CT326

PROGRAMMING III

Andreas Ó hAodha
University of Galway
2023-11-07

Contents

1 Introduction

1.1 Lecturer Contact Information

- Dr. Adrian Clear (`adrian.clear@universityofgalway.ie`).
- Office IT408.

1.2 Assessment

Continuous assessment consisting of five assignments will constitute 30% of the overall mark. The exam will constitute the remaining 70% of the overall mark. The exam will be a written exam on pen & paper. The marking will reflect this (small mistakes, e.g. missing semicolons, will not be penalised). Practising with past exam papers is highly recommended.

1.3 Textbook

The textbook for this course is *Java - How to Program by Deitel & Deitel*, which is available in the University Bookshop. It is not necessary to buy this book for this module.

2 Overloading Constructors, Abstract Methods, & Polymorphism

2.1 Revision

An **object** is something upon which your program performs different operations. Java uses a **class** to represent objects. Each class has a unique name. To create an instance of a class variable, you must use the `new` operator. A class contains members. These may be:

1. Information (or data), often called class variables.
2. Functions (methods) that operate on the data.

2.2 Overloaded Constructors

Methods in the same class may have the same name, so long as they have different **parameter lists**. An overloaded constructor is creating by declaring more than one constructor for one class, each with distinct parameter lists.

2.3 Class Inheritance

When your programs use inheritance, you use a **superclass** to derive a new class. The new class inherits the superclass's members. To initialise class members for an extended class (called a **sub-class**), the application invokes the super class and sub-class constructors. The `this` & `super` keywords are used to refer to the current class & the superclass, respectively.

There are three types of members: **public**, **private**, & **protected**.

	Class	Package	Subclass	World
private	Y	N	N	N
no specifier	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

Figure 1: Access Level Specifiers

The keyword `final` is used to indicate that a class member cannot be overridden or modified.

2.4 Abstract Classes

An **abstract** class is one that cannot be instantiated. They are declared with the keyword `abstract`, e.g.: `public abstract class Employee`. Abstract classes can have instance data and non-abstract (concrete) methods for sub-classes. Abstract classes can have constructors for sub-classes to initialise inherited data. Sub-classes of an abstract class must be also be abstract, and declared with the keyword `abstract`.

3 Command-Line Programming

3.1 Solving Common Coding Problems

- Problem: The compiler complains that it can't find a class
  - Make sure that you've imported the class or its package.
  - Unset the `CLASSPATH` environment variable, if it's set.
  - Make sure that you're spelling the class name exactly the same as it's declared.
  - Also, some programmers use different names for the class name from the `.java` filename. Make sure that you're using the class name and not the filename.
- Problem: The interpreter says that it can't find one of my classes.
  - Make sure that you specified the class name, not the class file name, to the interpreter.
  - Unset the `CLASSPATH` environment variable, if it's set.
  - If your classes are in packages, make sure that they appear in the correct sub-directory.
  - Make sure that you're invoking the interpreter from the directory in which the `.class` file is located.
- Generally, don't compare floating-point numbers using ==. Remember that floats are only approximations of the real thing, so therefore, the greater than & less than operators (> & <) are often more appropriate when conditional logic is performed on floating-point numbers.
- Don't overuse the negation operator `!` as it can be confusing & error-prone.

4 Testing

Testing is the systematic process of analysing a system or system component to detect the differences between specified (required) behaviour & observed (existing) behaviour. Testing attempts to show that the implementation of a system is inconsistent with the desired functionality. The goal is to design tests that exercise defects in the system and to reveal problems. We cannot test everything in a large system, so there are always trade-offs required with budget & time constraints.

4.1 Levels of Testing

- **Unit testing** involves testing individual classes & mechanisms.
- **Integration testing** involves testing groups of classes or components and the interfaces between them.
- **System testing** involves integration testing the system as a whole to check it meets the requirements.

4.2 Test-Driven Development (TDD)

Test-Driven Development (TDD) is a software development process that relies on the repetition of a very short development cycle. The general rationale behind this methodology is "first write the test, then the code" such that the tests *drive* the development of your code. TDD tries to find faults in participating objects and/or sub-systems with respect to the use cases from the use case model.

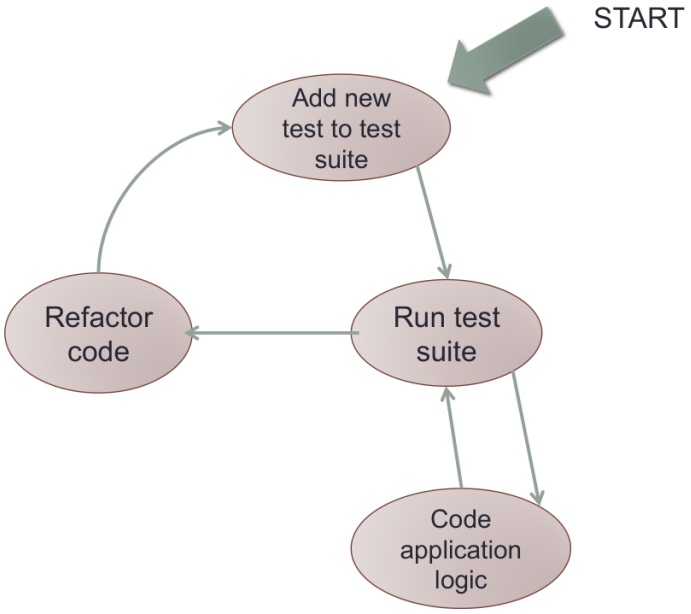

Figure 2: The TDD Cycle

Any test will initially fail; We then write the minimum amount of code to make our test pass. We refactor our application and test the code before moving onto the next test. We build a test suite as our implementation progresses.

4.2.1 Test Cases

A **test component** is a part of the system that can be isolated for testing. This could be an object, a group of objects, or one or more subsystems. **Unit testing** finds differences between a specification of an object and its realisation as a component. **JUnit** is a unit testing framework for test-driven development in Java, which is available in Eclipse out-of-the-box.

A **test case** is a set of inputs & expected results that exercises a test component with the purpose of causing failures & detecting faults. **Black box tests** focus on the input/output behaviour of the component, i.e. the functionality, not the internal aspects. **White box tests** focus on the internal structure & dynamics of the component.

