.1.jar" src\com\qa\*.java
```

Let's not forget to create our manifest in the project root:

```
Main-Class: com.qa.App
Created-By: <Your/software name here...>
Class-Path: .
```

We now need to unpack the external dependency, Lanterna, into the `target\classes` directory. It is important to be aware that conflicting files will be overwritten. We will use the `jar` command with the `xf` options to extract its contents:

```
copy lib\lanterna-3.1.1.jar target\classes\lanterna-3.1.1.jar
cd target\classes
jar xf lanterna-3.1.1.jar
del lanterna-3.1.1.jar
cd ..\..
```

- the `x` option indicates to extract files from a JAR archive
- the `f` option indicates that the JAR to extract files from is specified via the command line instead of stdin

Now we have all the required class files in one place, not hidden in JAR archives, we can create our own JAR file:

```
cd target\classes
jar cfmv ..\app.jar ..\..\manifest.txt .
cd ..\..
```

All of the above blocks of batch commands have been added to files, they can be called using the `build.bat` which contains:

```
rmdir /S /Q target
call init.bat
call compile.bat
call copy_deps.bat
call build_jar.bat
```

We can now run the JAR file using:

```
java -jar target\app.jar
```

21 Local Variable Type Inference

Java 10 introduced a new language feature known as **local variable type inference**, and it means exactly what it states - that a local variable can have its type inferred. The aim of local variable type inference is to increase the readability of code by reducing the boilerplate in variable declarations that results from generic types. You can also read the style guidelines.

21.1 The `var` keyword

In Java, the keyword `var` is a reserved type name that can be used to declare variables using local type inference. The general structure looks as follows:

```
1 var name = value;
```

It is important to remember that variables can only be declared with `var` when a **non-null initialiser** (the value) is supplied. For type inference to work, a value must be assigned at declaration - for example:

```
1 var str = "Hello World"; // String
2 var num = 32; // int
3
4 var x; // illegal
```

The first two declarations are valid (line 1 and 2), at compile-time the compiler will replace the type name `var` with the appropriate data type - `String` and `int` respectively in this case.

The final example, line 3, will not compile as no value has been supplied as the type cannot be inferred.

Once a variable has been declared and initialised using the `var` keyword, it becomes a variable of the initialisers type. This means we cannot assign a value of

a different type to a variable declared with `var`, it follows the same rules of static typing as everything else in Java:

```
jshell> var str = "Hello";
str ==> "Hello"

jshell> str = 32;
|  Error:
|  incompatible types: int cannot be converted to java.lang.
   String
|  str = 32;
|        ^^
```

22 TODO Design patterns

In this section, we aim to cover:

22.1 TODO What is a design pattern?

22.2 TODO The categories of design pattern in object-oriented programming

22.3 TODO The Strategy Pattern

22.4 TODO The Decorator Pattern

22.5 TODO The Builder Pattern

22.6 TODO The Facade Pattern

22.7 TODO The Adapter Pattern

22.8 TODO The Visitor Pattern

22.9 TODO The Command Pattern

23 TODO SOLID Principles

In this section, we aim to cover:

23.1 TODO What are the SOLID principles?

23.2 TODO Single responsibility principle

23.3 TODO Open-closed principle

23.4 TODO Liskov-substitution principle

23.5 TODO Interface segregation principle

23.6 TODO Dependency inversion principle

24 Generics

Generic programming is known as **parametric polymorphism**, Wikipedia defines this as follows:

> In programming languages and type theory, parametric polymorphism allows a single piece of code to be given a "generic" type, using variables in place of actual types, and then instantiated with particular types as needed. Parametrically polymorphic functions and data types are sometimes called generic functions and generic datatypes, respectively, and they form the basis of generic programming.
>
> (Parametric Polymorphism | Wikipedia)

What this means in Java is that we can create data types or methods whose algorithms can be applied across a variety of different data types, these generics were added to Java in 2004 with version 5 of the JDK. The built-in `Collection` classes are an example of this behaviour in the JVM, we can use a `List<String>` and a `List<Integer>` in the same code. The underlying algorithms which make the `List` work remain the same, it is just the data type they work on that is different. So, why do we use generics then? We use generics to allow applying algorithms across a variety of data types, we also use them as they enforce stronger type checks at compile time; helping to ensure that our code is type-safe. Generics also remove the need for casting, such as in the following example:

```
1 List numbers = List.of(1, 2, 3, 4);
2 Integer number = (Integer) list.get(0);
```

The generic version of the `List` interface removes the need for typecasting:

```
1 List<Integer> numbers = List.of(1, 2, 3, 4);
2 Integer number = list.get(0);
```

In this section, we aim to cover:

24.1 Some pre-requisite knowledge

To successfully work with generics in Java, we should know a couple of things. First is the naming conventions for type parameters, which are parameters that represent a type for a given generic type:

- `E` = Element
- `K` = Key
- `N` = Number
- `T` = Type
- `V` = Value
- `S, U, V, etc...` = 2nd, 3rd, 4th types

As we may already know from instantiating `List` instances, we must perform a **generic type invocation** to create an instance of a generic type. This will replace all values of the specified type parameter, such as `T`, with the specified data type, such as `Integer`. We use the, informally named, **diamond operator** for this:

