Java

Java was developed by Sun Microsystems, under James Gosling in 1991. The goal was to develop a language suitable for applications in embedded systems. This use case failed to manifest, and they instead reoriented Java as a language to develop browser applets.

On arrival Java had the unique feature that it was both compiled and interpreted. It was also platform-independent and portable. Finally it was object oriented.

While with a C program, plain text code is compiled by a compiler and then linked by a linker which produces a different executable for every platform. For Java, plain text source code is compiled and output as byte code, which is directly interpreted by a Java interpreter. Therefore, all that is needed to run Java on a given platform is an interpreter. This gives Java its platform independence.

In Java, two types of programs exist.

- An *application* is a stand-alone program, which has a main method and can be run directly as a program, for example from the command line.
- An *applet* is a program, initially intended for use in webpages which has no main method, and instead has a specific contruction which allows the environment to function. This form is less popular, generally considered outdated, and is not covered in detail in this subject.

The fundamental constructs in Java, and in all object oriented languages are classes, objects and methods.

Below lies a "Hello World" program written in Java.

```
import java.lang.*;

// Display "Hello World"
public class HelloWorld {
```

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

A variety of interesting things are visible in even this small snippet of code.

- The whole program is encased in a class; here `HelloWorld`.
- The `main` function is prefaced by `public static`.
- The `main` function has return type `void`.
- Where in C we might `#include` here we `import`.

Aside these differences, the code is rather similar to a comparable C program.

Three types of comments exist in java.

```
// This is a single line comment, as in C
/* This is a multi-line comment, as in C */
/** This is a documentation comment, to be explored later */
```

To run a Java program that has been saved as plain text, a few conditions must be satisfied.

- The filename must be the same as the name of the class. For example, our "Hello World" program would be `HelloWorld.java`.
- Java build and runtime environment must be installed on the machine. This can be checked with `javac -version` or `java -version`.

Then, one can compile with `javac HelloWorld.java`. If successful, an output file `HelloWorld.class` will be generated. This can be run with `java HelloWorld`.

Arguments passed in at program execution can be accessed in the `args[]` array. The number of arguments can be found through `args.length`.

Key differences between Java and C

- While C is a procedural language, Java is object oriented.
- Java lacks many of the lower level facilities of C; goto, sizeof, pointers, etc. No direct memory management.
- Java has no preprocessor, macros, defines, etc are not available.

Classes and Objects

All programming languages have four fundamental operations

- Calculation
- Selection
- Iteration
- Abstraction

The key focus of this subject is in abstraction, the process of creating constructs and defining interactions between them to solve a problem. The focus of oriented oriented programming is on a specific form of abstraction. In a procedural language like C, abstraction is provided largely through functions. In Java, abstraction is implemented through classes, which are a form of abstract data type.

A class might represent a real world object, a real world concept or a problem space object or concept. It has attributes and methods, which define its characteristics and behaviours. This class can then be used as a data type.

An *object* is an instance of a class. Many objects may be instances of the same class, with differing attributes which define their state. An objects type is the class of which it is an instance.

When designing an application in an object oriented way, one should focus initially on nouns; if a concrete concept or object is required for the application,

then it should be a strong candidate for a class in the program. The properties of this entity should then become attributes; colour, radius, etc. Things that this entity might do should then become methods; move, open, save, etc.

Features of designing in this way include

- Data Abstraction; through creating data types we can simplify the manipulation and storage of information around an entity.
- Encapsulation; by creating classes, we group related properties behind a common interface, often simplifying problems.
- Information hiding
- Delegation
- Inheritance
- Polymorphism

Classes

Defining a Class

The format for a class definition is as follows. The class must be placed in the file `Name.java`. In general classes are named in camelcase with a leading capital (`ClassName`), while attributes and methods are named in conventional camelcase (`methodName`).

```
<visibility> class <Name> {
    <attribute declarations>
        <visibility> <type> <name>
    <method declarations>
        <visibility> <void or type> <name> (<arguments>)
}
```

A simple class example follows.

```java
// in Person.java
public class Person {
    public String firstName;
    public String lastName;

    public String getFullName () {
        return firstName + lastName;
    }
}
```

Attributes included as part of a class are instance variables, differentiated from local variables defined in the namespace of a method. These variables are members of the class.

Using a Class

Once a file with a class has been compiled (with `javac Class.java`), other files in the same directory as the `.class` file can use the type without imports. Creating an object reference can then be done with `Class obj`. Until this object is *instantiated*, it will simply be a `null` reference.

```java
public class Main {
    public static void main(String args[]) {
        Class obja, objb; // obja is a null reference
        obja = new Class(); // now Class instance
        objb = new Class();
        obja = objb; // obja is now a reference to objb
    }
}
```

In the above example, two instances of `Class` are created, and then one is overwritten with another. Because no reference exists to `obja`, it will be garbage collected.

Members of classes are accessed with the member accessor, `.`. For example, `object.attr` or `object.method()`. The attributes of an object can be assigned in this way, i.e. `object.attr = value`.

The main method, while simply a method of a class like any other, is special, as it and it alone can be the entry point of a program.