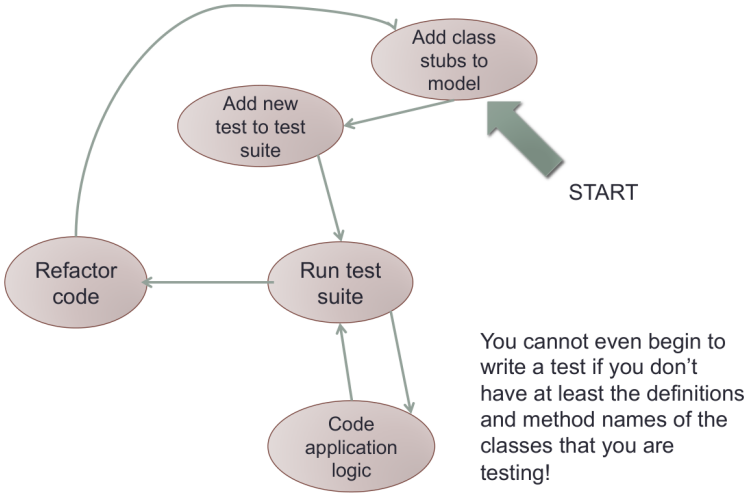

Figure 3: The TDD Cycle in Object-Oriented Development

4.2.2 Assert Methods

```
assertTrue(boolean test)
assertTrue(String message, boolean test)

assertFalse(boolean test)
assertFalse(String message, boolean test)

assertEquals(Object expected, Object actual)
assertEquals(String message, Object expected, Object actual)

assertSame(Object expected, Object actual)
assertSame(String message, Object expected, Object actual)

assertNotSame(Object expected, Object actual)
assertNotSame(String message, Object expected, Object actual)

assertNull(Object object)
assertNull(String message, Object object)

assertNotNull(Object object)
assertNotNull(String message, Object object)

fail()
fail(String message)
```

5 Throwing & Handling Exceptions

Alternatives to exception handling include using no error handling or just exiting the application on error.

5.1 Basics of Java Exception Handling

Code that could generate errors are put in `try` blocks, and the code for handling the error are enclosed in a subsequent `catch` block. The `finally` block always executes with or without an error. The `throws` keyword is used to specify the exceptions which a method might throw if a problem occurs. An exception is thrown when a method detects an error, the exception handler processes the error. The error is thus considered **caught** & **handled** in this model. Control moves from the `try` blocks to the `catch` block(s).

```
try {
    // statements that may throw an exception
}
catch (ExceptionType exceptionReference) {
    // statemetns to process the exception
}
// a `try` is followed by any number of `catch` blocks
catch (ExceptionType exceptionReference) {
    // statemetns to process the exception
}
```

Listing 1: `try` Blocks

5.2 Throwing & Catching Exceptions

The `throw` statement is used to indicate that an exception has occurred. The operand of the `throw` statement is any class that is derived from the class `Throwable`. The sub-classes of `Throwable` are `Exception` (used for problems that should be caught) & `Error` (used for serious problems that should not be caught).

An exception is caught by a **handler**. The program will terminate if there is no appropriate handler for an exception. A single `catch` statement can handle multiple exceptions.

The `throws` clause lists the exceptions thrown by a method.

```
int functionName (paremeterList) throws ExceptionType1, ExceptionType2 {
    // method body
}
```

Listing 2: `throws` Clause

`RunTimeExceptions` occur during the execution of a program, e.g. `ArrayIndexOutOfBoundsException` & `NullPointerException`.

5.3 Checked & Unchecked Exceptions

Unchecked Exceptions (or runtime exceptions) represent unrecoverable errors that occur during the execution of a program. These include sub-classes of `RuntimeException` such as `NullPointerException`. (A full list of runtime exceptions can be found here: https://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeException.html) It is not necessary to add the `throws` declaration for this type of exceptions and it is not necessary to handle them.

Checked Exceptions are caught at compile time. All exceptions other than sub-classes of `RuntimeException` are checked exceptions. An example of a checked exception is `FileNotFoundException`. Checked exceptions are predictable & recoverable. They must be handled in the method body, either with a `try/catch` statement or by re-throwing. It is necessary to add the `throws` declaration for this type of exceptions

5.4 Finalizers

We can throw an exception if a constructor causes an error. The `finalize` method is called the first time that garbage collection is attempted on an object. It will be called a maximum of one times (regardless of whether or not the garbage collection was successful), but won't necessarily ever be called. However, since this won't necessarily be called, it's better not to use it when a resource needs to be released. Instead, one should use the `finally` statement after a `try/catch` block. This always executes, and can be used to release resources.

5.5 Exception Methods

The `printStackTrace()` method of an exception is used to print the method call stack. The `getMessage()` method retrieves the `informationString` from an exception.

6 Unit Testing in Java

We should test single units of functionality in isolation. Unit tests are not integration tests. There should be multiple tests for a single piece of logic, one for each scenario. Each test will cover a single scenario and a single piece of logic.

6.1 Unit Testing Techniques

6.1.1 Equivalence Testing

In **equivalence testing**, possible inputs are partitioned into equivalence classes and a test case is selected for each class. This minimises the number of test cases. Systems usually behave in similar ways for all members of a class.

6.1.2 Boundary Testing

Boundary testing is a special case of equivalence testing that focuses on the conditions at the boundary of the equivalence classes. Instead of selecting any element in the equivalence class, boundary testing requires that the elements be selected from the "edges" of the equivalence class. The assumption is that developers often overlook special cases at the boundary of the equivalence classes.

6.1.3 Path Testing

Path testing exercises all the possible paths through the code at least once, which should cause most faults to trigger failures. This requires knowledge of the source code & data structures.

Code coverage is a measure of how many lines of your code are executed by automated tests. It is impractical to achieve 100% code coverage for large projects.

6.2 Testing Code with Dependencies

Often, the application logic that we want to test will have some dependencies on external services or components. In Unit Testing, we want to isolate our component under test from any dependencies (otherwise, we would be doing **integration testing**). This is problematic as our application logic won't work without its dependencies. The solution to this problem is to create a **stub** to simulate the functionality of this external component.

6.3 TDD Guidelines

- Test the expected outcomes of an example.
- Think about examples & outcomes, not code or how it should work in detail.
- Don't pre-judge design – let your tests drive it.
- Write the minimum code to get your tests to pass.
- Each test should validate one single piece of logic.

7 Strings

Strings allow us to store alphanumeric characters within a variable. In Java, you do not use the `'\0'` (null) character to indicate the end of a `String` object, as you would in C or C++. The `String` object itself keeps track of the number of characters it contains. The `String` `length()` method is the number of characters that a `String` object contains.

- `int length()` returns the length of the `String`.
- `char charAt(int index)` returns the `char` value at the specified index.
- `boolean isBlank()` returns `true` if the `String` is empty or contains only whitespace codepoints, otherwise it returns `false`.
- `boolean startsWith(String prefix)` tests if the `String` starts with the specified prefix.
- `boolean matches(string regex)` tells whether or not the `String` matches the given regular expression.

