Algorithms and Data Structures (WIP)

Based on the online MSc in Computer Science by the University of York
Notes taken by Nathan McKeown-Luckly (Discord Username: skeletman)

Autumn 2023

Contents

Chapter 1

The Java Programming Language

1.1 Software and the Java Programming language

1.1.1 Software and Code

Definition 1.1.1 (Computer Program)**.** A *program* is a set of instructions to be given to a computer.

Definition 1.1.2 (Software)**.** *Software* means a program or set of programs.

Definition 1.1.3 (Application Software)**.** *Application software* or *applications* are software that is designed for the user to interact with the computer.

Examples. Word processor, image editor, computer game, spreadsheet.

Definition 1.1.4 (System Software)**.** *System software* is software that provides a platoform for other software to function.

Examples. Operating system, network software, game engine, search engine, compiler.

Definition 1.1.5 (Embedded Software, Embedded System)**.** *Embedded software* refers to software designed to control devices that are do not primarily function as computers. Such devices are known as *embedded systems*.

Examples. Microwave ovens, cars, modems, robots.

A computer can only understand instructions given to it in binary, we call such instructions *machine code*. However, this is not very convenient for humans,

so we instead write in a human-readable *programming language*, for example *Java*. The code we write in these are translated into machine code by a piece of software called a *compiler*, to be executed by the computer.

Definition 1.1.6 (Program code/Source Code)**.** Code as it was written by a human, often in a programming language easily readable to humans, is called *program code* or *source code*.

Definition 1.1.7 (Syntax)**.** The rules that must be followed for a piece of code to be valid in a programming language is called *syntax*.

When code is written in this document, it will be presented in a box with line numbers on the left hand side. Blue font will be used for Java keywords, purple for string literals, and green for code comments. We see a sample of such a presentation in listing 1.1.1.

Listing 1.1.1: A sample of code

```
1  /*
2  A sample of Java code.
3  */
4  public class HelloWorld
5  {
6      public void main(String[] args)
7      {
8          System.out.println("Hello World");
9      }
10 }
```

When defining pieces of syntax, I will use *italic grey* font to represent fields where the programmer needs to choose the value. For example, we will see in section 1.4.4, the definition of a for loop will look as follows, to indicate that *initialisation*, *termination*, *increment* and *code* should be replaced to fit the programmer's needs

```
for(initialisation; termination; increment)
{
    code
}
```

1.1.2 The Java Programming Language

Java is an *object oriented* programming language.

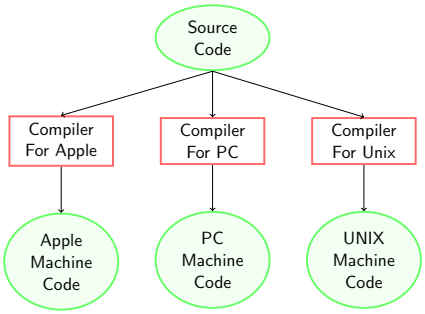
Definition 1.1.8 (Object Oriented Programming)**.** *Object oriented programming* or *OOP* is a programming paradigm based on the notion of objects. Objects may contain both data (fields, attributes or properties), and code (methods).

Example. Suppose we wish to simulate a physical scenario, where a cube has some force applied to it. In such a physics simulation, a cube object may have a field `mass`, and a method `calculateAcceleration` which takes an input of a force, and uses the `mass` field to calculate the acceleration that the cube would experience.

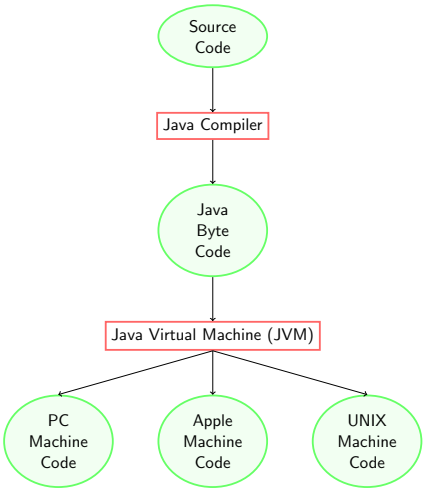
There are other programming paradigms: languages such as Haskell use the functional programming paradigm, which differs greatly from object oriented programming.
Compilers for many programming languages, such as C, compile program code directly into machine code. In figure 1.1.1a we see that this requires a different compiler for each type of machine for the program to be run on. However, Java is *platform-independent*: a compiled Java program can run on any type of machine. This is because Java compiles to *Java byte code*, which then can be run by the *Java virtual machine* or *JVM*, a piece of software that runs on various types of machines to translate the Java byte code into machine code. There are versions of JVM for Windows, Mac, Unix, Linux, mobile phones, and even many embedded systems. For a diagram see figure 1.1.1b.

Figure 1.1.1: A comparison between the compilation pipelines in other languages vs Java

(a) A typical pipeline for compilation



(b) The pipeline for Java compilation



Definition 1.1.9 (Libraries, Packages)**.** *Libraries* and *packages* are pre-compiled Java modules for use in your own programs.

The *Java Runtime Environment* or *JRE* is the JVM along with the Java libraries. The *Java Development Kit* or *JDK* consists of the JRE, the Java compiler, and some other tools for java development.

1.1.3 Command Line Interfaces (CLIs) and Integrated Development Environments (IDEs)

Henceforth we assume that the JDK is properly installed. If we wish to compile Java code using the command line, it should be saved as a .java file. Then we use `javac` to compile the program into a `.class` file, which contains Java byte code. Then to launch this with the JVM, we run the `java` command. For example, to compile a file called `test.java`, we would navigate to its directory, and enter `javac test.java`, and to launch the program we enter `java test`. During development, instead of doing this every time we wish to test a program, programmers will often use an *Integrated Devevlopment Environment* or *IDE* to handle this. An IDE will often have many features designed for development, such as syntax highlighting, version management tools and debugging tools.

1.1.4 User Interfaces

Definition 1.1.10 (User Interfaces)**.** A *user interface* or *UI* is the part of a program that prompts the user for input and provides the program output.