Instance Variables

In general, initialisation, updating and accessing instance variables is done through defining *Accessor / Mutator* methods (or *Getter / Setter* methods). This offers benefits when we come to information hiding, visibility control and privacy. Many IDEs will include support for automatic generation of getters and setters. This is a rather different paradigm to something like Python or C. A getter performs a similar function to something like a `@property` decorated function in Python.

```
public class Person {
    public String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Notice the use of `this`. `this` always refers to the class instance of which it is used from within. It is more or less equivalent to Python's `self`, though it isn't explicitly provided as an argument like `self`.

Constructors

A constructor is a method that exists for the purpose of instantiating an instance of a class. If a constructor is not defined, a default constructor is called when a `new` instance is created. A class can have more than one constructor, with different arguments through a process known as overloading. A constructor is always public, and cannot have a return value. They always have the same name as the class. A constructor is defined as follows:

```
public class Person {
    public String firstName, lastName;

    public Person() {
        firstName = "Owen";
        lastName = "Feik";
    }

    public Person(String newFirstName, String newLastName) {
        firstName = newFirstName;
        lastName = newLastName;
    }
}
```

Here, two constructors have been defined, one of which uses some default values, and another which is *overloaded*, with a different set of arguments. Any number of versions of the same constructor can be declared, as long as they have different arguments. The usual order for methods inside a class is attributes followed by constructors, followed by getters and setters and finally other methods.

Overloading

In the section on constructors above, we demonstrated method overloading. This is an example of *polymorphism*, where objects are processed differently depending on data type or class. Any method can be overloaded, they must simply follow the following rules

- They must have the same name
- They must have a different number of arguments or
- A different set of argument types or
- A different positioning of arguments
- To all other methods sharing the same name

Static Attributes and Methods

Static members are methods or attributes which are not specific to any instance of a class. A static variable will be shared among all objects of the class; only one instance exists, to which all class instances have a reference. They can therefore be very useful in certain instances where a global count or similar needs to be maintained.

```
public class Person {
    public static int population = 0;
    public String name;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;

        population++;
    }
}
```

In the above example, every time a person is created, the population counter will be incremented. It will not be reset to 0 on creation of an instance. Rather than being accessed through `instance.attr`, static members are accessed through `Class.attr`, because they are not associated with an instance. This is more or less to equivalent to a Python `@classmethod` decorated member.

A static method cannot access non static methods or attributes, and it cannot use the keyword `this` or the yet to be introduced `super`, because it is not associated with an instance. `static` should be used sparingly.

Standard Methods in Java

For all classes in a Java program, there are several standard methods that are very useful to define.

- the `equals` method, used to determine whether two instances of a class are equivalent. This is distinct from == which just checks whether two references are to the same object.

- the `toString` method, which returns a string representation of and object, similar to Python's `__str__` or `__repr__`. Like in Python `print`, this is automatically used with using `System.out.println`.
- the `copy` constructor, which is a constructor which accepts an instance of the class and creates an instance which is equivalent to the one passed in.

```
public class Person {
    public String firstName, lastName;

    public Person(Person toClone) {
        if (toClone == null) {
            System.out.println("Received a null reference.");
            System.exit(0);
        }

        this.firstName = toClone.firstName;
        this.lastName = toClone.lastName;
    }

    public boolean equals(Person other) {
        return this.firstName == other.firstName &&
            this.lastName == other.lastName;
    }

    public String toString() {
        return "Person: " + this.firstName + " " + this.lastName;
    }
}
```

Packages

A package is Java's version of a library in C or module in Python. They allow you to group classes to import at a later date. Packages prevent naming conflicts, allowing you to have two classes with the same name in the program without issue. Packages also give greater powers of encapsulation, improving our ability to control visibility or hide information.

To declare a package, we must have a `package` statement at the top of a file. This statement states the location of the file within the directory structure of the package.

```
package <..>.<.>;

// e.g.

package utilities.shapes;

public class Circle {

}
```

We can then reuse the classes from a package with the `import` statement. One can either import all of the classes from the package, or specific classes.

```
import <packageName>.*; // All classes in packageName
import <packageName>.<className>; // Just className
```

These classes are sourced from a directory specified via a Java environment variable, `CLASSPATH`. All classes in the current directory are included in an unnamed package called the `default` package. The classes will be available as long as `.` is included in `CLASSPATH`.

Information Hiding

Through classes, we gain access to power grouping tools which offer us encapsulation. However, thus far we have exclusively used `public` methods, losing some of the power available to us in Java's object oriented toolkit. Through information hiding tools such as method visibility modifiers, we gain the ability to hide information from external methods.

This allows a specific interface with the class (through getters, setters, etc) to be defined, which can be maintained while the internal functioning of the class can be modified at will.

In Java, we have three visibility modifiers.

```
public class Example {
    public void methodOne {
        // This method is available in all scopes.
    }

    protected void methodThree {
        /*
         * This method is visible to this class, as well as all
         * subclasses and other classes in the same package.
         */
    }

    void methodFour {
        /*
         * If no visibility is specified, the default visibility
         * is assumed. This is visible within the class and the
         * package, but not to subclasses or external code.
         */
    }

    private void methodTwo {
        /*
         * This method is visible only within the scope
         * of this class. It is not visible to subclasses,
         * and will not be inherited.
         */
    }


