Object-Oriented Programming Notes

written by Tyler Wright

*An important note, these notes are absolutely **NOT** guaranteed to be correct, representative of the course, or rigorous. Any result of this is not the author's fault.*

1 Classes, Objects, and the Basics of Java

1.1 Definition of a Class

A class is a type for variables and objects, encapsulating many key ideas, it is:

- A structure with fields
- A closed system
- A module
- A namespace (to avoid clashes)
- A unit (for testing).

1.2 Definition of an Object

An object is an entity that combines both data (attributes) and methods uniquely, objects...

- ...can have a state and/or act
- ...can have a unique identity
- ...are responsible for their own data
- ...can communicate with other objects via methods
- ...can expose its own information.

Object-oriented programming is (quite obviously) based around this concept.

1.3 Instances

Every object is an **instance** of a class, or rather, an object is an instantiation of a class. In this way, classes are like a blueprint for objects.

Objects are **constructed** by using the constructor for a class to instantiate the class. In Java, the constructor always has the same name as the class.

1.4 The Difference between Classes and Objects

Classes are:

- Catergorised
- Compile time
- Blueprints
- Static
- Immutable

Objects are:

- Instances
- Runtime
- Entities
- Dynamic
- Mutable

1.5 The Java Toolchain

In Java, the written `.java` files are compiled into `.class` files which are run by the **JVM** (Java Virtual Machine).

1.6 The Java Entry Point

Java programs start in a `main` function which takes in an array of strings (command line arguments).

1.7 Exception Handling

In Java, a `try-catch` block is used to catch exceptions at runtime.

1.8 Exit Codes

In Java, we can use `System.exit(i)` to close the program at anytime. The meaning of the `int i` passed to the function is defined by the programmer, but in general 0 is good, anything non-zero is an error.

1.9 Immutable Objects

Immutable objects cannot have their value change after declaration and their value must be assigned on declaration. The `final` keyword is used for this in Java. Final methods cannot be overriden and final classes cannot be extended.

2 Memory

2.1 The Stack

The stack is the memory given to a thread of execution. When a function is called, a block is reserved on the stack for local variables and bookkeeping data. This memory is freed once the function ends.

2.2 The Heap

The heap is the memory given to dynamic allocation. There's no pattern to the management of blocks on the heap, it can be allocated and freed at any time. This memory is used for large data to avoid overflowing the stack.

2.3 The Lifecycle of Objects

Once objects no longer have any references to them, they are eligible for garbage collection by the Java garbage collector. This is a system that deallocated unused memory in Java. So, at the end of function calls, all variables local to that function become unused and are collected.

2.4 Pass by Value

Primitive types such as `int`, `double` and, `boolean` are passed by value.

Objects in Java are also passed by value but a *reference* to the object is passed. This means we can call the methods of the object and have its values change, but we cannot reassign the variable inside a function.

2.5 Equality

It is important to note that `==` is reference equality in Java so two strings with the same underlying data may not be equal. This is why we have `String.equals` to test string equality.

3 Scope

3.1 The `this` Keyword

`this` references to the current object whose method is begin executed. So, the use of this keyword will throw errors in places where there is no object such as static methods.

3.2 The Private Modifier

Private elements are class-scope. Only the class can directly change or use the element, this underpins encapsulation.

3.3 The Default (or no) Modifier

Default elements are package-scope.

3.4 The Protected Modifier

Protected elements are subclass-scope.

3.5 The Static Modifier

Static elements are associated to a class rather than an object. This means they are processed at compile time and can be accessed **without** instantiation.

Static functions cannot use other non-static members unless given to the function or instantiated inside the function.

One key use of static is the `main` function, the entry point of the program.

4 Inheritance

If we need multiple classes with similar attributes, it makes sense for them all to inherit core functionality from a super (parent) class. This can be achieved with the `extends` keyword.

Additionally, inheriting classes can override the super classes methods to implement their own functionality (overriding the super classes base functionality). This can be similarly done by adding the `abstract` modifer to your attributes or methods

but this will mean there is no default implementation and the inheriting subclass must implement the method.

The abstract modifier can also be applied to classes where it means it can only be instantiated as a super class.

4.1 Polymorphism

When we have classes (say `ClassTest`) inheriting other Classes (say `Test`) we can refer to the subclass as an instance of the super class (all `ClassTest` objects are also `Test` objects). This doesn't change the object, it just changes how the object is viewed, so to speak.

4.2 Interfaces

We can only extend a single class in Java, but it would be nice to be able to implement multiple classes in one place, this is where interfaces come in. They are similar to abstract classes with only final abstract methods and no attributes - they are essentially a set of methods your subclass must implement.

4.3 Encapsulation

Encapsulation is all about keeping a classes functionality identical or adding more features without changing the implementation of the class.

This is accomplished by only having class data accessed by specified 'getter' and 'setter' methods, thus we can change the attributes and methods of the class at will as long as the getters and setters function as they did before.

4.4 Single Dispatch

Single dispatch is when we call a method of an object but the call is overridden by a method in the subclass of the referenced object. Such a decision is made at runtime.

4.5 Double Dispatch

Double dispatch is used to resolve interactions between two objects of known super class but unknown subclass. By making each subclass implement a set of overloaded methods (`visit` for example) with a definition for each subclass and one generic implementation.

The generic implementation will call the specific implementation of the other object as it knows its own type (of course). Thus, the other object then knows what is calling it and its own type so can produce the corresponding reaction.

4.6 Downcasting

Downcasting is a last resort programming technique where classes are casted to their subclasses. This can lead to complications if done incorrectly as there's no guarantee that it is in fact an instance of the subclass. We can use generics to avoid this technique most of the time.

4.7 Generics

We can define classes of a generic type to make them more versatile. For example, we can have a `Collection` of any instantiable object in Java.

5 Patterns

5.1 The Visitor Pattern

The visitor pattern allows a specific method within a visitor class to be called by a specific visiting class. So, in a Java board game, if we had a game class that processed different types of moves with a generic move super class we could declare the game class as a **visitor**.

Then, the moves would be the **visiting** classes. In our visitor, we would define an accepting move for each move subclass and then use a method call on the visiting classes to have them 'visit' the visitor. As the visiting class knows its type, it will call the specific method.

5.2 The Iterator Pattern

The iterator pattern is used to sequentially access the elements of an object without exposing it's underlying structure. This uses encapsulation as it doesn't rely on how the class stores or handles its values, it just requires that it be iterable.

In this pattern, we have an iterable class and an iterator class. The iterator will use the iterable interface to extract values from the class and handle the indexing.

5.3 The Strategy Pattern

The strategy pattern is used to define a set of algorithms that can be specifically chosen to carry out specific behaviour.

We have a context where we require that we can use multiple different algorithms (strategies) at will. So, we define specific strategy subclasses and these can be passed to the context who can use the implementation of the strategy super class to carry out the strategy on itself.

5.4 The Observer Pattern

In this pattern, we have a main subject class and many observer classes. The subject class registers and unregisters observers and can announce to update observer classes. There are two main types of observer, push and pull:

- Push observer subjects send their state to the observer when it updates
- Pull observer observers must request the state from the observer

5.5 The Composite Pattern

The composite pattern is used to store multiple components in a tree-like structure. Each component can store a simple component with no children or composite classes that can contain a set of more components.

5.6 The Model-View-Controller

This is a pattern of patterns and has no specific correct definition. It consists of a model and a view and the essence of the idea is regulating their communication thorough specific classes.

If we allow interaction between these intermediary classes, we get a Model-View-Presenter pattern.