A *console application* is an application which uses a *text based user interface*. Alternatively, an application can use a *graphical user interface* or *GUI*, which allows for elements such a text boxes and buttons.

1.1.5 "Hello World" Program

Once a Java program is compiled, running the program will execute the `main` method in the class of the .java file that was compiled. In listing 1.1.1, the `main` method is executed and hence the words "Hello World" are printed to the console when the program is executed. For more on methods, see section 1.5. For more on classes, see section 1.6.

1.2 Variables

1.2.1 Primitive Types

Definition 1.2.1 (Data type)**.** A *data type* is a collection of data values, or the representation of these values on a machine.

In the Java programming language, the *primitive types* or *scalar types* are a collection of basic types which take up a fixed amount of memory. The primitive types along with the data which they store are displayed in figure 1.2.1.

Figure 1.2.1: Primitive types in Java

Java type	Description of Data	Range of values
`byte`	Integers	-128 to 127
`short`	Integers	$-32,768$ to $32,767$
`int`	Integers	$-2,147,483,648$ to $2,147,483,647$
`long`	Integers	$-9,223,372,036,854,775,808$ to $9,223,372,036,854,775,807$
`float`	Real numbers	$\pm 1.4 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$
`double`	Real numbers	$\pm 4.9 \times 10^{-324}$ to $\pm 1.8 \times 10^{308}$
`char`	Characters	Unicode character set
`boolean`	True or false	True or false

1.2.2 Declaring and assigning variables

Definition 1.2.2 (Variable and Declaration)**.** When we create a named location in memory to store a value of a given type, we are *declaring* a *variable*. They are called variables as the content may vary during the runtime of the program.

In Java, we declare variables by writing: `dataType variableName`. For example, if we wanted to declare an integer variable named score, we'd write

```
1 int score;
```

Definition 1.2.3 (Assigning variables)**.** When we give a variable a value, this is called *assigning* the variable a value.

In Java, we use the = symbol to assign variables, for example

```
score = 10;
```

We can declare and assign in the same line of code:

```
int score = 10;
```

Variables can be declared as *constant* by being preceded by the keyword `final`.

```
final double speedOfLight = 300000000;
```

1.2.3 Arithmetic Operators

Arithmetic operations take numeric values and output a numeric value of the same type. In Java, these can be done using the operators listed in figure 1.2.2.

Figure 1.2.2: Arithmetic operations in Java

Operation	Java operator
Addition	+
Subtraction	-
Multiplication	*
Division	/
Remainder or Modulus	%

Definition 1.2.4 (Overloading)**.** A method or operator is said to be *overloaded* if it has multiple definitions, which may depend on context such as the data type or number of inputs.

Example. The arithmetic operations are overloaded so we can use them on all of the primitive numeric types. For example / is defined for both `int` and `double`. In the `int` case, it returns a rounded down value, whereas in the `double` case it returns a precise (as possible given double precision) value: `3 / 2` will have value `1` of type `int`, but `3.0 / 2.0` will have value `1.5` of type `double`.

1.2.4 Comparison Operators

Comparison operations take numeric values and output a boolean value. In Java, these can be done using the operators listed in figure 1.2.3.

Figure 1.2.3: Comparison operations in Java

Operation	Java operator
Equal to	==
Not equal to	!=
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=

Note. The == operator (and by extension !=) only compares the value for *primitive types*. We will see later in section 1.8.3 that for objects, == is unsuitable for comparing values.

1.2.5 Logical Operators

Logical operations take boolean values and output a boolean value. These can be done using the operators listed in figure 1.2.4.

Figure 1.2.4: Logical operations in Java

Operation	Java operator
Logical NOT	!
Logical OR	||
Logical AND	&&

1.3 Program design

Definition 1.3.1 (Program Design and Implementation)**.** *Program design* is working out how to go about developing software. *Implementation* is the process of developing the software.

1.3.1 Pseudocode

Pseudocode refers to code written with very loose syntax, not bound by that of any specific language.

Algorithm 1: An example of pseudocode to calculate a price after tax

```
1 display prompt for price;
2 price ← user input;
3 display prompt for tax percentage;
4 tax ← user input;
5 return price * (1 + tax/100);
```

Algorithm 2: An alternative way of writing the same algorithm as in algorithm 1

```
1 DISPLAY prompt for price;
2 ENTER price;
3 DISPLAY prompt for tax percentage;
4 ENTER tax;
5 RETURN price * (1 + tax/100);
```

1.4 Selection and Iteration

1.4.1 If Statements

An *if statement* executes a piece of code conditional on a boolean condition. The Java syntax for an if statement is

```
if(condition)
{
    code_if_condition_true
}
```

1.4.2 If-Else Statements

An *if-else statement* executes a piece of code conditional on a boolean condition, and if not executes another piece of code. The Java syntax for an if-else statement is

```
if(condition)
{
    code_if_condition_true
}
else
{
    code_if_condition_false
}
```

1.4.3 Switch Statements

A *switch statement* is a switch between many pieces of code based on the value of a varaible. In Java, the syntax for a switch statement is

```
switch(variable)
{
    case value1: code
        break;
    ...
    case valueN: code
        break;
    default: code
        break;
}
```

1.4.4 For Loops

A *for loop* iterates a piece of code a fixed number of times, with a varying parameter. The Java syntax for a for loop is

```
for(initialisation; termination; increment)
{
    code
}
```

The code in the header behaves as follows

- *initialisation* runs once, at the beginning of the loop's execution
- *termination* should be a boolean expression: when this evaluates to `false` the loop will terminate.
- *increment* runs after each iteration of the loop

Example. A typical use of a for loop would be to add all the integers between 0 and 10. This would be written as follows

Listing 1.4.1: A typical use of a for loop

```
1 int total = 0;
2 for(int i = 0; i <= 10; i++)
3 {
4     total += i;
5 }
```

In listing 1.4.1 we initialise a new variable `int i = 0`, and each iteration of the loop `i++` is executed, incrementing `i` by 1. The loop terminates when `i <= 10` no longer holds.

Note. When a variable is declared in the initialisation of a for loop, its scope is within that for loop.