7.1 Creating Strings

```
1 String myString = "Hello world!";
2 String myString = new String();
3 char [] ct326Array = { 'C', 'T', '3', '2', '6' };
4 String ct326String = new String(ct326Array);
```

```

String fs = tring.format("The value of the float variable is %f, while the value of the integer
↪  variable is %d, and the string is %s", floatVar, intVar, stringVar);
```

7.2 Concatenating Strings

The + operator lets your program concatenate one String's contents onto another. When you concatenate a numeric value, e.g. of type `int` or `float`, Java automatically calls a method called `toString()` on the numeric value to convert the value into a character String. To simplify the output of class-member variables, you can write a `toString()` method for the classes you create; In the same way as for built-in types, the `toString` method will be called automatically for your class. The `String concat(String str)` method of the `String` class concatenates the specified String to the end of this String.

7.3 Comparing Strings

The `int compareTo(String anotherString)` `String` method compares two `Strings` lexicographically.
The `boolean equalsIgnoreCase(String anotherString)` `String` method compares this `String` to another `String`, ignoring letter case.

7.4 Manipulating Strings

- `String substring(int beginIndex, int endIndex)` returns a `String` that is a sub-String of the `String` on which the method was called.
- `String [] split(String regex)` splits the `String` around matches of the given regular expression
- `String strip()` returns this `String` but with all leading & trailing whitespace removed.
- `String replace(char oldChar, char newChar)` returns a `String` resulting from replacing all of `oldChar` in this `String` with `newChar`.

7.5 String Handling

`String` objects are **immutable** – They cannot be changed once they've been created. The `java.lang` package provides a different class, `StringBuffer`, which you can use to create & manipulate character data on the fly.

8 Nested Classes

In Java, it is possible to define a class as a member of another class. Such a class is called a **nested class** and is of the following form:

```
class EnclosingClass {
    ...
    class ANestedClass {
        ...
    }
}
```

Nested classes are used to reflect & to enforce the relationship between two classes. You should define a class within another class when the nested class makes sense only in the context of its enclosing class or when it relies on the enclosing class for its function. For example, a text cursor might only make sense in the context of a text component.

As a member of its enclosing class, a nested class has a special privilege: it has unlimited access to its enclosing class's members, even if they are declared `private`.

8.1 Static & Inner Classes

Like other class members, a nested class can be declared `static`:

- A static nested class is simply called a **static nested class**.
- A non-static nested class is just called an **inner class**

```
class EnclosingClass {
    ...
    static class StaticNestedClass {
        ...
    }
    class InnerClass {
        ...
    }
}
```

As with static methods & variables (which we call class methods & variables), a static nested class is associated with its enclosing class. Like class methods, a static nested class cannot refer directly to instance variables or methods defined in its enclosing class; It can only use them through an object reference. As with instance methods & variables, an inner class is associated with an instance of its enclosing class and has direct access to that object's instance variables & methods.

The interesting feature of the relationship between the inner class & the enclosing class is *not* that the inner class is syntactically defined within the enclosing class. Rather, it's that an instance of the inner class can only exist within an instance of the enclosing class, and that it has direct access to the instance variables & methods of its enclosing instance. You may encounter nested classes of both static & inner types in the Java platform API and be required to use them. However, most nested classes that you write will probably be **inner classes**.

8.2 Using Anonymous Inner Classes

Using a Tokenizer, partition a `String` into individual sub-strings using a **delimiter**. To do this, Java offers `java.util.StringTokenizer`

```
// Fig. 10.20: TokenTest.java
// Testing the StringTokenizer class of the java.util package

// Java core packages
import java.util.*;
import java.awt.*;
import java.awt.event.*;

// Java extension packages
import javax.swing.*;

public class TokenTest extends JFrame {
    private JLabel promptLabel;
    private JTextField inputField;
    private JTextArea outputArea;

    // set up GUI and event handling
    public TokenTest()
    {
        super( "Testing Class StringTokenizer" );

```

```
22          Container container = getContentPane();
23          container.setLayout( new FlowLayout() );
24
25          promptLabel = new JLabel( "Enter a sentence and press Enter" );
26          container.add( promptLabel );
27
28          inputField = new JTextField( 20 ); // inputField contains String to be parsed by
            ↪ StringTokenizer
29
30          inputField.addActionListener(
31              // anonymous inner class
32              new ActionListener() {
33                  // handle text field event
34                  public coid actionPerformed(ActionEvent event)
35                  {
36                      String stringToTokenize = event.getActionCommand();
37                      StringTokenizer tokens = new StringTokenizer( stringToTokenize ); // Use
                        ↪ StringTokenizer to parse String stringToTokenize with default delimiters
                        ↪ "\n\t\r"
38
39                      outputArea.setText( "Number of elements: " + tokens.countTokens() + "\nThe
                        ↪ tokens are:\n" ); // count number of tokens
40
41                      // apend next token to outputArea as long as tokens exist
42                      while ( tokens.hasMoreTokens() )
43                          outputArea.append( tokens.nextToken() + "\n" );
44                  }
45              }   // end anonymous inner class
46          );  // end call to addActionListener
47
48          container.add( inputField );
49
50          outputArea = new JTextArea( 10, 20 );
51          outputArea.setEditable( false );
52          container.add( new JScrollPane( outputArea ) );
53
54          setSize( 275, 260 );  // set the window size
55          show();               // show the window
56      }
57
58      // execute application
59      public static void main( String args[] )
60      {
61          TokenTest application = new TokenTest();
62          application.addWindowListener(
63              // anonymous inner class
64              new WindowAdapter() {
65                  // handle event when user closes window
66                  public void windowClosing( WindowEvent windowEvent )
67                  {
68                      System.exit( 0 );
69                  }
70              }  // end anonymous inner class
```

```
71          ); // end call to addWindowListener
72      }  // end method main
73  }  // end class TokenTest
```

Listing 3: `TokenTest.java`

9 Enums

An **enumerated type** is a type whose legal values consist of a fixed set of constraints. Common examples include the cardinal directions, the days of the week, etc. In Java, you define an enumerated type by using the `enum` keyword. For example, you would specify a days-of-the-week enumerated type as:

```
1  enum Days { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY };
```

Note that by convention, the names of an enumerated type's values are spelled in all uppercase letters.] Enumerated types should be used any time that you need to represent a fixed set of constraints.