    /*
     * The order for modifiers is
     * <visibility> <static> <-> <final> <type>
     */
}
```

General form form a class follows the following guidelines.

- All attributes are `private` and are accessed through `public` getter and setter methods.

- Methods which should not be called externally are `private`, while those that are intended for external use are `public`.

Mutability

A class that contains `public` mutator methods, methods which alter its instance variables is a *mutable class*, and instances of this class can be described as *mutable objects*. On the other hand, classes with no methods which edit instance variables are *immutable*.

In general, an immutable class will lack setter methods. It may also have `final` attributes. All attributes should be set by the constructor.

Delegation

A class can *delegate* responsibilities to other classes. The object can invoke methods of other objects through *containership*. In this case, the two classes have an *association* relationship.

This essentially refers to using classes as data types. Rather than have an object store a series of x and y coordinates, I might create a point class and store an array of points. Now, my parent class contains points and is associated with them. It might delegate the function of calculating distance between points to the point class.

In doing this, we achieve seperation of concerns, making it easier to test the behaviour of isolated components and sub-components rather than having to deal with what might be an extremely complex single class.

Wrapper Classes

Wrapper classes in Java are essentially extensions of the primitive data types, which add some additional functionality and allow the primitive data types to be treated as classes. They are the same as their primitive counterparts, except that they change to the class naming convention. For example, `double` has the wrapper class `Double`. This convention is broken in two instances, with `int` becoming `Integer` and `char` becoming `Character`.

The wrapper classes add functionality to their primitives like `Integer.parseInt`. Every wrapper has a parse function of this form.

When these wrapper classes were introduced, the concepts of boxing and unboxing were introduced, which allow the simply conversion back and forth from primitive to wrapper.

Arrays

Arrays are useful when dealing with a large number of objects of the same type. They are declared as follows.

```
<type>[] <name>;
<type> <name>[];

// An array must be initialised.
// two methods for this:

int[] array = {0, 1, 2, 3};
int[] array = new int[100];
```

An array is a reference to an object, just like an instance of a class. Assigning one array to another will result in reference equality.

Multi-dimensional arrays (arrays of arrays) are available in Java. For example:

```
// 10 arrays of 10 items each
int[][] nums = new int[10][10];

// 10 arrays without defined length
int[][] nums = new int[10][];
```

Primitive arrays are initialised to 0. Object arrays are initialised to `null`. Arrays have an attribute `.length` which returns the number of objects the array can contain. Array equality is tested as follows, using the `java.util.Arrays` class.

```
import java.util.Arrays;

int[] arr1 = {1, 2, 3};
int[] arr2 = {4, 5, 6};
```

```
Arrays.equals(arr1, arr2);
```

`java.util.Arrays` also offers a few other useful methods.

```
import java.util.Arrays;

int[] arr1 = {1, 3, 2};

Arrays.sort(arr1);
System.out.println(Arrays.toString(arr1));
```

A for-each loop is available to arrays.

```
int[] arr = {0, 1, 2, 3, 4};

for (int a: arr) {
    System.out.println(a);
}
```

Strings

A `String` is a class which stores a sequence of characters. They can be initialised as literals or manually. Quotes in strings must be escaped.

```
"\n" // newline
"\t" // tab
"\"" // quotation mark
```

+ can be used to concatenate strings, in addition to any other object with a `toString` method. To get the length of a `String`, we have `String.length()`; note that this is not an attribute as in the array. Strings have some other useful methods

```
s = "Hello World";
s.toUpperCase(); // returns new String
s.split(" "); // returns array of Strings
s.contains("Hello"); // returns a boolean
```

```
s.indexOf("Hello"); // returns an integer
s.substring(2, 7); // returns new String
```

Because `String` is an immutable class, modifying operations will return a new `String` rather than modifying the original. Strings with the same characters will have the same reference by default, however if one creates an indentical string using `new`, they will have different references. It is therefore better to use `.equals()` rather than == for string comparison.

Input and Output

Input can be provided to programs through command line arguments, user input and through files. Programs can output data to `stdout`, the terminal or files.

Input

Command line arguments are received directly by the main function through its argument `String[] args`. This is an array of strings corresponding to the arguments provided when the program is run. When using an IDE, one usually has to configure the command line arguments to be passed to the program at execution.

To take input from a user interactively, we use the `Scanner` class provided by `java.util.Scanner`. `Scanner` is used as follows.

```
import java.util.Scanner;


Scanner scanner = new Scanner(System.in);
String s = scanner.nextLine(); // read chars until a newline.
boolean b = scanner.nextBoolean(); // read until a boolean.
int i = scanner.nextInt(); // read in an integer.
String s = scanner.next(); // reads one word (space delimited).

if (scanner.hasNext()) {
    s = scanner.next();
}
```

```
if (scanner.hasNextDouble()) {
    double d = scanner.nextDouble();
}
```

Note that we provide `System.in`, which is a static reference to `stdin`. Note that `Scanner.nextInt()` will error on `6.2`; it is quite fussy.
`Scanner.nextLine()` is the only method that will consume a newline character, so it can be useful to call it to chunk up data appropriately.
`Scanner.hasNext()` can be useful to check whether input is available.

File input and output are performed with the `java.io` package. It is important when working with files to use error handling.

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

try (BufferedReader br = new BufferedReader(new
    FileReader("text.txt"))) {
    String text = null;

    while ((text = br.readLine()) != null) {
        System.out.println(text);
    }
}
catch (IOException e) {
    e.printStackTrace();
}
```

Here, we use a `try` with resources block, which creates a `FileReader` object, upgrades this into a `BufferedReader` object and then reads an prints out the lines of the file one by one. In the event that the block runs into an error, it prints the stack trace. One can also use a `Scanner` to read a file by passing the `FileReader` to the `Scanner` object.