1.4.5 While loops

A *while loop* iterates a piece of code while a boolean condition is true. The Java syntax for a for loop is

```
while(condition)
{
    code
}
```

1.4.6 Do-While loops

A *do-while loop* executes a piece of code once, then iterates it while a boolean condition is true. The Java syntax for a for loop is

```
do
{
    code
} while (condition)
```

1.4.7 Break statements

While inside a loop, you can use a *break statement* to terminate iteration of the loop.

Example. If you wanted to terminate a for loop when its counter reaches 3, you could write the following

```
for(int i = 0; i < 100; i++){
    if (i == 3)
    {
        break;
    }
}
```

1.4.8 Continue statements

While inside a loop, you can use a *continue statement* to skip to the next iteration of the loop.

Example. If you wanted to skip an iteration in a for loop when its counter reaches 3, you could write the following

```
for(int i = 0; i < 100; i++){
    if (i == 3)
    {
        continue;
    }
}
```

1.5 Methods

Methods are pieces of code that perform some operation, possibly with parameters that affect that operation. The operation may return a result, or affect the state of the program. Methods can be *called* from elsewhere in the program to run their code, and they may be called multiple times.

1.5.1 Declaring, Defining and Calling Methods

Methods are defined using the following syntax:

```
modifiers return_type method_name(parameters)
{
    method_body
}
```

Convention is that a method name will have the first letter lowercase. The first line of the method declaration is called the *method header*.

Example. We could use a method to take an integer and return it multiplied by 2.

Listing 1.5.1: An example method

```
1 public static int multByTwo(int x)
2 {
3     return x * 2;
4 }
```

In this example `public` and `static` are modifiers, `int` is the return type, `multByTwo` is the method name, `int x` is the parameter, and `return x * 2;` is the method body.

Methods are called using the following syntax:

```
method_name(parameters)
```

Example. If we wanted to call the method in listing 1.5.1 to multiply 5 by 2 and assign it to variable `num`, we would write the following code.

```
1 int num = multByTwo(5);
```

We could call the method `multByTwo` twice if we so wished:

```
1 int num = multByTwo(multByTwo(5));
```

Methods may affect the state of the program, so sometimes we don't need to put the output into a variable. We may use `void` as the return type for such methods. Regardless, any method can also be run without assigning the return value to a variable. The following example would run the method multByTwo, but not put the return value in a variable.

```
multByTwo(5);
```

1.5.2 Modifiers for Methods

In a method header in Java, you can use *modifiers* to modify various behaviours of the method: one *access modifier* then potentially a combination of *non-access modifier*s. The access modifiers are as follows

- `default` (if you don't specify a modifier) - The method is accessible only in the same package.
- `public` - The method is accessible in all classes.
- `private` - The method is only accessible in the declared class.
- `protected` - The method is accessible in the same package and subclasses.

The non-access modifiers are

- `final` - The method cannot be overridden or modified.
- `static` - The method belongs to the class rather than objects of that class.
- `abstract` (can only be used in an abstract class) - The method has no method body, which must be supplied by any subclass.
- `transient` - Makes the method ignored when serialising the object containing it.
- `synchronized` - The method can be accessed only by a single thread at a time.

1.5.3 Method Overloading

It is allowed in Java to give multiple methods the same name, as long as they accept a different set of parameters. This is called *overloading*. For example, the following is valid Java code.

```
static int max(int firstIn, int secondIn)
{
    if(firstIn > secondIn)
    {
        return firstIn;
    }
    else
    {
        return secondIn;
    }
}

static int max(int firstIn, int secondIn, int thirdIn)
{
    return(max(firstIn, max(secondIn, thirdIn)))
}
```

Definition 1.5.1 (Polymorphism)**.** Methods and operators with the same name being allowed to perform different function is called *polymorphism*.

1.6 Classes and Objects

In definition 1.1.8, we defined object oriented programming, and said, objects may contain both data (fields, attributes or properties), and code (methods). The idea is that a *class* is a blueprint for an *object*.

1.6.1 Class Declaration

The syntax for class declaration is as follows:

```
modifiers class class_name
{
    class_body
}
```

Convention is that a class name will have the first letter capitalised.

1.6.2 The Class Body

The class body may consist of method and attribute declarations. You may also provide a special method called a *constructor*, which has the same name as the class, and the return type is not written. For example, if we were to run a physics simulation of an object with mass under a gravitational force, a class `PhysicsObject` may be defined as follows

Listing 1.6.1: PhysicsObject

```
1  public class PhysicsObject
2  {
3      private double mass;
4      private static double gravity = 9.81;
5
6      public PhysicsObject(double massIn){
7          mass = massIn;
8      }
9
10     public static void setGravity(double gravityIn){
11         gravity = gravityIn;
12     }
13
14     public double getGravityForce(){
15         return mass * gravityConst;
16     }
17 }
```

Notice that we have declared the attributes as private, so the only way to access the attributes are through the class' methods. This kind of data protection is called *encapsulation*.
Also, we have *initialised* the `gravity` attribute, which sets the value at the start of the lifetime of the `PhysicsObject` class, or if it were not `static`, at the start of the lifetime of each `PhysicsObject` object. Java always initialises attributes to a given value, numerical primitives to 0, `boolean` to `false`, `Object` to `null`

1.6.3 Instantiating an object

Creating a new object is called *instantiation*. This is achieved in Java as follows

```
class_name object_name = new class_name(parameters)
```

For example, to create an object called `box` of the `PhysicsObject` class above, with mass 5.0, we would write

```
1 PhysicsObject box = new PhysicsObject(5.0)
```

1.6.4 Accessing the Public Methods and Attributes of an Object or Class

To access the public methods of an object we have instantiated, we use the *dot operator*.

```
object_name.method_name(parameters)
```

Similarly for attributes we use

```
object_name.attribute_name
```

For example, to get the force acting on our box

```
1 double force = box.getGravityForce();
```

If the `static` modifier has been used in the definition, instead we need to refer to the class name rather than the object name, for example, to set the gravity constant to 1, we would write

```
1 PhysicsObject.setGravity(1.0);
```

1.6.5 The "this" keyword

The `this` keyword refers to the object which owns a method or constructor. It is useful to distinguish between class attributes and parameters with the same name.