Enumerated types in Java are much more powerful than their counterparts in other languages, such as C, which are just glorified integers. In Java, the `enum` declaration defines a class called an **enum type**. These are the most important properties of enum types:

- Printed values are informative.
- They are type-safe and exist in their own namespace.
- You can `switch` on an enumeration constant.
- They have a static `values()` method that returns an array containing all of the values of the enum type in the order they are declared. This method is commonly used in combination with the for-each construct to iterate over the values of an enumerated type.
- You can provide methods & fields, implement interfaces, and more.

Note that the constructor for an `enum` type is implicitly private. If you attempt to create a public constructor for an enum type, the compiler displays an error message.

9.1 Example

In the following example, `Planet` is an enumerated type that represents the planets in our solar system. A `Planet` has constant mass & radius properties. Each enum constant is declared with values for the mass & radius parameters that are passed to the constructor when it is created.

```
1  public enum Planet {
2      MERCURY (3.303e+23, 2.4397e6),
3      VENUS   (4.869e+24, 6.0518e6),
4      EARTH   (5.976e+24, 6.37814e6),
5      MARS    (6.421e+23, 3.3972e6),
6      JUPITER (1.9e+27,   7.1492e7),
7      SATURN  (5.688e+26, 6.0268e7),
8      URANUS  (8.686e+25, 2.5559e7),
9      NEPTUNE (1.024e+26, 2.4746e7),
10     PLUTO   (1.27e+22,  1.137e6);
11
12     private final double mass;   //in kilograms
13     private final double radius; //in meters
```

```
14
15      Planet(double mass, double radius) {
16          this.mass = mass;
17          this.radius = radius;
18      }
19
20      public double mass()   { return mass; }
21      public double radius() { return radius; }
22      public static final double G = 6.67300E-11; //universal gravitational constant  (m3 kg-1 s-2)
23
24      public double surfaceGravity() {
25          return G * mass / (radius * radius);
26      }
27
28      public double surfaceWeight(double otherMass) {
29          return otherMass * surfaceGravity();
30      }
31  }
```

Listing 4: `Planet` Enumerated Type Example

In addition to its properties, `Planet` has methods that allow you to retrieve the surface gravity & weight of an object on each planet. Here is a sample program that takes you weight on Earth in any unit and calculates & prints your weight on all of the planets in the same unit.

```
public static void main(String[] args) {
    double earthWeight = Double.parseDouble(args[0]);
    double mass = earthWeight/EARTH.surfaceGravity();

    for (Planet p : Planet.values()) {
        System.out.printf("Your weight on %s is %f%n", p, p.surfaceWeight(mass));
    }
```

Listing 5: Sample Program Using the `Planet` Enumerated Type

The output is as follows:

```
$ java Planet 175
    Your weight on MERCURY is 66.107583
    Your weight on VENUS is 158.374842
    Your weight on EARTH is 175.000000
    Your weight on MARS is 66.279007
    Your weight on JUPITER is 442.847567
    Your weight on SATURN is 186.552719
    Your weight on URANUS is 158.397260
    Your weight on NEPTUNE is 199.207413
    Your weight on PLUTO is 11.703031
```

The limitation of `enum` types is that, although `enum` types are classes, you cannot define a hierarchy of enums. In other words, it's not possible for one enum type to extend another enum type.

10 Packages

A **package** is a collection of related types (such as classes, interfaces, enums, & annotations) providing access protection & namespace management. Programmers bundle groups of related types into packages to make them easier to find &

use, to avoid naming conflicts, and to control access. The types that are part of the Java platform are members of various packages that bundle classes by function, such as fundamental classes in `java.lang`, classes for reading & writing in `java.io`, etc. Related types should be bundled into a package for several reasons:

- Packages allow programmers to easily determine that the types which they contain are related.
- The names of the types in the package won't conflict with the names of types in other packages because the package creates a new namespace.
- One can allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package (package private access).

10.1 Creating Packages

To create a package, you simply put a type (such as a class, interface, enum, or annotation) in a directory corresponding to the name of the package that you want to create and put a **package statement** at the top of the source file in which the type is defined. You must include a package statement at the top of every source file that defines a class or an interface that is to be the member of your package, and the name of the package must correspond to that of the directory.

10.2 Package Scope

The copse of the package statement is the entire source file, including all the classes, interface, enums, & annotations defined in the file. If there are multiple classes in a single source file, only one may be public, and it must share the name of the source file's base name. Only public package members are accessible from outside the package. If you do not use a package statement, your type will be treated as part of the **default package** which is a package that has no name.

10.3 Naming Packages

With programmers all over the world defining types in Java, it is not unlikely that two programmers will use the same name for two different types, but conflicts can be circumvented via the use of packages. If the types are in different packages, no conflict will occur, as the fully-qualified name of each type includes the package name. This generally works fine, unless two independent programmers use the same name for their packages. This problem is prevented by convention: companies will, by convention, use to their reversed Internet domain name in their package names, e.g. `com.company.package`. Name collisions that occur within a single company need to be handled by convention within that company, perhaps by including the region or the project name after the company name, e.g. `com.company.region.package`.

10.4 Static Imports

The **static import** feature, implemented as `import static <package>` enables you to refer to the static constants from a class without needing to inherit from it.

11 Useful Java Features

11.1 For-Each Loops

The `Iterator` class provides a mechanism to navigate sequentially through a Collection; it is used heavily by the Collections API. The **For-Each loop** can replace the Iterator when simply traversing though a Collection – the compiler generates the looping code required with generic types so that no additional casting is required.

```
1 ArrayList<Integer> list = new ArrayList<Integer>();
2
3 // looping though an ArrayList using the Iterator class
4 for (Iterator i = list.iterator(); i.hasNext();) {
5     Integer value = (Integer) i.next();
6     // do some stuff
```

```
}

// looping though an ArrayList using a For-Each loop
for (Integer i : list) {
    // do some stuff
}
```

11.2 Formatted Input & Output

Developers have the option of using `printf`-type functionality to generate formatted output. Most of the common C `printf` formatters are available, and some classes like `Data` & `BigInteger` also have formatting rules. Although the standard UNIX newline `'\n'` character is accepted, the Java `%n` is recommended for cross-platform support. See the `java.util.Formatter` class for more information.

```
System.out.printf("name count%\n");
System.out.printf("%s %5d%\n", user, total);
```

The Scanner API provides basic input functionality for reading data from the system console or any data stream. If you need to process more complex input, then there are also pattern-matching algorithms available from the `java.util.Formatter` class.

```
Scanner s = new Scanner(System.in);
String param = s.next();
int value = s.nextInt();
s.close();
```