```
1 List<String> data = new List<String>();
```

From Java 7 onwards, as long as the compiler can infer the type arguments from the context, we don't need to specify it as an argument to `new List<>()`. The variable being assigned to is the context in this case.

24.2 Generic methods

Generic methods are methods which declare their own type parameters, this limits the type parameter's scope to the method where they are declared. Static and non-static methods, alongside class constructors, can be made generic. Here is an example of declaring a generic method which accepts a list of elements of type `T` and a `Consumer` of type `T`:

```
1  public class App {
2      public static void main(String[] args) {
3          List<String> names = List.of("Bob", "Sally", "Fred");
4          forEach(names, s -> System.out.println(s));
5      }
6
7      public static <T> void forEach(List<T> elements, Consumer<T>
           consumer) {
8          for (T element : elements) consumer.accept(element);
9      }
10 }
```

The `forEach` method is indicated to be generic by the type parameters specified before the methods return type, the <T> type parameter. Let's contrast how this might have looked if we wanted it to only deal with strings:

```
public static void forEach(List<String> elements, Consumer<
    String> consumer) {
    for (String element : elements) consumer.accept(element);
}
```

The code is mostly the same, we just have `String` in place of the type parameter `T`. For such a common operation, iterating over a list and applying a consumer to each element, it would be ideal for us to have a generic version available. The generic version means we don't have to retype out the method declaration for every `List` of some type that we want to iterate over. Let's look at a more complex `map` method, which takes a value of type `T` and transforms it to a value of type `R`:

```
public class Main {
    public static void main(String[] args) {
        Function<String, Integer> stringToLength = s -> s.length
            ();
        System.out.println("LENGTH OF 'HELLO': " + map("HELLO",
            stringToLength));
    }

    public static <T, R> R map(T value, Function<? super T, ?
        extends R> mapper) {
        return mapper.apply(value);
    }
}
```

This is a much more complex example. We have two type parameters, `<T, R>`, where `T` is the type of the value to be transformed and `R` is the resulting type. The first parameter, `T value`, is simple to understand. It is the second, `Function<? super T, ? extends R> mapper` that is less understandable at first. We have used the idea of **wildcards** in the generics. The `? extends R` represents an **upper bounded wildcard**, which relaxes the restrictions on the type used. It basically means that the value `R` can be `R` or any of its subclasses. A **lower bounded wildcard** is different, symbolised by `? super T` above - we use a lower bounded wildcard to restrict the type to be the specified type or a supertype. You would use this lower bound when you want to apply the same operations to the specified type and its supertype. In this case, we say that the `mapper` function can map values of type `T` or any of its supertypes; you can read more about this pattern here.

24.3 Generic classes

A generic class is like a generic method, but it works at the class-level instead of being specific to one method. That means the generic type parameters can be used anywhere in the class that they are not shadowed. First, we will use the example of a `Box` class which can store any `Object`:

```
1 public class Box {
2     Object contents;
3
4     public Box(Object contents) { this.contents = contents; }
5
6     public Object getContents() {
7         return this.contents;
8     }
9 }
```

We could then use this box as follows:

```
1 Box numberBox = new Box(20);
2 Integer number = (Integer) numberBox.getContents();
```

To retrieve the contents inside the box in its original form, we must do a typecast. This is an unsafe operation as it will occur at runtime and there is no guarantee that the `contents` of the `Box` instance will contain the correct type. A generic implementation fixes this problem by allowing us to work with a specific type:

```
1 public class Box<T> {
2     T contents;
3
4     public Box(T contents) { this.contents = contents; }
5
6     public T getContents() {
7         return this.contents;
8     }
9 }
```

We can use this generic box as follows:

```
1 Box<Integer> numBox = new Box<>(20);
2 Integer num = box.getContents();
```

No type cast was required, and we can even use the `Box<T>` class with different types like a `Box<String>` or `Box<List<String»` or something else... Now the question is, how does this work? Java has a concept called **type erasure**, which

erases the types specified to generic parameters at *compile-time*. Let's consider our usage of the `Box<T>` above where we specified `T` as an `Integer`, the compiler will actually expand that code to:

```
Box numBox = new Box(20);
Integer num = (Integer) box.getContents();
```

The reason this occurs is to maintain backwards compatibility with older APIs. When we create an instance of a generic type, without specifying a type parameter, they default to `Object`. The Java compiler, as long as our code is used correctly, will automatically apply the correct type cast for each time we call `box.getContents()` on some `Box<T>` instance. Let's consider the following code which uses `Box<T>`:

```
Box<Integer> numBox = new Box<>(20);
Integer num = box.getContents();

Box<String> nameBox = new Box<>("Bob");
String name = box.getContents();
```

The Java compiler would erase the generic types so that the code looks like:

```
Box numBox = new Box(20);
Integer num = (Integer) box.getContents();

Box nameBox = new Box("Bob");
String name = (String) box.getContents();
```

This example demonstrates that type erasure will be applied regardless of the type bound we pass to `Box`.

24.4 Generic interfaces

A generic interface functions like a normal interface, but allowing for type bounds to be specified. Here is an example interface for the `map` method we defined earlier:

```
@FunctionalInterface
public interface Mapper<T, R> {
    R map(T value, Function<? super T, ? extends R> mapper);
}
```

We could use this interface as we did before, with the `map` method, to map a string to an integer length:

```
Mapper<String, Integer> stringToIntMapper = (value, mapper) ->
    mapper.apply(value);
System.out.println("LENGTH OF 'HELLO': " + stringToIntMapper.map
    ("HELLO", s -> s.length()));
```

The difference now is that we create implementations of the `Mapper` type, such as a `Mapper` which can convert `String` data to `Integer` data. When we then call `map`, it is applied on a value and a passed in lambda expression. The benefit of this is that we could create a variety of different `Mapper` instances for mapping different data types. Here is an example which maps a `User` to a `UserDTO`, a type designed for transferring objects between the different layers of a software system. First, here is the `User` class:

```
class User {
    private String username;
    private String password;
    private int age;

    public User(String username, String password, int age) {
        this.username = username;
        this.password = password;
        this.age = age;
    }

    public String getUsername() {
        return username;
    }

    public String getPassword() {
        return password;
    }

    public int getAge() {
        return age;
    }
}
```

It is a simple POJO which has `username`, `password` and `age` fields. The `UserDTO` does not have the `password` field:

```
class UserDTO {
    private String username;
    private int age;

    public UserDTO(String username, int age) {
        this.username = username;
```

```
7           this.age = age;
8       }
9
10      public String getUsername() {
11          return username;
12      }
13
14      public int getAge() {
15          return age;
16      }
17  }
```

Now, we can create a `Mapper` for mapping a `User` to a `UserDTO`. As we create a `Mapper` which works across a `User` and `UserDTO`, we can specify different implementations for mapping (the lambda expression) when we call the `Mapper.map(value, mapperFn)` instance method. An example of this might look as follows:

```
1  Mapper<User, UserDTO> userToDtoMapper = (value, mapper) ->
       mapper.apply(value);
2  User user = new User("bob", "password", 30);
3  UserDTO userDto = userToDtoMapper.map(user, target -> new
       UserDTO(target.getUsername(), target.getAge()));
```

25 Introductory theory: Algorithms and Data Structure

This section will briefly introduce some thoery on the study of algorithms and data structures by introducing some key terminology.

In this section, we aim to cover:

25.1 Types

There are two ways of using the word type:

- A **type** represents a collection of values, typically a set (no duplicates) such as that of the boolean type whose values are only `true` and `false`. The integer type is also a collection of values represented by the set of all whole numbers. These types are also called **simple types** as we generally consider them the primitives of any given language, that is we cannot break them down further without introducing large amounts of complexity by dealing directly with binary representations. This is generally used in a very high-level of abstraction, not caring at all about how the type is implemented or operations.
- An **aggregate type**, synonym **composite type**, is a type made up of other types. Modern versions of the JDK introduced record types, these are an example of composite types. A `User` record could be seen as a composite type, it would be made of other types such as a `Date` for the date of birth and `String` for the names. These bits of information are called **date items**, they are **members** of a type.

There is also the term **data type**, which is a *type* and a collection of operations which can operate on the type. In Java, we have the primitive `int` data type which represents the integer type (the set of all whole numbers) and the basic operations which can be applied to the type; some operations include addition, subtraction, multiplication, division, modulo, etc...

25.2 Abstract data types vs data types

There are two representations for a data type:

- **Logical**: The logical form of a data type is its abstraction, so we call it the **abstract data type** (**ADT**). This represents the data and operations but does not specify an implementation. This is generally used within the mathematical side of computer science (discrete maths) but also has applications within programming. The closest abstraction we have to an ADT in Java is an **interface**, as that is all an ADT really is - an interface, what I like to call a contract, defining the data type and behaviours that operate on it. The interface is the data type, the abstract methods are the operations. This leads us onto the other representation.
- **Physical**: The physical manifestation of a data type is its implementation,

we call this a ***data structure**. In Java, the `List` interface could be considered the abstract data type representation - we know a `List` is some consecutive sequence of data and that we should be able to read and modify the lists contents and properties. Aside from this though, we don't care how it is implemented. Two common physical implementations, the **data type** or concrete type, of the `List` interface in Java (and many other languages) are the `ArrayList` and `LinkedList` classes. The `ArrayList` is backed by an array while the `LinkedList` works through node objects which each have a pointer to the next item in the list. Arrays are stored contigously whereas the linked list stores data somewhat randomly in memory. This has performance implications for different operations. Anyway, the main point is that the `List` interface in Java is the logical form of a data type (ADT) whereas the `LinkedList` and `ArrayList` classes are the physical implementations, the data structures.

26 Connecting to an SQL database using the JDBC API

The **Java Database Connectivity** (**JDBC**) API is a built-in abstraction defining how a client can connect to and perform operations upon a database. To use this API, we must provide a **JDBC driver** which is usually provided by the vendor of a database. MySQL uses the *Connector/J* driver, Java DB comes with its own JDBC driver, etc... The JDBC drivers are categorised as follows:

- **Type 1**: These drivers implement the JDBC as a way to map data to another data access API, the driver is generally dependant on native code (limits portability).
- **Type 2**: These drivers are partially implemented in both Java and native code, the drivers use a native client library which is specific to the data source they connect to.
- **Type 3**: These drivers are completely implemented using Java and communicate with a database through some form of middleware using a *database-independent protocol*. The middleware is responsible for sending and receiving requests between your app and the data source.
- **Type 4**: These drivers are also completely implemented in Java, using a network protocol for a specific data source - the client (your app) will have a direct connection to the data source, unlike a type 3 which relies on some middleware to handle the communication. The MySQL Connector/J is a type 4 driver.

Working with a JDBC database generally involves 5 steps:

1. Establishing a connection
2. Creating a statement
3. Executing a query
4. Processing the `ResultSet` object

5. Closing the `Connection`

Step 5 will depend on whether you are using a connection pool or some other design pattern, a **connection pool** being an object which pools database connections. If a connection is needed, it is retrieved from the pool if a free connection is available.... If there is not a free connection, the pool will generally attempt to open a new connection with the database within the defined limits (number of minimum and maximum connections).

26.1 Project setup

To make working with the JDBC not require a database server to be installed, we will use a file database through the `H2` dependency. We will set this up as a Maven project with the following dependency:

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>2.1.214</version>
</dependency>
```