1.6.6 Modifiers for Classes

When declaring an class, you can use *modifiers* to modify various behaviours of the class: one *access modifier* then potentially a combination of *non-access modifier*s. The access modifiers are as follows

- `default` (if you don't specify a modifier) - The class is accessible only by classes in the same package.
- `public` - The class is accessible in any other class.

The non-access modifiers are

- `final` - The class cannot be inherited by other classes.
- `abstract` - Objects of this class cannot be created.

1.6.7 Modifiers for Attributes

When declaring an attribute, you can use *modifiers* to modify various behaviours of the attribute: one *access modifier* then potentially a combination of *non-access modifier*s. The access modifiers are as follows

- `default` (if you don't specify a modifier) - The attribute is accessible only in the same package.
- `public` - The attribute is accessible in all classes.
- `private` - The attribute is only accessible in the declared class.
- `protected` - The attribute is accessible in the same package and subclasses.

The non-access modifiers are

- `final` - The attribute cannot be overridden or modified.
- `static` - The attribute belongs to the class rather than objects of that class.
- `transient` - Makes the attribute ignored when serialising the object containing it.
- `synchronized` - The attribute can be accessed only by a single thread at a time.
- `volatile` - The attribute is always read from main memory, not from a thread-local cache. In practice, this means any thread will be able to read the most recently updated value for this attribute.

1.7 Inheritance

1.7.1 Superclasses and Subclasses

Often we wish to make classes which represent a "kind of" another class. To do this, we use inheritance.

Definition 1.7.1 (Inheritance)**.** *Inheritance* refers to fields and methods being shared between classes. In particular, a newly defined class, the *subclass*, will have the fields and methods of some previously defined class, the *superclass*, but can also implement new fields and methods of its own.

In Java, the syntax used for inheritance is the `extends` keyword,

```
modifiers class subclass extends superclass
{
    code
}
```

Example. We define a class `Book` with fields `title` and `contents` which are both of type `String`, as well as a constructor which sets these values, and a method `display` which prints to console the title and contents of the book

Listing 1.7.1: Book

```
1  public class Book
2  {
3      private String title;
4      private String contents;
5
6      public Book(String title, String contents){
7          this.title = title;
8          this.contents = contents;
9      }
10
11     public void display(){
12         System.out.println("Title: " + title);
13         System.out.println("Contents: " + contents);
14     }
15 }
```

Suppose we further wanted to define a class `Textbook` which is a book with an extra `String` field `subject`, and a new constructor. We would use inheritance as follows:

Listing 1.7.2: Textbook

```
public class Textbook extends Book
{
    private String subject;

    public Textbook(String title, String contents, String subject){
        this.title = title;
        this.contents = contents;
        this.subject = subject;
    }
}
```

1.7.2 Method Overriding

In a subclass, we can redefine some methods that are already defined in the superclass, in particular, those with the `public` or `protected` modifier (or `default` if the superclass lies in the same package as the subclass). To override a method in Java, you need only define the method as usual, although you can optionally add the `@Override` annotation beforehand, which tells the compiler to check that this method name has been used in the superclass.

Example. As it stands, the `Textbook` class is not very useful, as the `display()` method does not show the subject, so we override it

Listing 1.7.3: Textbook

```
public class Textbook extends Book
{
    private String subject;

    public Textbook(String title, String contents, String subject){
        this.title = title;
        this.contents = contents;
        this.subject = subject;
    }

    @Override
    public void display(){
        System.out.println("Title: " + title);
        System.out.println("Subject: " + subject);
        System.out.println("Contents: " + contents);
    }
}
```

1.8 Assignment and Equality

1.8.1 The Assignment Operator (=)

The = operator assigns a value to a variable. For example, the following assigns the value 10 to `x`.

```
int x = 10;
```

When dealing with objects instead of primitives, the value that is assigned is the memory address of the object on the right hand side.

Example. Suppose we have a class `Rectangle`

Listing 1.8.1: Rectangle

```
public class Rectangle
{
    private double length;
    private double width;

    public Rectangle(double length, double width)
    {
        this.length = length;
        this.width = width;
    }

    public setLength(double length)
    {
        this.length = length;
    }

    public setWidth(double width)
    {
        this.width = width;
    }

    public double getArea()
    {
        return length*width;
    }
}
```

The listing 1.8.2, line 1 creates a new `Rectangle` in memory with length 1.0 and width 1.0, and assign `rect` to its memory address.
In line 2, `rect2` would be assigned the value of `rect`, i.e. the memory address of the rectangle.
Then in line 3, we use the `setWidth` method to set `rect2.width` to 5.0. As `rect2` points to the same object as `rect`, this method sets `rect.width` to 5.0.
In line 4, we set `area` to `rect.getArea()`. This returns `5.0`, as `rect.length` is 1.0 and `rect.width` is 5.0.

Listing 1.8.2: A sample of code to interact with the Rectangle class

```
1 Rectangle rect = new Rectangle(1.0,1.0);
2 Rectangle rect2 = rect;
3 rect2.setWidth(5.0);
4 double area = rect.getArea();
```

Definition 1.8.1 (Aliasing)**.** *Aliasing* is when an object is assigned multiple names. In listing 1.8.2, we see `rect2` is an *alias* of `rect`.

1.8.2 The new Keyword

The *new keyword* creates a new object, at runtime it assigns an area in memory on the heap to store that object, and returns a reference to that object. It has one argument, being the constructor of the object to be initialised.

1.8.3 The Equality Operator (==)

The == operator will tell you if two non-null references refer to the same object in memory. This is called *reference equality*. In practice, this means that when comparing objects using ==, you are comparing their memory address, not their data. **WARNING: It is a common mistake to use == to compare the data contained by an object. You should instead use a method which compares the data you require to be compared (often the overridden `equals` method).** With the following declarations, we would have that `a==b` would return `true`, but `a==c` would return `false`, despite `a` and `c` containing the same data!

```
1 String a = new String("str");
2 String b = a;
3 String c = new String("str");
```

In contrast, for primitives, the == operator returns the *value equality*, i.e. true if and only if the variables contain the same value.