Output

The simplest form of output is through `System.out`, through the terminal.

```
System.out.println("Hello World");
System.out.format("%3.2f", 3.14159");
```

For file writing, we also use `java.io`. We can use a `PrintWriter`, which implements the same interface as `System.out`.

```
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;

try (PrintWriter pw = new PrintWriter(new
    FileWriter("text.txt"))) {
    pw.println("Hello World");
    pw.format("%3.2f", 3.14159);
}
catch (IOException e) {
    e.printStackTrace();
}
```

Build Management

Buil management and automation tools such as `make` or the one used in this subject, Maven are useful because they allow rapid building of large and complex projects. When using Maven, the structure and contents of a project are declared in a file `pom.xml`, the project object model file.

Bagel

Bagel, Basic Academical Graphical Engine Library is a graphics package for SWEN20003. Developed by Eleanor McMurtry is uses an existing library `LWJGL`.

```
import bagel.*;

public class BagelTest extends AbstractGame {
    public BagelTest() {
        super(800, 600, "Window Title");
    }
```

```
    public static void main(Sting[] args) {
        BagelTest game = new BagelTest();
        game.run();
    }

    @Override
    public void update(Input input) {
        // update method called each tick

        Image bagel = new Image('resources/bagel.png');
        bagel.draw(Window.getWidth() / 2.0, Window.getHeight() /
            2.0);

        if (input.isDown(Keys.DOWN)) {
            doAThing();
        }

        if (input.wasPressed(Keys.ESCAPE)) {
            Window.close();
        }
    }
}
```

Bagel provides a framework which one can extend to create a graphical program. Bagel automatically clears the screen, checks input and calls the user defined update method every tick. Documentation is available through Canvas.

Inheritance and Polymorphism

The best way to learn the motivation for these concepts is by considering a case where they haven't been applied. Let us consider for this purpose a chess game. This game needs a board to manage the game state, as well as implementations of various pieces.

We might reasonably suggest then that classes for a board, a pawn, a rook, a knight, a bishop, a queen and a king would be required. While the board would be more or less unique, the other pieces would be quite similar, needing to store

their colour, position and implement movement and capture logic. The solution to the duplication inherent in these classes is inheritance.

Rather than have a class for each of these pieces, we can create a "piece" super class and instead subclass from this to reduce code duplication.

```
public abstract class Piece {
    private boolean colour;
    protected int x, y;

    public Piece(boolean colour, int x, int y) {
        this.colour = colour;
        this.x = x;
        this.y = y;
    }

    public abstract void move();
}

public class Pawn extends Piece {
    public Pawn(boolean colour, int x, int y) {
        super(colour, x, y);
    }

    @Override
    public void move() {
        // move
    }
}
```

Here, our super class is `Piece` and it is subclassed or extended by `Pawn`. By marking our parent (super) class as `abstract`, we tell the compiler that objects of this class should not be directly instantiated, and instead only subclasses of `Piece` should be created. When we declare the `abstract` method `move()`, we provide no function body; this is instead an indication that all subclasses of `Piece` should define `move()`.

All instances of `Pawn` will have the attributes defined in `Piece`, though they will only be able to directly access `public` or `protected` members; subclasses will not be able to directly access their own colour. When we call `super()`, we are

running the constructor of `Piece` to set up these attributes for its child class `Pawn`. `super()` can only be called within the constructor of a subclass, and should be called as the first thing in that constructor. Parent class functionality can be accessed through `super.<method>`; super is always a reference to the parent class.

The `@Override` annotation in `Pawn` is used to indicate that this method is being overridden from the super class. It has no function and is a purely aesthetic inclusion, but can improve code clarity. Overridden methods must share parent return types or have the same parent type. `private` methods of the parent class cannot be overridden as they are out of scope. `final` methods are in scope, but cannot be overridden.

Although attributes of a parent class can be accessed by a child class if they are defined as `protected`, this is considered bad practice and they should instead be accessed through public or protected getters and setters. Privacy level must be maintained when overriding methods.

By setting up our pieces in this way, we avoid the duplication inherent in defining similar constructors and other methods like getters and setters in each different piece. The subclass `Pawn` can be thought of as a more specialised or specific version of `Piece`; all `Pawn`s are `Piece`s, but not vice versa.

`Object`

All classes in Java (implicitly) inherit from `Object`. Therefore all objects are of the type `Object` and if not defined elsewhere they also have the `toString`, `equals`, etc methods of the `Object` class.

When comparing objects it can be useful to know a few concepts.

```
boolean sameClass(Object a, Object b) {
    return a.getClass() == b.getClass();
}

boolean isSubClass(Object a, Object b) {
    return b instanceof a && !sameClass(a, b);
}

Piece p = new Pawn(); // here I assign a Pawn to a Piece
```

```
Pawn p = (Pawn) p; // here I downcast the Piece to a Pawn
```

The `Object` method `getClass` and the operator `instanceof` can be useful for type checking. It is important to note that a subclass can be stored in a variable of its parent type, and that typecasts can be used to convert between the two.

Types of Inheritance

A variety of different types of inheritance exist, which have various different applications.

- Single inheritance describes a subclass with a single super class.
- Multiple inheritance describes a situation with several super classes.
- Hierarchical inheritance exists when a single super class has many subclasses.
- Multi-Level inheritance implies subclasses of subclasses.
- Hybrid inheritance implies some combination of other types.
- Multi-path inheritance implies multiple sources of different properties.