11.3 Varargs

Varargs can be used to indicate that a flexible number of arguments is required for a given method, which allows us to pass an unspecified number of arguments to the method. The built-in `printf()` function makes use of varargs.

```
void argtest(Object... args) {
    for (int i = 0; i < args.length; i++) {

    }
}

argtest("test", "data");
```

12 Lambda Expressions

A **lambda expression** is a short block of code that takes parameters and returns a value; they are similar to methods but lambda expressions do not require a name. Lambda expressions can be implemented in the body of a method.

```
parameter -> expression
(parameter1, parameter2) -> expression
```

For example, to print out all the elements of an Iterable data structure:

```
elements.forEach(e -> System.out.println(e));
```

13 IO Streams

To take in information, a program opens a **stream** on an information source (such as a file, memory, or a socket) and reads the information **serially**. No matter where the information is coming from or going to and what type of data is being read or written, the algorithms for reading & writing data are usually the same:

- **Reading:**
    1. Open a stream.
    2. While there is more information: read information.
    3. Close the stream.
- **Writing**
    1. Open a stream.
    2. While there is more information: write information.
    3. Close the stream.

The `java.io` package contains a collection of stream classes that support reading & writing streams. The stream classes are divided into two class hierarchies based off their data: **character streams** or **byte streams**. However, it is often more convenient to group the classes by their purpose rather than the type of data that they handle, so we often cross-group the streams by whether they read data from & write data to **data sinks** or process the information as it's being read or written.

13.1 Data Sink Streams

Data sink streams read from or write to specialised **data sinks** such as Strings, files, or pipes. Typically, for each reader or input stream intended to read from a specific kind of input source, `java.io` contains a parallel writer or output stream that can create it. Note that both the character stream group and the byte stream group contain parallel pairs of classes that operate on the same data sinks.

13.2 Processing Streams

Processing streams perform some kind of operations such as buffering or data conversion as they read & write. Like the data sink streams, `java.io` often contains pairs of streams: one that performs a particular operation during reading and one that performs the same operation or reverses it during writing. Note that in many cases, `java.io` contains character streams & byte streams that perform the same processing but for different data types.

13.3 Byte Streams

Programs should use **byte streams**, descendants of `InputStream` & `OutputStream` to read & write 8-bit bytes. `InputStream` & `OutputStream` provide the API and some implementation for input streams (streams that read 8-bit bytes and output streams (streams that write 8-bit bytes). These streams are typically used to read & write binary data such as images and sounds.

Subclasses of `InputStream` & `OutputStream` provide specialised I/O. Those that read from or write to data sinks are shown in grey in the following figures, while those that perform some sort of processing are shown in white:

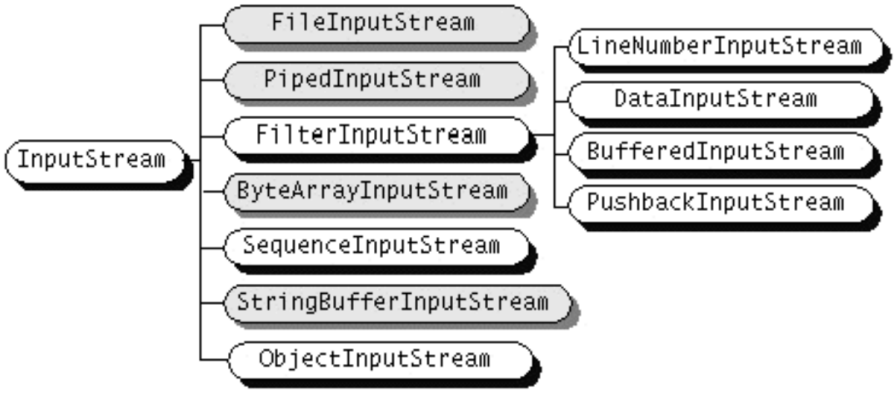

Figure 4: Data Sink & Processing Subclasses of `InputStream`

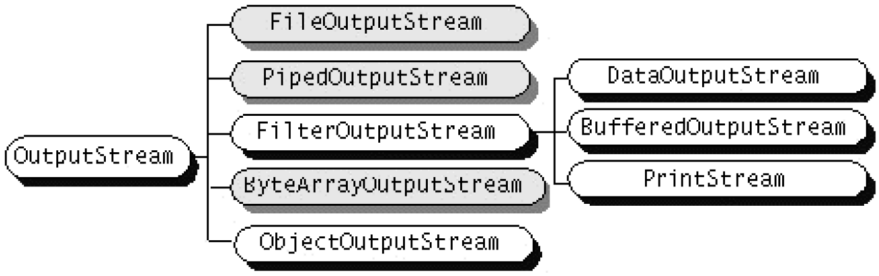

Figure 5: Data Sink & Processing Subclasses of `OutputStream`

13.4 Character Streams

`Reader` & `Writer` are the abstract superclasses for character streams in `java.io.*`. `Reader` provides the API & partial implementation for readers, i.e. streams that read 16-bit Unicode characters and `Writer` provides the API & partial implementation for writers, i.e. streams that write 16-bit characters. Subclasses of `Reader` & `Writer` implement specialised streams.

The following figures show the readers or writers that read/write to/from data sinks in grey, and those that performs some sort of processing are shown in white:

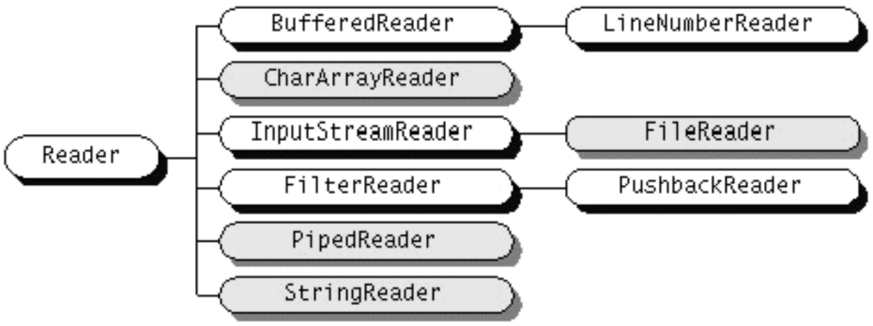

Figure 6: Data Sink & Processing Subclasses of `Reader`

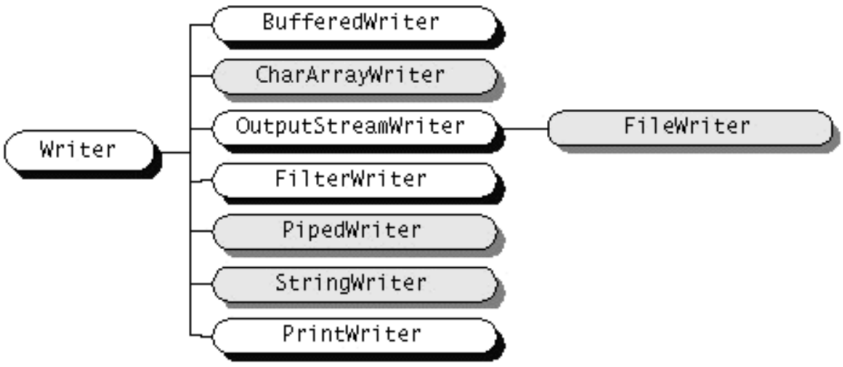

Figure 7: Data Sink & Processing Subclasses of `Writer`

Most programs should use readers & writers to read & write information because they can handle any character in the Unicode character set whereas byte streams are limited to ISO-Latin-1 8-bit bytes.