1.9 Common Classes

1.9.1 Object

Every class is a subclass of `Object`, so inherits its methods. Examples of such methods are

- `public boolean equals(Object obj)` - Returns true if the current object is equal to `obj`. The equals method for the class `Object` this is the same as the `==` operator. **WARNING: This means that two objects may be indistinguishable by the data and methods they hold, but are still distinguished by the equals method!** As a result, for most classes the equals method is overriden.
- `protected Object clone()` - Produces a clone of the current object, such that `x.clone() == x` will return false.
- `public String toString()` - Returns a string representation of the object. Usually overridden to give more useful information.

1.9.2 String

A *string* is a sequence of characters. The `String` class represents *immutable* strings, that is, they cannot be modified by methods.

Definition 1.9.1 (String Literal)**.** A *string literal* is a piece of code which represents a fixed string.

In Java, string literals are represented as a string within quotation marks, for example `"This is a string literal"`. String literals are implemented as instances of the class `String`.
Use of the same string literal twice will reference to the same `String` object in memory. For example, with the following declarations, `a == b` would return `true`.

```
1 String a = "str"
2 String b = "str"
```

The string class has many associated methods, such as:

- `public int length()` - Returns the length of the string.
- `public boolean equals(Object obj)` - Returns true if and only if `obj` is a `String` object and has the same value as the current string. Overrides the `equals` method for the `Object` class.

- `public String substring(int beginIndex, int endIndex)` - Returns the substring of this string beginning with the `beginIndex`th character, ending with the `(endIndex-1)`th character.
- `public String toLowerCase()` - Returns the string in lowercase.
- `public String toUpperCase()` - Returns the string in uppercase.

1.9.3 Wrappers

For each primitive data type, there is a *wrapper class* which hold a single field of the corresponding primitive type.

Figure 1.9.1: The wrapper classes

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Autoboxing allows us to use primitive values in place of wrapper objects, and *unboxing* allows vice versa.

1.9.4 Scanner

The *Scanner class* allows text input to the program, with ability to parse primitive types, and use regular expressions to parse strings. Scanners can scan from user input to a console, from files, and from String objects.
This class comes in a package called `util`. We use the `import` keyword to import packages.

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

or we can use an asterisk to import the whole package

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

Listing 1.9.1: Sample use of the Scanner class

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
long j = sc.nextLong();
String k = sc.next();
System.out.println(i + ", " + j + ", " + k);
sc.close();
```

In listing 1.9.1, 3 user inputs are taken from the console, and they are outputted separated by commas.

1.10 Arrays

An *array* stores a fixed number of values of the same type.

Note. Arrays do not entirely follow object oriented philosophy, we will look at other collection structures that conform to the object oriented philosophy much better.

In Java, the syntax to declare an array is to use square brackets after the type declaration.

type[] *array_name*;

1.10.1 Assigning Arrays

There are multiple ways to assign an array in Java.

To assign an array with all default values,

array_name = new *type*[*capacity*];

To assign an array using an array literal,

array_name = {*comma_separated_values*};

Alternatively, you can write

array_name = new type[]{*comma_separated_values*};

Like variables, you can do the declaration and assignment in one line.

Examples. Array declarations and assignments

```
int[] a = {1,2,3};
double[] b;
b = new double[5];
char[] c = new char[] {'A', 'Z'};
```

1.10.2 Accessing Array Elements

To access the i^{th} element of an array called `array_name` in Java, we use square brackets

```
array_name[i]
```

Note that array indices start from 0, so in the previous example, `a[0]` is 1 and `c[1]` is `'Z'`.

1.10.3 Arrays of Objects

An array of objects is in fact an array of references, in particular, the array can contain multiple of the same reference, so can refer to the same object multiple times.

1.10.4 Multi-dimensional Arrays

A *multi-dimensional array* is an array of arrays, which may themselves be multi-dimensional arrays. In Java, to declare a 2-dimensional array, we would write the following

```
type[][] array_name;
```

For more dimensions, we would write more square brackets.

Definition 1.10.1 (Jagged and Rectangular Arrays)**.** A multi-dimensional array is *rectangular* if in each dimension it always has the same length. Otherwise, the array is called *jagged*.

We can assign multi-dimensional arrays similarly to 1-dimensional arrays, here we do 2-dimensional arrays.

To assign a rectangular 2-dimensional array of dimensions `capacity_x` and `capacity_y`, with all default values,

```
array_name = new type[capacity_x][capacity_y];
```

You may also assign the length of each element of the array separately to form a jagged array, for example to declare a new array of `capacity_x` arrays, where the i^{th} had capacicty `capacity_y_i`, the following would be written.

```
array_name = new type[capacity_x][];
array_name[i] = new type[capacity_y_i];
```

You can also assign a 2-dimensional array using array literals,

```
array_name = {comma_separated_arrays};
```

To access the elements, we use the same syntax as for 1-dimensional arrays

array_name[*i*][*j*]

All of this notation extends to higher-dimensional arrays by increasing the number of square brackets

Examples. If we wanted to create a rectangular array of doubles called a, with dimensions 3 and 4, containing all 0s, we would write

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

If we wanted to create a jagged array of integers called a, with 4 rows, and the i^{th} row has i columns, and a[i][j] has value i+j, we could write the following

```
int[][] a = new int[4][];
for(i = 0; i < 4; i++)
{
    a[i] = new int[i];
    for(j = 0; j < i; j++)
    {
        a[i][j] = i + j;
    }
}
```

Alternatively, we would get the same result by the following

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

1.11 Dynamically Sized Collections

Arrays can be quite limiting, as they have a fixed number of elements. To get around this, we instead use a data structure that can store an arbitrary number of elements.

1.11.1 ArrayList

One example of a dynamically sized collection in Java is *ArrayList* in the `util` package