You should also add a `properties` section to your `pom.xml` file to specify the source encoding and compilation version:

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.
      sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.
      outputEncoding>
  <maven.compiler.release>11</maven.compiler.release>
  <maven.compiler.target>11</maven.compiler.target>
  <maven.compiler.source>11</maven.compiler.source>
</properties>
```

We will also create a `InputUtilities` class inside a `utils` package for getting input from the console, this will abstract the `Scanner` to make the API simpler to work with:

```
package com.qa.jdbc_example.utils;

import java.util.Scanner;

public class InputUtilities {

    private Scanner sc;
```

```
8
9       public InputUtilities(Scanner sc) {
10          this.sc = sc;
11      }
12
13      public int getInteger(String prompt) {
14          do {
15              System.out.print(prompt);
16              try {
17                  return Integer.valueOf(sc.nextLine());
18              } catch (NumberFormatException nfe) {
19                  System.out.println("Please enter a valid number.
                        ");
20                  System.out.print(prompt);
21              }
22          } while (true);
23      }
24
25      public String getString(String prompt) {
26          System.out.print(prompt);
27          return sc.nextLine();
28      }
29  }
```

26.2 Connecting to a JDBC database

To connect to a JDBC database, we will need to supply a JDBC url of the form:

`jdbc:[vendor]:[custom_format]`

Every JDBC URL starts with `jdbc:` followed by the vendor, such as `jdbc:mysql`. What then follows will differ dependant on the database. A simple MySQL connection string might look like: `jdbc:mysql://localhost:3306/`. The different connection strings for the H2 database can be found here. We want the *Embedded (local) connection* for this example, which have the forms:

```
jdbc:h2:[file:][<path>]<databaseName>
jdbc:h2:~/test
jdbc:h2:file:/data/sample
jdbc:h2:file:C:/data/sample (Windows only)
```

We will setup a class precisely for making connecting to a database simpler:

```
1  package com.qa.jdbc_example.utils;
2
```

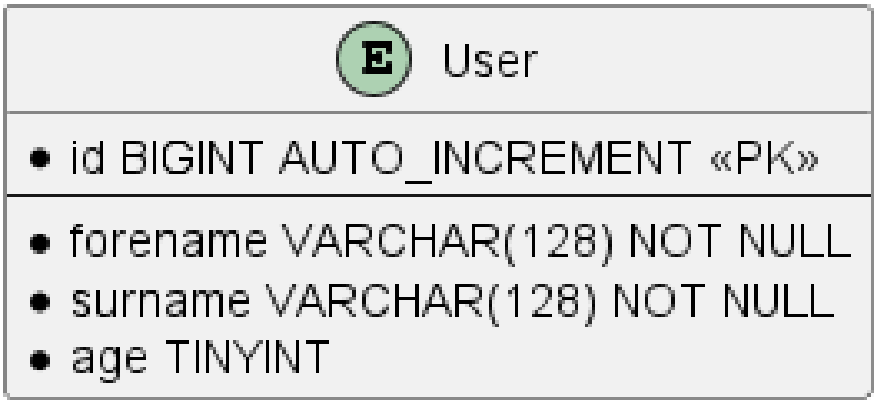
```
3  import java.sql.Connection;
4  import java.sql.DriverManager;
5  import java.sql.SQLException;
6  import java.sql.Statement;
7
8  public class ConnectionUtilities {
9
10     private static final String URL = "jdbc:h2:~/test_h2.db";
11     private static Connection conn;
12
13     public static Connection getConnection() throws SQLException
           , ClassNotFoundException {
14         if (conn == null || conn.isClosed()) {
15             // load the driver and then the connection if we
                   never established one
16             Class.forName("org.h2.Driver");
17             conn = DriverManager.getConnection(URL);
18         }
19         return conn;
20     }
21
22     public static void initialiseDatabase(final String SQL) {
23         if (SQL == null) throw new IllegalArgumentException("
               Must provide SQL to initialise the database");
24
25         try {
26             if (conn == null) conn = getConnection();
27             final Statement statement = conn.createStatement();
28             statement.execute(SQL);
29             System.out.println("Created table");
30         } catch (SQLException | ClassNotFoundException e) {
31             e.printStackTrace();
32             System.err.println("Something went wrong
                   initialising the database, exiting.");
33             System.exit(1);
34         }
35     }
36 }
```

For this example, we connect to a file database using the URL `jdbc:h2:~/test_h2.db`. This file will be created in the users home directory, symbolised by the tilde character. We also have a field `static Connection conn`, we are using the **Singleton** pattern for this app as there is only one user of the app and we do not require a connection pool. If this was an actual production app, consider a proper solution which makes use of a connection pool to enable and manage concurrent connections to a database. In the `getConnection()` method, we simply check if the `conn` object has been initialised, if it hasn't we initialise it and then re-

turn the connection object. In the `intialiseDatabase` method, we accept a string of SQL to initialise the database with. We first get our `Connection` object, we then create a `Statement` object using the connection `conn`. We then call `statement.execute(SQL)` with the passed in string of SQL to be executed against the database. If something goes wrong during initialisation, we indicate this by exiting with a code 1: `System.exit(1)`. An exit code of `0` signifies the app ran successfully.

26.3 The domain of the application

Before building up an application, we have to consider a domain. For this example, we will choose a simple `User` domain for which we represent the data of a systems users. This will be kept simple for example purposes, our entity will have the following model:



The SQL for this table is as follows:

```
CREATE TABLE IF NOT EXISTS `user` (
        `id` BIGINT AUTO_INCREMENT,
        `forename` VARCHAR(128) NOT NULL,
        `surname` VARCHAR(128) NOT NULL,
        `age` TINYINT,
        PRIMARY KEY (`id`)
);
```

We also supply some test data for inserting into the database:

```
INSERT INTO `user` (`forename`, `surname`, `age`)
VALUES ('Fred', 'Smith', 32), ('Aiden', 'Walker', 27), ('Sarah',
    'Hills', 19);
```

Our domain, `users`, will have a *Plain Old Java Object* (POJO) representation in our application:

```
1  package com.qa.jdbc_example.domain;
2
3  import java.util.Objects;
4
5  /**
6   * User is a domain object (represents the target business
        domain), also represents the entity in our database
7   * @author Morgan Walsh
8   *
9   */
10 public class User {
11
12     private long id;
13     private String forename;
14     private String surname;
15     private int age;
16
17     public User() { super(); }
18
19     public User(String forename, String surname, int age) {
20         super();
21         this.forename = forename;
22         this.surname = surname;
23         this.age = age;
24     }
25
26     public User(long id, String forename, String surname, int
           age) {
27         super();
28         this.id = id;
29         this.forename = forename;
30         this.surname = surname;
31         this.age = age;
32     }
33
34     public long getId() { return id; }
35
36     public void setId(long id) { this.id = id; }
37
38     public String getForename() { return forename; }
39
40     public void setForename(String forename) { this.forename =
           forename; }
41
```

```
42      public String getSurname() { return surname; }
43
44      public void setSurname(String surname) { this.surname =
            surname; }
45
46      public int getAge() { return age; }
47
48      public void setAge(int age) { this.age = age; }
49
50      // hashCode is used internally by certain data structures to
             optimise access times to data
51      // - this is in structures such as HashMaps and HashSets
52      @Override
53      public int hashCode() { return Objects.hash(age, forename,
            id, surname); }
54
55      // we use this for comparing two objects to see if they are
            equal, do their fields
56      // contain the same data, very important for testing
57      @Override
58      public boolean equals(Object obj) {
59          if (this == obj)
60              return true;
61          if (obj == null)
62              return false;
63          if (getClass() != obj.getClass())
64              return false;
65          User other = (User) obj;
66          return age == other.age && Objects.equals(forename,
                other.forename) && id == other.id
67                  && Objects.equals(surname, other.surname);
68      }
69
70      @Override
71      public String toString() {
72          return "User [id=" + id + ", forename=" + forename + ",
                surname=" + surname + ", age=" + age + "]";
73      }
74
75
76  }
```

This sets us up ready for working the JDBC, in the next subsection we will look at creating the data access object which connects to our database.

26.4 The Data Access Object (DAO)

The **Data Access Object** will be our abstraction over database communication using the JDBC. Our DAO will have the following package statement, imports and type definition:

```
1  package com.qa.jdbc_example.domain;
2
3  import java.sql.Connection;
4  import java.sql.PreparedStatement;
5  import java.sql.ResultSet;
6  import java.sql.SQLException;
7  import java.sql.Statement;
8  import java.util.ArrayList;
9  import java.util.List;
10
11 import com.qa.jdbc_example.utils.ConnectionUtilities;
12
13 public class UserDAO {}
```

The first method we will implement is the `findAll()` method, which looks as follows:

```
1  public List<User> findAll() {
2      try {
3          Connection conn = ConnectionUtilities.getConnection();
4          Statement statement = conn.createStatement();
5          ResultSet rs = statement.executeQuery("SELECT * FROM `
               user`");
6          List<User> users = unwrapSet(rs);
7          return users;
8      } catch (SQLException | ClassNotFoundException e) {
9          e.printStackTrace();
10         throw new RuntimeException("Something went wrong
               connecting to the user database");
11     }
12 }
```

This method first retrieves a `Connection` object, it then creates a `Statement` which represents an SQL statement to execute against a database. For executing queries which do not modify a database, we use the `Statement.executeQuery(sql)` instance method. This returns a `ResultSet`, which contains the results of our query against the database. We then use a `unwrapSet` method defined on the DAO to get the `User` data out of the result set. This utility method looks as follows:

```
1 private List<User> unwrapSet(ResultSet rs) throws SQLException {
2     List<User> users = new ArrayList<>();
3
4     while (rs.next()) {
5         User user = unwrap(rs); // get a user out of the result
              set
6         users.add(user); // add them to a list for returning
              after
7     }
8     return users;
9 }
```

This method is nice a simple, we set up an empty `List<User>` for containin the users retrieved from the database. We then call `rs.next()` to check if there is a next row, by default the `ResultSet` instance returned from a query points to 1 row before the first row. We must call `rs.next()` at least once to move the record pointer to the next table record (the first row). This returns `true` if there is more records available, and `false` otherwise. Inside of the `while` statement, we use a custom `unwrap` method for actually getting the user data out of the result set:

```
1  private User unwrap(ResultSet rs) throws SQLException {
2      User user = new User();
3
4      Long id = rs.getLong("id");
5      String forename = rs.getString("forename");
6      String surname = rs.getString("surname");
7      int age = rs.getInt("age");
8
9      user.setId(id);
10     user.setForename(forename);
11     user.setSurname(surname);
12     user.setAge(age);
13     return user;
14 }
```

All we do to retrieve the data is call the appropriate `getXXX` methods on the `ResultSet` object for the type we require, we pass the column names to indicate which data we actually want to retrieve. Once we have retrieved the data, it is a simple case of setting up a `user` object and returning it.

The final method demonstrated here is a `save` method, for saving a new user into the database. This will make use of a `PreparedStatement` object, which will help prevent SQL injection attacks by separating the data from a query when sending it to a data source. On the `PreparedStatement` object, we will also call `executeUpdate()` when we are ready to either insert, update or delete data from

a database; `executeQuery` is used for non-destructive operations:

```
public User save(User user) {
        final String SQL = "INSERT INTO `user` (`forename`, `
            surname`, `age`) VALUES (?, ?, ?);";
        try {
            Connection conn = ConnectionUtilities.getConnection
                ();
            PreparedStatement ps = conn.prepareStatement(SQL,
                Statement.RETURN_GENERATED_KEYS);
            // bind data to the prepared statement to prevent
                SQL injection
            ps.setString(1, user.getForename());
            ps.setString(2, user.getSurname());
            ps.setInt(3, user.getAge());

            // executeUpdate() is used for INSERT, UPDATE and
                DELETE
            int modifiedCount = ps.executeUpdate();

            // try retrieving the primary key of the new user
            if (modifiedCount > 0) {
                ResultSet keys = ps.getGeneratedKeys();
                if (keys.next()) {
                    user.setId(keys.getLong(1));
                    return user;
                }
                return user;
            }
        } catch (SQLException | ClassNotFoundException e) {
            System.out.println("Something went wrong saving the
                user to the database.");
            e.printStackTrace();
        }
        return null;
    }
```

We also make use of the `Statement.RETURN_GENERATED_KEYS` constant value to indicate that we want the primary key of a new database record to be returned, if the vendor supports it.

26.5 The Controller

The `Controller` is a gatekeeper type, we normally use them for directing control flow to the relevant service. For this example, we will use it as a way of executing a command from the user:

```java
package com.qa.jdbc_example.controller;

import java.util.List;

import com.qa.jdbc_example.domain.User;
import com.qa.jdbc_example.domain.UserDAO;
import com.qa.jdbc_example.utils.InputUtilities;

public class UserController {

    private InputUtilities inputUtils;
    private UserDAO repo;

    public UserController(InputUtilities inputUtils, UserDAO
        repo) {
        this.inputUtils = inputUtils;
        this.repo = repo;
    }

    public User createUser() {
        String forename = inputUtils.getString("Forename: ");
        String surname = inputUtils.getString("Surname: ");
        int age = inputUtils.getInteger("Age: ");
        return repo.save(new User(forename, surname, age));
    }

    public List<User> readAll() {
        return repo.findAll();
    }
}
```

It is a simple and small class which requests user input, and then delegates the work to the `UserDAO`.

26.6 The App class

Something we generally try to avoid is putting the main logic for an application inside the `main` method, this helps with maintainability. We have a `App` class for this, it has the following package statement, imports and type definition:

```java
package com.qa.jdbc_example;

import com.qa.jdbc_example.controller.UserController;
import com.qa.jdbc_example.domain.User;
import com.qa.jdbc_example.utils.InputUtilities;

```

```
7 import java.util.List;
8
9 public class App {}
```

This pulls in the required data types for use in this class. We have two fields, a `InputUtilities` and `UserController`, which are both used by the programs main loop:

```
1  public class App {
2
3      private InputUtilities inputUtilities;
4      private UserController controller;
5
6      public App(InputUtilities inputUtilities, UserController
           controller) {
7          this.inputUtilities = inputUtilities;
8          this.controller = controller;
9      }
10
11 }
```

Now that we have the initial structure out of the way, lets start adding the implementation. We will kick this off with the beating heart, the main program loop:

```
1  public class App {
2
3      public void run() {
4          boolean isRunning = true;
5
6          do {
7              printMenu();
8              isRunning = executeInput(inputUtilities.getString(">
                   "));
9          } while (isRunning);
10         System.out.println("Goodbye...");
11     }
12
13 }
```

It is a simple `do-while` loop which iterates while `isRunning` is `true`. On each iteration, we print the menu using `printMenu()` and execute the users input using `executeInput`. The `printMenu()` method looks as follows:

```
1  public class App {
2
3      private void printMenu() {
4          System.out.println("C) Create a new user");
```

```
        System.out.println("R) Display all users");
        System.out.println("E) Exit");
    }
}
```

This shouldn't require any explaining aside from that `C`, `R` and `E` are the command keys to enter. Let's take a look at `executeInput` now:

```
public class App {
    private boolean executeInput(String input) {
        boolean output = true;
        switch (input.toUpperCase()) {
            case "C":
                User newUser = controller.createUser();
                if (newUser != null) {
                    System.out.println("Created a new user");
                    System.out.println(newUser);
                    return true;
                } else {
                    System.out.println("Something went wrong
                        creating the user!");
                }
                break;
            case "R":
                List<User> users = controller.readAll();
                for (User user : users) {
                    System.out.println(user);
                }
                break;
            case "E":
                // return false to signal exit
                output = false;
                break;
            default:
                System.out.println("Invalid input supplied,
                    please try again.");
        }
        return output;
    }
}
```

This is just a simple `switch` statement which dependent on the input, chooses a specific option. If we have the letter `C` as input, we call `controller.createUser` to hand the flow of control to the controller. The same can be said for `R`, except that we call `controller.readAll()` instead.

26.7 Finally, running the application

Now that we have the application setup, we can load up the database and run the app:

```
package com.qa.jdbc_example;

import com.qa.jdbc_example.controller.UserController;
import com.qa.jdbc_example.domain.UserDAO;
import com.qa.jdbc_example.utils.ConnectionUtilities;
import com.qa.jdbc_example.utils.InputUtilities;

import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        String userSql = "CREATE TABLE IF NOT EXISTS `user` ("
                + "   `id` BIGINT AUTO_INCREMENT,"
                + "   `forename` VARCHAR(128) NOT NULL,"
                + "   `surname` VARCHAR(128) NOT NULL,"
                + "   `age` TINYINT,"
                + "   PRIMARY KEY (`id`)"
                + ");";
        String dataSql = "INSERT INTO `user` (`forename`, `surname`, `age`) "
                    + "VALUES ('Fred', 'Smith', 32), ('Aiden', 'Walker', 27), ('Sarah', 'Hills', 2);";
        // uncomment line below to include some default data
//        userSql += dataSql;
        ConnectionUtilities.initialiseDatabase(userSql);

        InputUtilities utils = new InputUtilities(new Scanner(System.in));
        App app = new App(utils, new UserController(utils, new UserDAO()));
        app.run();
    }
}
```

27 Basic Git usage

Git is a version control system developed by Linus Torvalds in the 1990's as a way of managing projects, it replaced tools such as SVN as the main VCS in-use.

> This section of the document assumes you already have `git` installed on your machine. You can confirm this by typing `git --version` into your console. If you get a version number printed to the console, you are ready to proceed. If you get an error, check that you do have Git installed.

The terminal commands used in this section assume a Unix-like environment, such as Git Bash for Windows, Windows Subsystem for Linux (WSL), Linux distributions or macOS distributions. If on Windows, using `CMD` or `PowerShell`, you will need to replace forward slashes `/` with backslashes in file paths. You will also need to `%USERPROFILE%` to represent your users home directory instead of `~/`.

You should read the following subsections in order, as each subsection builds upon the prior.

27.1 Initialising a new local repository

Git works using the concept of a repository, a **repository** being a storage area for some given work. On a file system, a Git repository is any folder/directory which contains a hidden `.git` folder:

```
| project1
| -- .git/
| -- src/
| -- pom.xml
```

If we wanted to initialise a new local Git repository from the terminal, we can use the `git init` command.

```
mkdir ~/projects
cd ~/projects
rm -rf git_project_1
mkdir git_project_1
cd git_project_1
git init
```

We get the following results from running the commands:

```
Initialized empty Git repository in C:/Users/morga/projects/git_project_1/.git/
```

We can confirm this is a Git repository now by doing an `ls -a` in the terminal (from the `~/projects/git_project_1` directory), we should see a folder called `.git`. You can also run `git status` to confirm this as well. `git status` will indicate that a repository doesn't exist in the case that the current directory does not contain a valid `.git` folder. We could check the project we created as follows:

```
cd ~/projects/git_project_1
git status
```

Which will then output:

```
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

As we can see, we are on the `main` branch of the repository with nothing committed - this indicates we are inside a Git repository.

27.2 Adding work to the stage

The **stage** is an intermediate area in a Git repository that is used as a holding area for changes to a repository. This means, we could consider changes to be in one of three states:

- Unstaged: The change has not been added to the stage yet
- Staged: The change has been added to the stage, in preparation for comitting to the local repository

- Committed: The change has been taken off the stage and stored permanently in the repository

Let's take a look at the current state of our projects stage using `git status`:

```
cd ~/projects/git_project_1
git status
```

This produces the following indicating that we have no staged changes to commit, and we also have no changes to add to the stage:

```
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Let's add a file to our repository, and then check the status of the stage again:

```
cd ~/projects/git_project_1
echo "Hello World" > test.txt
git status
```

This will produce the following results:

```
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
test.txt

nothing added to commit but untracked files present (use "git add" to track)
```

As we can see, there is an `Untracked file` called `text.txt`. To add this to the stage, we need to specify the `git add <args>` command where `<args>` represents what we want to add to the stage. We can add this as follows:

```
git add .
git status
```

We used `git add .` to say that we want to add everything to the stage, that is not currently staged or tracked by the local repository, that is in the current directory

and its subdirectories. We could also add just a specific file or folder to the stage, such as the `test.txt` file, using `git add test.txt`. The output is as follows:

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
new file:   test.txt
```

Git indicates that we have a `new file` on the stage, meaning that the file was not previously tracked by the repository. This file is still not tracked by the repository for versioning, it is currently in the holding area (the stage). This holding area, the stage, holds the changes we might want to commit to a repository.