13.5 `Scanner`

The **`Scanner`** class is in the `java.util` package but can be passed an `InputStream` as a constructor parameter. Scanners are often used for reading from the console, i.e. the `System.in` InputStream. They are used for working with inputs of formatted data consisting of primitive types & Strings. Scanners translate the input to tokens based on their data type and using a delimiter pattern. They also have a collection of `next()` methods for different data types.

13.6 Formatting

`PrintWriter` is a processing character stream that includes common methods for formatting such as `print`, `println`, & `format`. The `format` method formats multiple arguments based on a **format String** that includes format specifiers, which can be used similar to tokens in Scanners. See the `java.util.Formatter` class for documentation on format String syntax.

14 Object Serialization

The streams `java.io.ObjectInputStream` & `java.io.ObjectOutputStream` are special in that they can read and write actual objects. The key to writing an object is to represent its state in a serialized form sufficient to reconstruct the object as it is read. Thus reading & writing objects is a process called **object serialization**. Object serialization can be useful in a lot of application domains.

Uses of serialization include:

- **Remote Method Invocation (RMI)**: communication between objects via sockets, e.g. to pass various objects back & forth between the client & server.
- **Lightweight persistence**: the archival of an object for use in a later invocation of the same program

Reconstructing an object from a stream requires that the object first be written to a stream; writing objects to a stream is a straightforward process. The following code sample gets the current time in milliseconds by constructing a `Date` object and then serializes the object.

```
1 FileOutputStream out = new FileOutputStream("theTime");
2 ObjectOutputStream s = new ObjectOutputStream(out);
3 s.writeObject("Today");
4 s.writeObject(new Date());
5 s.flush();
```

Listing 6: Object Serialization Example

The above code constructs an `ObjectOutputStream` on a `FileOutputStream`, thereby serializing the object to a file named `theTime`. Next, the String `Today` & a `Date` object are written to the stream with the `writeObject` method of `ObjectOutputStream`.

14.1 Object Serialization with Related Objects

if an object refers to other objects, then all of the objects that are reachable from the first must be written at the same time so as to maintain the relationships between them. Thus, the `writeObject` method serializes the specified object, traverses its references to other objects recursively, and writes them all. The `writeObject` method throws a `NotSerializableException` if it's given an object that is not serializable. An object is serializable only if its class implements the `Serializable` interface.

14.2 Reconstructing Serialized Objects

Once you've written objects & primitive data types to a stream, you'll likely want to read them out again and reconstruct the objects. The below code reads in the `String` & the `Date` object that was written to the file named `theTime` in the previous example:

```
1 FileInputStream in = new FileInputStream("theTime");
2 ObjectInputStream s = new ObjectInputStream(in);
3 String today = (String) s.readObject();
4 Date date = (Date) s.readObject();
```

Listing 7: Object Reconstruction Example

Note that there is no standard file extension for files that store serialized objects.

Like `ObjectOutputStream`, `ObjectInputStream` must be constructed on another stream. In this example, the objects were archived in a file, so the code constructs an `ObjectInputStream` on a `FileInputStream`. Next, the code uses the `readObject` of the `ObjectInputStream` method to read the `String` & `Date` objects from the file. The objects must be read from the stream in the same order in which they were written. Note that the return value from `readObject` is an object that is cast to and assigned to a specific type. The `readObject` method deserializes the next object in the stream and traverses its references to other objects recursively to deserialize all objects that are reachable from it. In this way, it maintains the relationships between objects. The methods in `DataInput` parallel those defined in `DataOutput` for writing primitive data types.

14.3 Serializing Classes

AN object is serializable only its class implements the `Serializable` interface. Thus, if you want to serialize an instance of one of your classes, the class must implement the `Serializable` interface, i.e. it must have the ***`implements`*** `Serializable` clause. `Serializable` is an empty interface, i.e. it does not contain any method declarations; its purpose is simply to identify classes whose objects are serializable. Its complete definition is as follows:

```
1 package java.io;
2
3 public interface Serializable {
4     // there's nothing in here!
5 };
```

Listing 8: Definition of the `Serializable` Interface

You don't have to write any methods to make a class serializable. The serialization of instances of a serializable class are (by default) handled by the `defaultWriteObject` method of the `ObjectOutputStream`.

All instance variables of a class must be serialized for it to be serializable, including referenced objects & those within referenced data structures In Java, all primitive type variables are serializable by default. We can ignore instance variables in the process by declaring them as `transient`

14.4 The `defaultWriteObject` Method

The `defaultWriteObject` method automatically writes out everything required to reconstruct an instance of the class, including the class of the object, the class signature, and values of non-transient & non-static members, including members that refer to other objects. For many classes, the default behaviour is fine. However, default serialization can be slow, and a class might want more explicit control over the serialization.

14.5 Customising Serialization

You can customise serialization for your classes by providing two methods: `writeObject` & `readObject`. The `writeObject` method controls what information is saved and is typically used to append additional information to the stream. The `readObject` method either reads the information written by the corresponding `writeObject` method or can be used to update the state of the object after it has been restored.

The `writeObject` method must be declared exactly as follows. Note that it should call the stream's `defaultWriteObject` as the first thing it does to perform default serialization (any special arrangements can be handled afterwards):

```
ptivate void writeObject(ObjectOutputStream s) throws IOException {
    s.defaultWriteObject();
    // customised serialization code
}
```

Listing 9: Declaration of the `writeObject` Method

The `readObject` method must read in everything written by `writeObject` in the same order in which it was written. The following `readObject` method corresponds to the above `writeObject` method:

```
private void readObject(ObjectInputStream s) throws IOException {
    s.defaultReadObject();
    // customised deserialization code
    // followed by code to update the object if necessary
}
```

Listing 10: Declaration of the `readObject` Method

The `readObject` method can also perform calculations or update the state of the object in some way. The `writeObject` & `readObject` methods are responsible for serializing only the immediate class. Any serialization required for the superclasses is handled automatically. However, a class that needs to explicitly co-ordinate with its superclasses to serialize itself can do so by implementing the `Externalizable` interface.