Interfaces

Interfaces are in some ways similar to `abstract` classes. They also model concepts or ideas and cannot be themselves instantiated. They may contain only constans and abstract methods, and are used primarily to define behaviours of classes which can *implement* the interface.

A class which implements an interface "can do" all of the things the interface defines. Interfaces are often named according to the `<...>able` convention implying all of their members are e.g. `Driveable`. One might use an interface rather than a class for things that happen to share behaviour but which are not logically connected, for example a seatbelt and a hat might both be wearable, but wouldn't share a common super class.

```
public interface Printable {
    int MAXIMUM_PIXEL_DENSITY = 1000;

    void print();
}
```

Notice that despite not being explicitly abstract, the method `print()` has no body; the `abstract` is implicit. Again despite not explicitly stating as such, all attributes are `static` and `final`. All of an interfaces members are `public`.

```
public class Image implements Printable {
    public void print() {
        doThing();
    }
}
```

The `Image` class `implements` the `Printable` `interface`, and must therefore implement all methods defined by `Printable`. If a class wants to implement and interface without defining all of its methods, it must be `abstract`.

If one wants to have a method be optional in implementing classes, one can use `default`.

```
public interface Digitisable extends Printable {
    int MAXIMUM_PIXEL_DENSITY = 1000;

    default void digitise() {
        doThing();
    }
}
```

Here, `digitise()` is a standard implementation of the method, which can be overridden by classes which need different behaviour. As shown here, interfaces can extend other interfaces just as classes do, to allow greater specialisation.

Comparable

A useful interface in the standard library is `Comparable<Class>`, where class is the class to which the class implementing might be compared to. For instance, `String` implements comparable, allowing it to be compared to other strings. To implement comparable, a class must have the method `compareTo`.

```
public class Person implements Comparable<Person> {
    private String firstName, lastName;

    public String getFirstName() {
        return this.firstName;
    }

    public String getLastName() {
        return this.lastName;
    }

    public int compareTo(Person other) {
        int first =
            this.firstName.compareTo(other.getFirstName());
        if (first) {
            return first;
        }
        return this.lastName.compareTo(other.getLastName());
    }

}
```

`compareTo` must return a negative integer if the object is less than, zero if it is equal to and a positive integer if it is greater than the other object. This is used by `Arrays.sort` among other methods to generalise comparison behaviour. The behaviour of `compareTo` can be whatever the developer deems useful; in the above example, the two will be compared alphabetically.

Inheritance and Implementation

While a class can only extend a single parent class, it can implement any number of interfaces. In general, where inheritance is for generalising share

properties, implementation is for generalising shared behaviour.

Inheritance is more of a "is a" relationship; all children *are* parents.

Implementation is more of a "can do" relationship; all implementors *can do* the thing.

Representation of Relationships (i.e. UML)

When designing a Java or other object-oriented application, one should work through the process of identifying classes or knowns, followed by relationships between these nouns (has-a, is-a, can-do, etc) and refine these properties. It can then be useful to produce a diagram summarising this information. A standard way to do this is with a Unified Modeling Langauge (UML) diagram.

In UML classes are represented as boxes, with a title, a section for attributes and a section for methods. An abstract class depicted this way has its name italicised. An interface has its name (somewhat less elegantly) preceded by `<<interface>>`

An attribute is represented with some combination of the following qualities, in the listed order.

- Privacy (required); one of +, representing `public`, ~ i.e. package, # (`protected`), - (`private`).
- Name (required).
- Multiplicity, or quantity. This can be an array specifier like `[10]`, a range like `[1..10]`. It can also be open ended such as `[1..*]` or `[*]`. This does not specify the type of collection; a finite multiplicity might use an array while an open ended one might use a list. If omitted, a single item is assumed.
- Type (required), preceded by a colon.
- Default value, shown as an assignment like `= 10`.

For example `-name: String = ""` or `+values[*]: int` are valid attribute representations. A static attribute is underlined.

A method is represented similarly.

- Privacy (required), same as with attributes.
- Name (required).
- Arguments (required), shown as a parameter list of the form `(arg1: int, arg2: String)`. Can be empty, i.e. `()`.
- Return type (required), preceded by a colon. Like `: int`, or `: void`.

Some valid specifications include `+render(x: int, y: int): void` and `-reload(): void`. In the same way that an abstract class has its name italicised, the name of an abstract method is italicised.

Using these principles, we can now describe a single class in UML. To extend this, we need a way to describe relationships between classes. There are four main relationships in Java projects.

Association, which represents containment; objects which hold other objects internally. To represent this, rather than list other classes as attributes, we use an arrow from the containing object to the contained object. A unidirectional relationship is represented by an arrow, however if two classes contain each other, a line is used.

This line can have a name which defines what the relationship is, and can have multiplicity numbers at either end. The number defining how many of the object are held goes on the side of the object. A class may have association with itself.

If two objects are associated without one being dependent on the other (i.e. if either is held seperately to the attribute), this is represented with an unfilled diamond. A filled diamond implies that the class exists exclusively as an attribute.

Inheritance is represented with unfilled arrow heads from subclass to super-class. Implementation is represented through dashed arrows with unfilled heads. Depedency is a non-concrete relationship which suggests that a change to one class may affect another. It is represented through a dotted line from one class to another, with a chevron arrow head.