```
1 import java.util.ArrayList;
```

An `ArrayList<T>` object stores an array of type `T`, and if an index needs to be accessed beyond the array's size, dynamically resizes the array. As a result, we obtain a data structure that can store an arbitrary number of elements.

Chapter 2

Algorithms & Data Structures

Preamble

In this and further chapters, I will make use of various logical symbols, figure 2.0.1 indicates their meanings

Figure 2.0.1: Logical symbols

Symbol	Meaning
$\top$	True
$\bot$	False
$\wedge$	Logical and
$\vee$	Logical or
$\veebar$	Logical xor
$\neg$	Logical not
$\forall$	For all
$\exists$	There exists
$\exists!$	There exists a unique
$\nexists$	There does not exist
$\implies$	Logically implies
$\iff$	Logically equivalent to/If and only if
$=$	Propositionally equals
$:=$	Is defined as
$\in$	Is an element of
$\subseteq$	Is a subset of
$\supseteq$	Is a superset of

2.1 Introduction

Definition 2.1.1 (Computational Problem)**.** A *computational problem* is a specified desired output of a computer program given specified input data.

Definition 2.1.2 (Algorithm)**.** An *algorithm* is a set of instructions. In our use case, an algorithm must consist of basic instructions which a computer can carry out.

Here is an example computational problem:

Problem 2.1.3 (The Maximum Problem)**.** Given a finite sequence of integers, find the maximum

Input: A sequence $(a_1, \ldots, a_n)$ of integers.

Output: a_k such that $\forall i \in \{1, \ldots, n\},\ a_k \geq a_i$.

Algorithm in Plain English (Maximum Problem)**.** The idea is to store a varaible which stores the maximum so far as we search the sequence, and compare each new element to this.

1. Decalare a variable `currentMax`
2. Set `currentMax` to a_1.
3. For each $i \in \{2, \ldots, n\}$, if $a_i >$ `currentMax` then set `currentMax` to a_i.
4. Return `currentMax`.

2.1.1 Psuedocode

Pseudocode is a language with very loose syntax, designed to be easily readable to humans. We use the following conventions, but diferrent people use different conventions.

- We use $=$ for assignment, $==$ for equality.
- The body of a loop, if-statement or if-else-statment will be indented
- // indicates a line comment, and /* */ indicate a block comment.
- Variables have scope restricted to the current procedure.
- Objects are assigned by reference, i.e. if x is an object, and we use the assignment $y \leftarrow x$, then y and x refer to the same object.

- Objects are passed by reference.
- `NIL` is a reference to a null object.
- Parameters are passed by value.
- Arrays are treated as objects.
- We can refer to elements of an array using square brackets.
- The **return** keyword will return control to the calling method. It may also return one or many values.
- Boolean operators are *short-circuiting*, they evaluate from left to right, and will return a value as soon as their value is determined.
- The **error** keyword is used to indicate an error. The calling procedure will handle errors.

Example. Suppose the data we would like to find the maximum of is stored in an array A of length n. We could write the previous algorithm in pseudocode as follows.

Algorithm 3: max(A)

```
1 currentMax = A[1];
2 for i = 2 to n do
3 |   if currentMax < A[i] then
4 |   |   currentMax = A[i];
5 |   end
6 end
7 return currentMax;
```

2.1.2 Algorithm Analysis

The Experimental Approach

The experimental approach to algorithm analysis is to implement the algorithm, then measure the time taken for the algorithm to give output on a range of test data. This data can then be analysed using a range of methods such as data plots.

The experimental approach has the following limitations

- The algorithm must be implemented.
- It takes time to run the algorithm.
- Results will be limited to the specific data tested in the experiment.
- Results are dependent of hardware and software environments.

The Theoretical Approach

The theoretical approach is to use a mathematical description of the algorithm, and then calculate the theoretical running time of the algorithm. When taking this approach, we make the following assumptions

- We use a *random access machine* model, that is, the CPU can access arbitrarily many memory cells in constant time, given an address.
- Instructions are executed in series.
- Basic instructions such as evaluating expressions, assigning value to variables, indexing an array, calling methods and returning from methods are considered to have constant time cost.

We analyse pseudocode, giving each line a time cost, and calculate the number of times that line will run.

Example. An analysis of algorithm 3. Time $= a + e + (b + c + d)(n - 1)$.

Algorithm 4: max(A)

```
1 currentMax = A[1];                          // cost a, times 1
2 for i = 2 to n do                           // cost b, times n-1
3 |   if currentMax < A[i] then                // cost c, times n-1
4 |   |   currentMax = A[i];                  // cost d, times n-1
5 |   end
6 end
7 return currentMax;                          // cost e, times 1
```

Asymptotic Bounds on Running Times

When analysing the runtime or memory usage of programs, the constants are often unimportant. Hence we often classify these using *asymptotic bounds*, which tell us about the behaviour of our programs with large inputs.

Intuition. We define a notation, big-Theta. $f(x) = \Theta(g(x))$ means, for large inputs, $f(x)$ grows at approximately the same relative rate as $g(x)$. In practice, this means we can ignore all but the fastest growing term in the formula for $f(x)$.
To come up with a proper definition, we define two weaker notions, big-O and big-Omega. $f(x) = O(g(x))$ means, for large inputs, $f(x)$ grows at a slower or equal relative rate as $g(x)$, and $f(x) = \Omega(g(x))$ means, for large inputs, $f(x)$ grows at a faster or equal relative rate as $g(x)$.
In practice, we use big-O and big-Omega for things that may vary based on the input of the algorithm, such as running time, and use big-Theta for things that do not depend on the input, for example best/worst case running times.

Examples. Correct usage of Big-Theta notation.

- $x^2 + 2x + 1 = \Theta(x^2)$
- $2^x + x^{99} = \Theta(2^x)$
- $x \log(x) + x = \Theta(x \log(x))$

Correct usage of Big-O notation.

- $x^2 + 2x + 1 = O(x^4)$
- $2^x + x^{99} = O(2^x)$
- $x \log(x) + x = O(x^2)$

Correct usage of Big-Omega notation.

- $x^2 + 2x + 1 = \Omega(x)$
- $2^x + x^{99} = \Omega(2^x)$
- $x \log(x) + x = \Omega(1)$

Example. We calculated the runtime of algorithm 3 to be $a + e + (b + c + d)n$ for input an array of length n, for some constants a, b, c, d and e. We see that this grows linearly, independent of input, so the runtime is $\Theta(n)$.

Example. A problem, algorithm, and runtime analysis.

Problem 2.1.4 (The Array Element Problem)**.** Given an array A and a value x, determine if x is an element of A.

Input: An array A and value x

Output: True if x is in A, False otherwise.

Let $A.length = n$.

Algorithm 5: LinearSearch(A, x)