27.3 Committing work to the local repository

In the previous section, we saw how to add work to the stage. Our repository, when we run `git status`, will give us the following output:

```
On branch main

No commits yet

Changes to be committed:
   (use "git rm --cached <file>..." to unstage)
    new file:   test.txt
```

We will now commit these changes using `git commit -m "commit message"` to the repository:

```
1 git commit -m "Added test.txt file to repository"
```

Committing work is the process of telling Git that you want the staged work to be version controlled by Git, in response Git will add the file to the repositories internal version tracking inside the `.git` folder in the root of the repository. We get a commit number back reflecting this, `3ce31b0` in the below example:

```
[main (root-commit) 3ce31b0] Added test.txt file to repository
```

```
 1 file changed, 1 insertion(+)
 create mode 100644 test.txt
```

27.4 Viewing the previous commits

Git provides the `git log` command for showing the commit history of a repository, it's usage is as follows:

```
1 git log
```

For the repository from previous examples, `/projects/git_project_1`, `git log` would produce:

```
commit 3ce31b00ef2548e0610e4cc51e0be2754f448d34
Author: MrWalshyType2 <mwalsh@qa.com>
Date:   Mon Jun 5 14:06:43 2023 +0100

    Added test.txt file to repository
```

This shows all of our commits to the local repository in the console. You can navigate using the up and down keys, or the page up and down keys. Press the `q` key in the terminal to quit the text viewer that you are inside, known as a *paging tool* as it allows us to page through text.

We can also view the commit history in short form by using the `--oneline` flag:

```
1 git log --oneline
```

This would produce the following output, reflecting our single commit:

```
2c32e31 Added other.txt file
3ce31b0 Added test.txt file to repository
```

27.5 Changing branches

Git allows us to create *branches* where a **branch** represents some area of work in the repository. At a minimum, we usually have two branches:

- **develop**: The branch in which in-development code is saved to
- **main**: The branch in which ready-to-deploy, production code is saved to

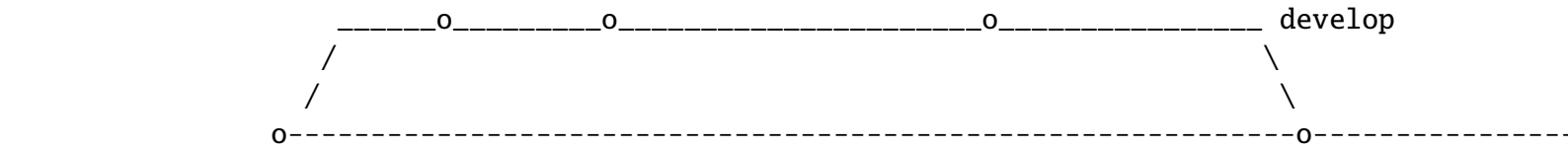
So far, we have worked on the `main` branch - we can check which branch we are on by running:

```
git branch
```

The output shows a list of branches with a star next to `main`, indicating that we are on the `main` branch:

```
* main
```

When we create a new branch, we create it so that it *diverges* off of the current branch:



Each dot on the above diagram represents a commit, the bottom line is the `main` branch which our production code lives on. When we diverge from `main` onto `develop`, we take 1 commit with us. There are then a number of commits made on the `develop` branch, which is then merged back into `main` on the right-side of the diagram. Let's now create our own `develop` branch:

```
git branch develop main
git checkout develop
git branch
```

We first run `git branch develop main`, this creates the `develop` branch off of the `main` branch - usage is as follows: `git branch <new_branch> <base_branch>`. If we are already on the `base_branch` which the new one will diverge from, we could also do `git checkout -b <new_branch_name>` to create the new branch instead. The `git branch` command ran at the end of the last few commands will output the following:

```
* develop
  main
```

This confirms that we have created the `develop` branch and then *checked it out*. We **checkout** the branch we want to work on. We can then add new files, modify existing ones, etc... add the changes to the stage and then commit those changes.

Let's add a new file into the `develop` branch:

```
echo "Hi there" > other.txt
git add .
git commit -m "Added other.txt file"
```

We will get the following output to the console, indicating that we are now tracking the `other.txt` file on our `develop` branch:

```
[develop 2c32e31] Added other.txt file
 1 file changed, 1 insertion(+)
 create mode 100644 other.txt
```

We can confirm this by looking at the logs:

```
git log --oneline
```

This will show us two commits, the commit `3ce31b0` came from the `main` branch whereas the `2c32e31` commit came from the `develop` branch. Both of the commits are present on the `develop` branch:

```
2c32e31 Added other.txt file
3ce31b0 Added test.txt file to repository
```

27.6 Merging branches

Merging branches is the process of taking one branches commits and joining them with another branch, so that the other branch reflects all of the work done. This process only works if both branches have a common base commit and their trees have not fell out of alignment. To merge the `develop` branch into the `main` branch, we can use the `git merge <branch>` command. We will first have to checkout the `main` branch as this is the branch we want to merge changes into. Our merge will look as follows:

```
git checkout main
git merge develop
git log --oneline
```

The output of the commands follows:

```
Updating 3ce31b0..2c32e31
Fast-forward
 other.txt | 1 +
```

```
 1 file changed, 1 insertion(+)
 create mode 100644 other.txt
2c32e31 Added other.txt file
3ce31b0 Added test.txt file to repository
```

We first see that a fast forward merge was used to efficiently join together the develop and main branches, this is done in a fast-forward merge by aligning the head of the target branch (`main` in this case) with the branch that is being merged in (`develop` in this case).

You can read more about merging here.