Generics

If one were to look at the comparable interface discussed earlier, the definition would look something like

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

Here, `T` is a *type parameter*. When one implements `Comparable<String>`, all instances of `T` in this implementation are now `String`. The benefit of defining classes or interfaces this way is that it allows one to write very generalisable classes which can work with a wide variety of objects.

A useful generic in the standard library is `java.util.ArrayList`, a generic list implementation.

```
import java.util.ArrayList;

public static void main(String[] args) {
    ArrayList<String> strings = new ArrayList<String>;
    strings.add("string one");
    strings.add("string two");

    for (String s: strings) {
        System.out.println(s);
    }
}
```

`ArrayLists` do not automatically release memory when they are shrunk; instead `trimToSize()` must be called to shrink them down to size.

To write a generic class, one does essentially what was shown in the interface example; include a type argument through `<T>`.

```
public class Wrapper<Type> {
    private Type obj;

    public Wrapper(Type obj) {
        this.obj = obj;
    }
```

```
    public Type getObj {
        return this.obj;
    }
}
```

Conventionally, `T` or another single character is used as the type argument. Multiple type arguments can be accepted, in the form `<T1, T2>`. If one wants to gaurantee some behaviour of the class, one can use the form

```
public class Generic<T extends Comparable<T>>
public class Generic<T extends SomeClass>
public class Generic<T extends SomeClass & Comparable<T> &
    List<T>>
```

To ensure the class fits a given model. A method with an argument of a generic type is known as a *generic method* and can be implemented like so

```
public <T> int genericMethod(T arg)
public <T> T genericMethod(String name)
public <T> T genericMethod(T arg)
```

Here, the first example is a generic method which accepts an argument of generic type and returns an integer, the second is a generic method which accepts a string and returns an object of a specified type and the third does both. The type passed to a generic method is inferred by context; either it will be the typed passed in or the variable in which the output is stored.

Collections and Maps

Collections

Java has two main constructs for storing data. These are collections and maps. Collections like `ArrayList` store ordered lists of objects while maps store key-value pairs.

If an `ArrayList` has a type which implements `Comparable`, it can be easily sorted through `java.util.Collections`' `sort` method. It is as simple as

`Collections.sort(list)`.

If one wants to create a variety of comparison types, one can use the `java.util.Comparator` class. `Comparator` is a generic interface which is implemented with a `compare(T obj1, T obj2)` method. One can then use `Collections.sort(list, comparator)` to sort according to a different metric. If one doesn't want to create a full class as a comparator, they can instead use an *anonymous inner class*.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public static void main(String[] args) {
    ArrayList<String> strings = new ArrayList<String>;
    strings.add("AAAAAAAAAAAHHH");
    strings.add("AAAAAAAAHHH");
    strings.add("AAAHHHAAAAHH");

    Collections.sort(strings, new Comparator<String>() {
        @Override
        public int compare(String s1, String s2) {

            asInS1 = s1.length() - s1.replace("H", "").length();
            asInS2 = s2.length() - s2.replace("H", "").length();

            return asInS1 - asInS2;
        }
    });
}
```

Here, we used a similar principle to a Python `lambda` sorting key, except we instead used a class because of course we did, its Java.

Collections in Java implement the following methods

- `int size()` which returns the number of elements in the collection.
- `boolean contains(Object element)` which checks if `element` is in the collection; this relies on `element.equals`.

- `boolean add(T obj)` which adds an element to the collection.
- `Iterator<T> iterator()` which returns an iterator for the collection.
- `for (T t: Collection<T>)` i.e. `for` loop iteration.
- `Object get(int index)` is not available in all collections, but is available in those that inherit from `AbstractList` such as `ArrayList` or `Vector`.

Several useful collections are

- `ArrayList` as seen, `ArrayLists` are like more flexible arrays.
- `HashSet`, which ensures uniqueness of elements, much like Python's `set`.
- `PriorityQueue` which allows ordering of elements in useful ways.
- `TreeSet` which offers very rapid lookups of unique elements.

Maps

Maps implement the methods

- `int size()`
- `boolean containKey(Object key)`
- `boolean containValue(Object value)`
- `boolean put(K key, V value)` which updates the value held under `key`.
- `boolean remove(Object key)`
- `Set<K> keySeta()` which returns a set of the keys in the map iteration.
- `Set<Map.Entry<K, V>> entrySet()` which returns a set of key-value pairs for iteration.
- `V get(Object key)`

As is evident from these methods, maps require two classes, `<K, V>`, which define the type of their keys and values respectively. For example

```
import java.util.HashMap;

public static void main(String[] args) {
    HashMap<String,Person> nameMapping = new HashMap<>();

    Person p1 = new Person("John Smith");
    Person p2 = new Person("Susan Wills");

    nameMapping.put(p1.name, p1);
    nameMapping.put(p2.name, p2);
}
```

Enumerations

Sometimes we want to define highly specific pieces of data as useful constructs for object oriented programming. The classical solution to this is the enumeration or `enum`.