```
1 i = 0;                                   // cost a, times 1
2 while i < n do                    // cost b, 1 <= times <= n
3 |   if A[i] == x then              // cost c, 1 <= times <= n
4 |   |   return True;                    // cost d, times <= 1
5 |   end
6 end
7 return False;                            // cost e, times <= 1
```

We see that, worst case, we have runtime

$$a + d + e + (b + c)n = \Theta(n)$$

In best case, we have runtime

$$a + b + c + d = \Theta(1)$$

So we can make the following statements:

- The runtime of LinearSearch is $O(n)$.
- The runtime of LinearSearch is $\Omega(1)$.
- The best-case runtime of LinearSearch is $\Theta(1)$.
- The worst-case runtime of LinearSearch is $\Theta(n)$.

Precise Definitions of Asymptotic Bounds

Following are precise mathematical definitions, these are not entirely necessary unless dealing with particularly nasty algorithms.

Definition 2.1.5 (Asymptotic Bound)**.** We say a function f is asymptotically upper bounded by a function g, written $f \lesssim g$, if eventually $f(x)$ is always less than $g(x)$. i.e.

$$\exists t,\ x > t \implies f(x) < g(x)$$

Similarly, we say a function f is asymptotically lower bounded by a function g, written $f \gtrsim g$, if eventually $f(x)$ is always less than $g(x)$. i.e.

$$\exists t,\ x > t \implies f(x) > g(x)$$

Definition 2.1.6 (Big-O)**.** We say a function f is *big-O* of g, written $f(x) = O(g(x))$, if there is a constant c such that f is asymptotically upper bounded by $c \times g(x)$, i.e.

$$\exists c,\ f(x) \lesssim c \times g(x)$$

Definition 2.1.7 (Big-Omega)**.** We say a function f is *big-Omega* of g, written $f(x) = \Omega(g(x))$, if there is a constant c such that f is asymptotically lower bounded by $c \times g(x)$, i.e.

$$\exists c,\ f(x) \gtrsim c \times g(x)$$

Definition 2.1.8 (Big-Theta)**.** We say a function f is *big-Theta* of g, written $f(x) = \Theta(g(x))$, if it is both big-O and big-Omega of g.

2.2 Stacks and Queues

Definition 2.2.1 (Stack)**.** A *stack* is a collection equipped with two operations.

- *push* adds an element.
- *pop* reads and removes the last added element.

It is sometimes called a *LIFO* or *last-in-first-out* collection.

Implementation (Array-Based Stack)**.** The idea of an *array-based stack* is that we can store a pointer to the top of the stack, and use this to implement Push and Pop in $O(1)$ time. Below we use an implementation that checks for underflows, but not overflows (although in this case it can be passed to the array's overflow handling).
A stack S will have 2 attributes: an array S, of size $S.length$, and a positive integer $S.top$.

Algorithm 6: Stack-Empty(S)

```
1 if S.top == 0 then
2 |   return True;
3 else
4 |   return False;
5 end
```

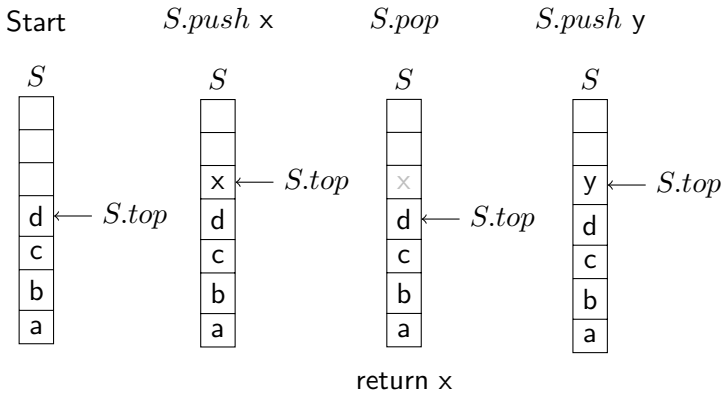
Algorithm 7: Push(S, x)

```
1 S.top = S.top + 1;
2 S[S.top] = x;
```

Algorithm 8: Pop(S)

```
1 if Stack-Empty(S) then
2 |   error Underflow;
3 else
4 |   S.top = S.top − 1;
5 |   return S[S.top + 1];
6 end
```

Figure 2.2.1: A diagram illustrating Array-Based Stacks



Definition 2.2.2 (Queue)**.** A *queue* is a collection equipped with two operations.

- *enqueue* adds an element.
- *dequeue* reads and removes the first added element.

It is sometimes called a *FIFO* or *first-in-first-out* collection.

Implementation (Array-Based Circular Queue)**.** The idea of an *array-based circular queue* is that we can add two attributes that point to the beginning (head) and end (tail) of the queue, so we can implement Enqueue and Dequeue in $O(1)$ time. Additionally, we imagine the end of the array wraps around to the beginning, as to not cause overflows when we have space free at the beginning of the array. There are many possible implementations, here is one that does not handle overflow and underflow.
An array-based circular queue Q will have 3 attributes: An array Q, of size $Q.length$, and positive integers $Q.head$ and $Q.tail$.

Algorithm 9: Enqueue(Q, x)

```
1 Q[Q.tail] = x;
2 Q.tail = (Q.tail + 1)%Q.length;
```

Algorithm 10: Dequeue(Q)