14.5.1 The `Externalizable` Interface

For complete, explicit control over the serialization process, a class must implement the `Externalizable` interface. For `Externalizable` objects, only the identify of the object's class is automatically saved by the stream. The class is responsible for reading & writing its contents, and it must co-ordinate with its superclasses to do so

15 Random File Access

The input & output streams that we have been dealing with so far have been **sequential access streams**, i.e. streams whose contents must be read or written sequentially. Sequential access files are a hangover from sequential media such as magnetic tape, but are still very useful today. **Random access files** permit non-sequential (or random) access to the contents of a file.

15.1 Random File Access Example Use Case

Consider the "zip" archive format. Zip archives contain files and are typically compressed to save space. Zip archives also contain a `dir-entry` at the end of the file that indicates where the various files contained within the zip archive begin:

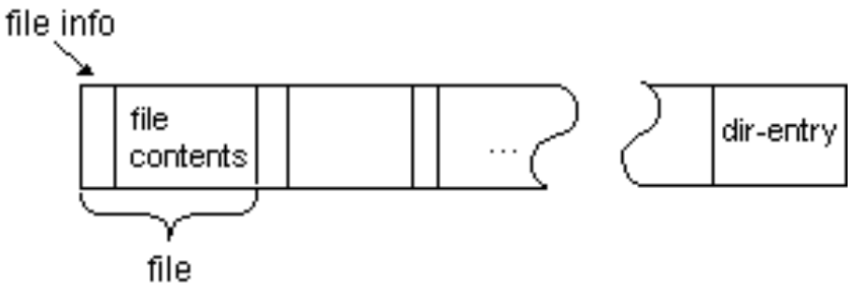

Figure 8: Zip Archive Format

Suppose that you want to extract a specific file from a zip archive. If you use a sequential access stream, you have to do the following:

1. Open the zip archive.
2. Search through the zip archive until you have located the file that you want to extract.
3. Extract the file.
4. Close the zip archive.

Using this algorithm, you'd have to read half the zip archive on average before finding the file that you want to extract. You can extract the same file from the zip archive more efficiently using the **seek** feature of a random access file:

1. Open the zip archive.
2. Seek to the `dir-entry` and locate the entry for the file you want to extract from the zip archive.
3. Seek backwards within the zip archive to the position of the file to extract.
4. Extract the file
5. Close the zip archive.

This algorithm is more efficient because you only read the `dir-entry` and the file that you want to extract.

15.2 The `RandomAccessFile` Class

The `RandomAccessFile` class in the `java.io` package implements a random access file. Unlike the input & output stream classes in `java.io`, `RandomAccessFile` is used for both reading & writing files; it implements both the `DataInput` & `DataOutput` interfaces and therefore can be used for both reading & writing.

We create `RandomAccessFile` objects with different arguments depending on whether you intend to read or write. `RandomAccessFile` is similar to `FileInputStream` & `FileOutputStream` in that you specify a file on the native file system to open, either with a filename or a `File` object. When you create `RandomAccessFile`, you must indicate whether you will be reading the file or also writing to it. The following code snippet creates a `RandomAccessFile` to read the file named `farrago.txt`:

```
1 new RandomAccessFile("farrago.txt", "r");
```

The following code snippet opens the same file for both reading & writing:

```
new RandomAccessFile("farrago.txt", "rw");
```

After the file has been opened, you can use the common read & write method to perform I/O on the file. The `RandomAccessFile` class supports the notion of a file pointer, which indicates the current location in the file. When the file is first created, the file pointer is `0`, indicating the start of the file. Calls to the read & write methods adjust the file pointer by the number of bytes read or written.

In addition to the normal file I/O methods that implicitly move the file pointer when the operation occurs, `RandomAccessFile` also contains three methods for explicitly manipulating the file pointer:

- `skipBytes()`: moves the file pointer forward the specified number of bytes.
- `seek()`: positions the file pointer just before the specified byte.
- `getFilePointer()`: returns the current byte location of the file pointer.

`RandomAccessFile` is somewhat disconnected from the input & output streams in `java.io`: it doesn't inherit from either `InputStream` or `OutputStream`. This has some disadvantages in that you can't apply the same filters to `RandomAccessFiles` that you can However, `RandomAccessFile` does implement the `DataInput` & `DataOutput` interfaces: if you design a filter that works for either `DataInput` or `DataOutput`, it will work on any `RandomAccessFile`.

16 Type-Wrapper Classes

Collections manipulate & store **Objects**, not primitive types. Each Java primitive type has a corresponding **type-wrapper class** in `java.lang` that enables primitives to be manipulated as Objects, e.g. `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short`. Numeric type-wrapper classes extend the `Number` class. Methods related to primitive types are contained in the type-wrapper classes, e.g. `parseInt` is located in the class `Integer`.

16.1 Autoboxing & Auto-Unboxing

A **boxing conversion** converts a value of a primitive type to an Object of the corresponding type-wrapper class. An **unboxing conversion** converts an object of a type-wrapper class to a value of the corresponding primitive type. When these conversions are done automatically, they are referred to as **autoboxing** & **auto-unboxing**. For example:

```
Double[] myDoubles = new Double[10];
myDoubles[0] = 22.7;                    // autoboxed to a Double Object
double firstDoubleValue = myDoubles[0]; // auto-unboxed to a double primitive
```

17 Collections

A **Collection** (sometimes called a container) is an object that groups multiple elements into a single unit. Collections are used to store, retrieve, & manipulate data, and to transmit data from one method to another. Collections typically represent data items that form a natural group.

17.1 Interfaces

The core Collections interfaces are the interfaces used to manipulate Collections, and to pass them from one method to another. The basic purpose of these interfaces is to allow Collections to be manipulated independently of the details of their representations. The core Collections interfaces are the main foundation of the Collections framework.