```
public enum Rank {
    ACE, TWO, THREE, FOUR, FIVE, SIX, SEVEN,
    EIGHT, NINE, TEN, JACK, QUEEN, KING;

    private boolean isFaceCard() {
        return this.ordinal() > Rank.TEN.ordinal();
    }
}

public enum Colour {
    RED, BLACK
}

public enum Suit {
    SPADES, CLUBS, DIAMONDS, HEARTS
}

public class Card {
```

```
    private Rank rank;
    private Suit suit;
    private Colour colour;

    public Card(Rank rank, Suit suit, Colour colour) {
        this.rank = rank;
        this.suit = suit;
        this.colour = colour;
    }
}
```

By declaring enumerations rather than simply hardcoding strings or other constants, we keep our code cleaner. Java `enums` can include methods or attributes like any other class. They provide the methods `toString`, `compareTo` and `ordinal`

Variadic Parameters

Like Python's `*args`, Java has a facility for creating methods with an arbritrary number of parameters.

```
public int countThrees(int... numbers) {
    int count = 0;
    for (int num: numbers) {
        if (num == 3) {
            count++;
        }
    }
    return count;
}

countThrees(1, 2, 3, 4, 5); // => 1
countThrees(3, 3, 3); // => 3
```

The arguments will be passed in as an array of the specified type.

Method Manipulation

Functional Interfaces

A functional interface is an interface which contains only a single non-static method.

```
@FunctionalInterface
public interface Predicate<T> {
    public boolean test(T obj);
}
```

Some examples include

- `Predicate<T>`, a functional interface which represents a predicate, a single argument function which returns `true` or `false`.
- `UnaryOperator<T>`, a functional interface which represents a single argument function which returns an object of the same type. An example might be negation.

Lambdas

The primary purpose of functional interfaces is for use with lambda expressions. Lambda expressions allow us to "treat code as data"; in the example of functional interfaces, it allows us to instantiate them without implementing them.

```
Predicate<Integer> p = i -> i > 0;

p.test(10) // => true
p.test(-10) // => false
```

The syntax for a lambda, visible above is `<Type> <name> = <arg> -> <body>`. Lambda expressions are often useful as somewhat less clunky anonymous classes. They are fundamentally instances of functional interfaces.

Method References

While lambdas are useful for many settings, we have a facility to pass existing methods to functions directly when that is needed. This facility is the method reference.

```
List<String> names = Arrays.asList("Joe", "Margaret", "Steve");
names.replaceAll(name -> name.toLowerCase());
names.replaceAll(String::toUpperCase);


// methods can also be stored as functional interfaces
UnaryOperator<String> operator = String::toLowerCase;
```

Streams

A *stream* is a way to apply a series of methods to a collection of elements.

```
// Create a list of all elements in the list which are longer
    than 5
// characters and start with a "C", shifted to uppercase.
list = list.stream()
    .filter(s -> s.length() > 5)
    .filter(s -> s.startsWith("C"))
    .map(String::toUpperCase)
    .collect(Collectors.toList())
```

Streams are a very useful technique. The fundamental operations on streams are

- `map`; apply a function to each element
- `filter`; select a subset of elements with a predicate
- `limit`; perform a maximum number of iterations
- `collect`; combine all elements into some output format
- `reduce`; aggregate a stream into a final value

Software Design

Design Patterns

A design pattern is a solution to a recurrent problem in software development. Some of these include

- The singleton pattern involves having a class with a single instance which can be accessed globally. This is useful for situations when you need access to a given object from a variety of places in your program.
- The factory pattern defines a class or method which is used to create other objects with certain properties. This allows more flexible creation of objects. It can reduce code duplication, avoid issues with information access and improve abstraction.
- The template method pattern describes an implementation through an abstract class, leaving method implementation to subclasses. For example `AbstractSort` might have `abstract void swap` and `abstract void compare` left to subclasses. This pattern allows easy one-time definition of an algorithm, making it easily extensible to other types or objects.
- The strategy pattern is similar to the template method pattern but instead defines a class which accepts a "handle". This handle implements an interface which defines the behaviours requried by the class accepting the handle.
- The observer pattern allows multiple objects to subscribe to events ocurring in another object. A subject class will store a collection of observer objects which it will notify when a relevant event occurs. It often makes sense for a subject of be an interface like `Observable`.

In a general sense, the following classes of problems are more or less solved.

- Solutions for creating objects, such as the singleton or factory approaches.
- Solutions for dealing with how classes are structured and related.
- Solutions for handling interaction between classes.

Design Principles

Modularity

Designing code with modularity in mind means decomposing problems into smaller units or modules which are easier to understand, develop, manager, re-use and test.

In object oriented programming, classes are the main modules.

Cohesion

Cohesive designs have very clear focusses for each module. Each class has a strongly defined purpose which all of its methods and attributes help with.

Coupling

Coupling points between classes are points where two classes interact with each other. In ideal designs, coupling is minimised. Interfaces support this, as can the passing of objects as arguments to methods.

Open-Closed Principle

This dictates that modules should be easily extensible, through inheritance or polymorphism, but should not be directly modified.

Abstraction

Abstraction suggests that we should solve problems by creating classes or other objects which model data-flow and actions to solve problems.

Encapsulation

By keeping the implementation details of a class hidden (or in Java `private`) and exposing only certain methods we make our programs easier to work with. This is encapsulation, also known as data or information hiding.

Polymorphism

Polymorphism is the capacity to use an object (or method) in a wide variety of ways. In Java this takes the form of overloading, subtyping through overriding or similar or generics.

Delegation

Ideally, a necessary computation should be performed by the class with the greatest amount of relevant information, to minimise gathering before the computation. Thus we *delegate* this task to the relevant task.

Symptoms of Poor Design

- The overall application is difficult to modify; it displays *rigidity*
- Changing a single component causes the whole application to break; it displays *fragility*
- The application cannot be broken down into reusable modules; it is *immobile*
- The code contains "hacks" and workarounds; compromises which exist to preserve the initial design; *viscosity*
- The application contains lots of overly clever or unnecessarily highly optimised code that reduces clarity; it displays excessive *complexity*
- Lots of *repetition* of code occurs
- The application has a large amount of complex flow of data and convoluted logic; it displays *opacity*

Event Driven Programming

The general form of a program is sequential; programs run through from top to bottom, taking actions according the evaluation of logical statements. This

form is useful for a very many applications, where the purpose is straightforward and largely static.

If, however, we want to create an application which responds dynamically to changes in the state of the program, the *event-driven programming* paradigm can be useful.

Event driven programming involves using events and callbacks to allow the program to respond to changes in state, such as user input. Applications of this include the observer pattern explored earlier, and exception handling. Event driven programming is most commonly used for the development of graphical user interface (GUI) programs. In this setting, one generally uses a framework, providing their own functions to undertake specific functionality in response to predefined events.

Event driven programming is generally (at least practically) *asynchronous*; there is little gaurantee as to the order that functions will be called in. The key words associated with this paradigm are

- State; the properties of a given object in the system; whether it is actived, focussed, hidden, etc.
- Event; a flag which responds to a change in state. When some aspect of state changes, the event is fired and related functions are called.
- Callback; also known as a listener, a callback is a method triggered by an event.

Event driven programming can allow for better encapsulation and information hiding, can make things more consistent in structure, making it easier to maintain a cohesive codebase. It can also allow very easy addition or removal of behaviour from a program.

On Frameworks

A software framework will often take care of the core execution loop for the programmer, doing things like rendering the application and firing events. Frameworks generally consist of a set of abstract classes and interfaces, as well as a core which knows how to use these components. The programmer uses

these tools to allow them to remove much of the busy-work of application development.

Exceptions

Errors in Java (or any programming language) can be broken down into syntax, semantic and runtime errors. Of these, syntax errors prevent compilation, being errors in the way code is written resulting in an illegal program, semantic errors describe logical errors precluding correct output and runtime errors resulting in illegal states and crashes.

Runtime errors include division by zero, index errors, IO errors and similar. While syntax and semantic errors cannot be effectively guarded against, we have tools in our arsenal to deal with runtime errors, through exception handling.

In Java a runtime error is modelled as an `Exception` object. Exceptions are handled through `try-catch-finally` blocks.