```
1 x = Q[Q.head];
2 Q.head = (Q.head + 1)%Q.length;
3 return x;
```

Recall, $x\%y$ is the remainder of x when divided by y. In the case of queues, this means that if the head/tail is at the end of the array, and we wish to dequeue/enqueue, the new value wraps back to 0. An if-else-statement could have been used instead in both dequeue and enqueue.

Below are diagrams illustrating the operation of queues.

Figure 2.2.2: A diagram illustrating Array-Based Circular Queues

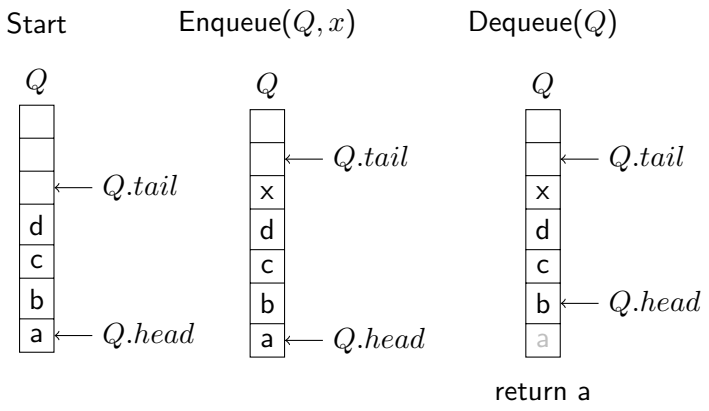
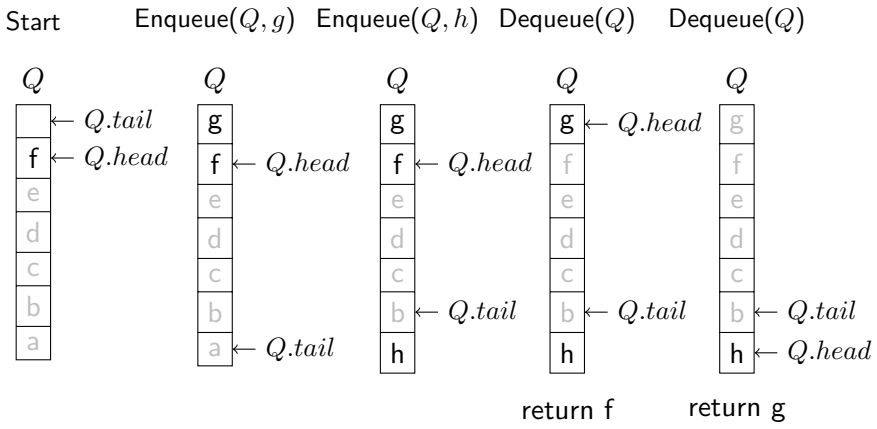


Figure 2.2.3: A diagram illustrating wrapping in Array-Based Circular Queues



Implementation (Array-Based Circular Queue with Error Handling)**.** Our previous implementation did not handle overflow or underflow errors. One way of checking for overflow and underflow is to interpret $Q.head = Q.tail$ as "The queue is empty". This only allows us to store $Q.length - 1$ elements, as if, from empty, we attempt to enqueue $Q.length$ elements, we will return to having $Q.head = Q.tail$.
An array-based circular queue Q will have 3 attributes: An array Q, of size $Q.length$, and positive integers $Q.head$ and $Q.tail$.

Algorithm 11: Queue-Empty(Q)

```
1 if Q.head == Q.tail then
2 |   return True;
3 else
4 |   return False;
5 end
```

Algorithm 12: Queue-Full(Q)

```
1 if Q.head == (Q.tail + 1)%Q.length then
2 |   return True;
3 else
4 |   return False;
5 end
```

Algorithm 13: Enqueue(Q, x)

```
1 Q[Q.tail] = x;
2 if Queue-Full(Q) then
3 |   error Overflow;
4 else
5 |   Q.tail = (Q.tail + 1)%Q.length;
6 end
```

Algorithm 14: Dequeue(Q)

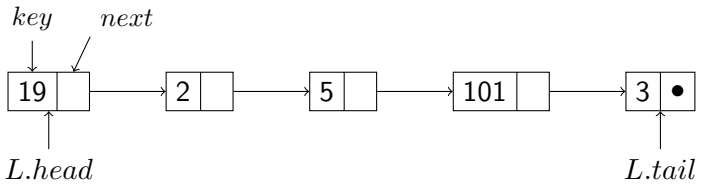
```
1 x = Q[Q.head];
2 if Queue-Empty(Q) then
3 |   error Underflow;
4 else
5 |   Q.head = (Q.head + 1)%Q.length;
6 end
7 return x;
```

2.3 Linked Lists

Here we define a data structure called a *linked list*, which allows for the storage of arbitrarily many elements.

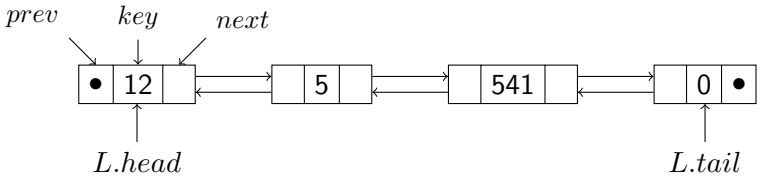
Definition 2.3.1 (Singly Linked List)**.** In a *singly linked list* each element is stored alongside a pointer to the next element.

Figure 2.3.1: A singly linked list L representing the set $\{3, 101, 5, 2, 19\}$



Definition 2.3.2 (Doubly Linked List)**.** In a *doubly linked list* each element is stored alongside pointers to the next and previous elements.

Figure 2.3.2: A doubly linked list L representing the set $\{0, 541, 5, 12\}$



Definition 2.3.3 (Circular Linked List)**.** A singly linked list is circular if the next pointer of the tail points to the head.
A doubly linked list is circular if the prev pointer of the head points to the tail, and the next pointer of the tail points to the head.

2.4 Computational Problems

2.4.1 The Sorting Problem

Problem 2.4.1 (The Comparison Sorting Problem)**.** The *comparison sorting problem* refers to the task of sorting a sequence of values into ascending order.

Input: A sequence $(a_1, \ldots, a_n)$ of values of a type equipped with a total order relation $\leq$.

Output: A permutation $(a'_1, \ldots, a'_n)$ of $(a_1, \ldots, a_n)$ such that

$$\forall i, j \in \{1, \ldots, n\}, \quad i \leq j \implies a'_i \leq a'_j$$

i.e. a permutation in (monotone) ascending order.

Index