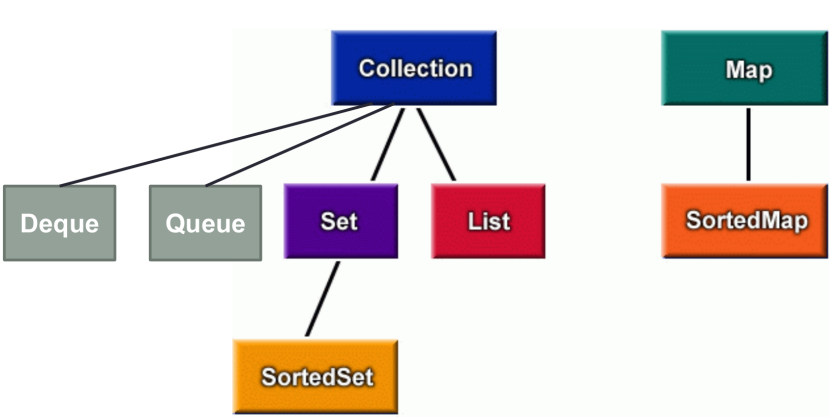

Figure 9: Core Collections Interfaces

The core collection interfaces form a hierarchy: a `Set` is a special kind of `Collection`, a `SortedSet` is a special kind of `Set`, etc. Note that the hierarchy consists of two distinct trees: a `Map` is not a true `Collection`.

To keep the number of core collection interfaces manageable, the JDK doesn't provide separate interfaces for each variant of each collection type. Among the possible variants are immutable, fixed-size, & append-only. Instead, the modification operations in each interface are designated optional, i.e. a given implementation does not have to support all of these operations. If an unsupported operation is invoked, the Collection will throw an `UnsupportedOperationException`. Implementations are responsible for documenting which of the optional operations they support. All of the JDK's general-purpose implementations support all of the optional operations.

17.1.1 The `Collection` Interface

The `Collection` interface is the root of the Collection hierarchy. A `Collection` represents a group of objects, which are known as *elements*. Some `Collection` implementations allow duplicate elements, while others do not; some `Collection` implementations are ordered, and others are not. The JDK doesn't provide any direct implementations of this interface: it provides only implementations of the more specific sub-interfaces like `Set` & `List`.

The `Collection` interface is the lowest common denominator that all Collections implement. It is used to pass Collections around and to manipulate them when maximum generality is desired.

```
public interface Collection {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);        // Optional
    boolean remove(Object element);     // Optional
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);       // Optional
    boolean removeAll(Collection c);    // Optional
    boolean retainAll(Collection c);    // Optional
    void clear();                       // Optional

    // Array Operations
    Object[] toArray();
```

```
19      Object[] toArray(T[] a);
20  }
```

Listing 11: The `Collection` Interface

18 Graphical User Interfaces

Graphical User Interfaces (GUIs) give a program a distinctive "look & feel", provide users with a basic level of familiarity, and are built from GUI components such as controls, widgets, etc. The user interacts with GUI components via the mouse, keyboard, etc.

18.1 Swing

Swing GUI components are from the package `javax.swing`. They are lightweight components written entirely in Java. The components originate from AWT (`java.awt`). A common superclass structure of many of the Swing components is: `java.lang.Object` → `java.awt.Component` → `java.awt.Container` → `javax.swing.JComponent`.

Some basic GUI components include:

- `JLabel`: an area where uneditable text or icons can be displayed.
- `JTextField`: an area in which the user inputs data form the keyboard. Can also display information.
- `JButton`: an area that triggers an event when clicked.
- `JCheckBox`: a GUI component that is either selected or not.
- `JComboBox`: a drop-down list of items from which the user can make a selection by clicking an item in the list or by typing into the box.
- `JPanel`: a container in which components can be placed.

19 INSERT MISSING GUI SHIT HERE

20 Multithreaded Programming

A **process** is an instance of a program being executed. A **multitasking** environment allows multiple process to be ran concurrently. (In reality, only one process is running at a given time on a single CPU. Time slicing between processes makes it appear as if they are all running at the same time.)

A **thread** is a "lightweight process", i.e. it does not have all the overhead that a process has. Both processes & threads have their own independent CPU state, i.e. they have their own processor stack, their own instruction pointer, & their own CPU register values. Multiple threads can share the same memory address space, i.e. share the same variables. Processes, on the other hand, generally do not share their address space with other processes. As a rule of thumb, if an application or applet performs a time-consuming task, it should create and use its own thread of execution to perform that task.

20.1 Threads & Java

Java has language-level support for threads and it is therefore easy to create multiple independent threads of execution in Java. Any class that is a subclass of `java.lang.Thread` or that implements the `java.lang.Runnable` interface can be used to create threads.

When a program creates a thread, it can pass the `Thread` class constructor as an object whose statements will be executed. The program can start a thread's execution by using its `start()` method; the `start()` method will in turn call the `Thread` object's `run()` method.

20.1.1 `java.lang.Runnable`

To create a thread, we instantiate the `java.lang.Thread` class. One of `Thread`'s constructors takes objects that implement the `Runnable` interface. The `Runnable` interface contains only a single method:

```
1  public interface Runnable {
2      public void run();
3  }
```

Listing 12: The `Runnable` interface

To create a thread, we create an instance of the `Thread` class. We start the thread by calling the `start()` method of the thread, which in turn invokes the thread's `run()` method. The thread will terminate when the `run()` method terminates.

```
1  class Fred implements Runnable {
2      public void run() {
3          // insert code to be ran here
4      }
5
6      public static void main(String args[]) {
7          Thread t = new Thread(new Fred()) {
8              t.start();
9          }
10     }
11 }
```

Listing 13: Creating & Starting a Thread

20.2 Thread Priority & Daemons

Every thread has a **priority**: threads with higher priority are executed in preference to threads with lower priority. Threads can also be marked as a **daemon**: a background thread that is not expected to exit. New threads initially have their priority set to the priority of the creating thread. The new thread is a daemon thread if & only if the creating thread is a daemon thread

20.3 Threads & the JVM

When the Java Virtual Machine starts up, there is usually a single non-daemon (user) thread initially; this thread calls the `main()` method of an application. The JVM continues to execute threads until either the `exit()` method of the class `Runtime` is called or all non-daemon (user) threads have died.

20.4 Thread Death

Threads can be killed by calling their `stop()` method but this is not a good idea in general. It is preferable to have a thread die naturally by having its `run()` return. This is often done by altering some variable that causes a while loop to terminate in the `run()` method.

20.5 Synchronisation

Multithreaded programming requires special care, and is generally more difficult to debug. We want to prevent multiple threads from altering the state of an object at the same time. Sections of code that should not be executed simultaneously are called **critical sections**. We want to have **mutual exclusion** of concurrent threads in critical sections of code. In Java, this is done by using **synchronised methods** or **synchronised statements**.

20.5.1 The `synchronized` Statement

A ***`synchronized`*** statement attempts to acquire a **lock** for the object or array specified by the `expression`.