```
try {
    // execute code block which may throw Exception
}
catch (<ExceptionClass> e) {
    // code to execute if the exception is thrown
}
finally {
    // code that is always executed irrespective of any
        exception
}
```

We can also include multiple `catch` blocks to catch multiple exception types. To create exceptions to be caught, we use the `throw new Exception()` statement. Methods which throw unhandled exceptions should have the `throws <ExceptionClass>` descriptor to indicate this.

```
public int method(String input) throws NullPointerException {
    if (input == null) {
        throw new NullPointerException();
    }
```

```
        return input.length();
    }
```

New exceptions can also be defined by extending another `Exception` class. When catching an `Exception`, all subclasses of that exception will also be caught. Thus `catch (Exception e)` will catch all exceptions. If an unhandled exception could arise, the compiler will not allow the program. `Error` and it's subclasses are the exception to this as these conditions will be allowed to compile and will cause a program exit.

Good Coding Practice

It is important to observe good coding practices including

- Consistent layout
- Avoidance of long lines
- Consistent use of tabs or spaces
- Neat arrangement
- Good variable, method and class naming
- Minimal code duplication
- Constant commenting

Comments should be used to document the code for future maintainers of the code. Ideally, code should be readable without comments. If all code were removed, good comments should allow a program to be reconstructed. Comments should be associated with a block of code associated with a step in an algorithm.

Comments should appear before code, representing a kind of primer for it.

`javadoc` is a tool provided with the JDK to allow compilation of specially formatted comments into html documentation.

```java
/**
 * Adds two to the input value.
 *
 * @param addTo the value to add two to
 * @return the value with two added to it
 */
public int addTwo(int addTo) {
    return addTo + 2;
}
```

Testing

Bugs and other errors in software are generally discovered through testing. Conventionally testing is performed in three stages; unit testing, where minimal sections of a program are tested; integration and systems testings where already unit tested sections are tested to ensure properly functioning integration between them and finally acceptance testing where the overarching system is verified to work per requirements.

Code can be made more testable by breaking it down as much as possible into the most minimal units.

JUnit

JUnit is a Java utility used to test Java code. Tests are written in Java

```java
import static org.junit.Assert.*;
import org.junit.Test;

public class PersonTest {
    @Test
    public void testPersonInstantiation() {
        Person person = new Person("Owen", "Feik");
        assertEquals(person.getFullName(), "Owen Feik");
    }
```

```
    @Test
    public void testEquality() {
        Person person0 = new Person("Owen", "Feik");
        Person person1 = new Person("Joe", "Bloggs");
        Person person2 = new Person("Joe", "Bloggs");

        assertNotEquals(person0, person1);
        assertEquals(person1, person2);
    }
}
```

Automated testing tools like JUnit are very useful for large projects as they are fairly easily to set up, well scaleable, easily to run repeatedly and effective at finding introduced bugs.