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.
