Object Oriented Programming

To be maintainable is to be
- Simple to locate code responsible for a particular feature.
- Simple to understand what the code does.
- Simple to modify behaviour.
- Difficult to introduce bugs.

Types of Languages

Definitions
- **Declarative languages** specifies what to do (not how to do it).
  - Includes **functional, logic and reactive languages** (which reasons about streams of data).
  - E.g. HTML and SQL
  - OCaml: you give an example on how to achieve something, the compiler has the choice to do something completely different (e.g. tail recursion).
- **Imparative languages** specifies what and how to do it.
  - **Procedural languages** group things into functions.
  - **OOP** groups procedures together - dividing a large procedural program into many small chunks, which are easier to reason about.

Characteristics of OOP are:
- Encapsulation
- Abstraction
- Inheritance
- *Subtype* polymorphism

Languages often pick and mix concepts for the language creator to complete their tasks.
- E.g. procedural programming in OCaml.
- Functional programming in C++.

OOP is not always appropriate, smaller programs **don't need abstractions**, e.g. using Python for data science. But OOP is better for larger scale programs.

The JVM

Java was made for the web, where each machine visiting a page has different processors.

Model	Note
Traditional model	The compiler compiles source code for a particular machine, that can only run on that type of system. But how do you compile for every system and serve the correct binaries to visitors?
Using an interpreter	High level languages are not space efficient (not suitable for the 2000s internet speed), and source code is visible to everyone.
The Java model	The Java compiler compiles Java source code to **bytecode**, which is translated to machine code on the fly.

Definition
Bytecode is machine code for a fictional machine.

Translating from bytecode to machine code is quick, as the toughest part of compilation is going from source code to compiled code.

Bytecode is slower than native code, but compared to interpreted code:
- Bytecode is compiled and not easy to reverse engineer.
- JVM ships with libraries making the bytecode small.

The JIT

JVM uses a **JIT** which profiles your code. If it sees a code running over and over again, it compiles it to machine code. So Java programs become faster the longer you run them.

It is used for backend: servers are long running programs that the JVM can learn and optimise.

Java

Basic Language Features

Project Layout

There is **one class per file**, and the class name and file name must match (this is to make the code more maintainable).
- The **JRE** is what you need to run Java bytecode.
- The **JDK** is what you need to compile Java. The JRE is a subset of JDK.

Data Types

Java is strongly typed. Types are either built-in **primitive types** or **reference types**, which starts with a capital letter.

- `boolean`
- `byte`
- `short`
- `int`
- `long`
- `float`
- `double`
- `char`, which is the only **unsigned integer**.

Variables can be **promoted** or **narrowed** to another type.

(DANGER!) Inferring types on local variables can make the code more clear... or more confusing.

```
var courseName = "Java"; // clearly a String
var n = 1;               // what type is this?
```

Procedures

A Java *"function"* is made up of a **prototype** and a **body**, the prototype specifies the function name, arguments and return type. There are **no pure functions** in Java. There are only procedures which can manipulate state outside the functions.

Encapsulation

Definitions

Keyword	Definition
Object	A bundle of state and behaviour.
Class	A template for a specific type of object.

- The state of an object is called a **field**, the behaviour are called **methods**.
- A class defines a **type** and the **implementation of methods**.

Class

All code in Java are contained in classes.

For best readability, in each file declare in order.

1. Imports
2. Constants
3. Fieilds
4. Constructors
5. Methods

Constructors

Constructors have the same name as the class, have no return types (why?), and can be overloaded.

```
public class MyClass {
  public MyClass(int a) {}
  public MyClass(int a, int b) {}
}
```

> **Note**
> If you don't declare any constructors, Java will declare one for you.
>
> You can look at class file signatures with
>
> `javap -c MyClass.class`

Parameterised Classes

Polymorphism can be done through generics. Classes are defined with placeholder types.

```
public class Vec2D<T> {
  public T x;
  public T y;
}
```

We fill them in when we create an object with them.

```
Vec2D<Integer> l = new Vec2D<Integer>();
```

> **Note**
> The type parameter must be a **reference type**.

Statics

Objects don't allow global state as state is self contained.

> **Definition**
> A **static field** is created only once in the program's execution.

Pros	Cons
Auto synchronised cross instances.	Make code harder to understand.
Space efficient.	

> **Note**
> Make static fields `final` whenevr possible.

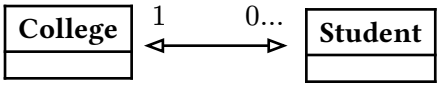
Static methods don't belong to an object, they are used to group related methods.

Universal Modelling Language

MyClass
- privateField + publicField
- privateMethod() + publicMethod()

Model **has-a** association with arrows.

- A college has zero or more students.
- A student has one college.



The college can store its students in a list, the student can store his college in a single variable.

Visibility

We want to provide an API that exposes what it needs to and hide everything else. Encapsulation allows us to decouple the API from the underlying state: changing how the state is represented internally will not affect code depending on it.

	Class	**Subclass**	**Package**	**Everywhere**
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	
no modifier	✓	✓		
`private`	✓			

Note
Encapsulation is also called information hiding.

Public state is almost never needed:
- Create private variables by default.
- Never make a public variable private, it can break code.

Immutability

Immutable	**Mutable**
Easier to reason about.	Updating values in place is more efficient.
Reduces scope for bugs.	
Thread safe.	

Definition
Immutable classes: state can be set at initialisation but cannot be changed.

Immutable classes can be achieved with the `final` modifier, or using Java **records** (since Java 16).

```
public record Vec2D(int x, int y) {}
```

Immutable classes are everywhere in the JDK: `Integer`, `String`, `LocalDate`, `Optional`, etc.
