

## 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 reason 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).
- **Imperative 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 <b>bytecode</b> , 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.

- |   |   |
|---|---|
| <ul style="list-style-type: none"> <li>• boolean</li> <li>• byte</li> <li>• short</li> <li>• int</li> </ul> | <ul style="list-style-type: none"> <li>• long</li> <li>• float</li> <li>• double</li> <li>• char, which is the only <b>unsigned integer</b>.</li> </ul> |
|---|---|

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. Fields
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** whenever possible.

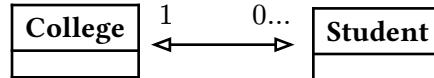
**Static methods** don't belong to an object, they are used to group related methods.

## 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.

## Data Representation

### Primitive and Reference Types

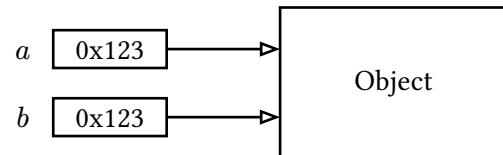
#### Definitions

- A **primitive value** is a chunk of memory that holds the value of the variable.
- **Objects** are referred to by reference - the variable links to a separate chunk of memory.

Changes to the block of memory being pointed to will reflect in both reference variables.

#### Note

An array is a reference type.



### String Optimisation

A string is immutable and stored in a read-only part of memory. Variables assigned the same string content will have the same reference address from optimisation.

### The Call Stack

A function puts 3 things on the **call stack** when called.

- Local variables created by the function.
- Memory address to jump to when complete (return address).
- A copy of the arguments passed to it.

They are held in the function's **stack frame** in the call stack.

#### Definition

A **stack** is a FIFO data structure.

**Local variables** are deleted when a stack frame is popped. Static variables have global scope.

#### Note

There is no tail call optimisation in Java.

### The Heap

We cannot resize variables in the stack: that would require moving everything above it on the stack.

**The heap** is more flexible, there are gaps between objects in the heap.

- Primitives and references cannot change size, so they go on the stack.
- Everything else goes on the heap.

### Pointers and References

#### Definition

A **pointer** is a chunk of memory that holds an address to another memory.

Pointers	References
Can be treated like numbers, can do arithmetic with it.	References are restricted pointers.
Gives us raw memory access (which could be dangerous).	
No way to test if a pointer is valid.	References can be tested.

C, C++ Supports both references and pointers, ML and Java only allow references.

---

### Pass by Value/Reference

- **All Java parameters are pass by value.** These *values* passed to a function can include references that points to an object in heap.
- In C, the variables passed to a function can point to a variable from the calling function.

### Inheritance

Inheritance is where a **subclass** inherits state and functionality from a **superclass**.

- Subclass use the `extends` keyword to inherit from a superclass.
- The subclass can directly access public and protected members of the superclass.

All Java classes must inherit from some other class, if not specified it will inherit from `java.lang.Object`.

```
public class MyClass; // will be replaced by the compiler as:  
public class MyClass extends Object;
```

In a UML diagram, inheritance is an unfilled arrow pointing towards the superclass.

```
class A;  
class B extends A;  
class C extends B;
```

When constructor is not explicitly written for C, when creating C, we will call:

1. Constructor of C
2. Which calls the constructor of B
3. Which calls the constructor of A

We can explicitly call the constructor of the superclass with `super()`:

### Type Casting

- **Widening conversion** cast up the tree, e.g.  $C \rightarrow B \rightarrow A$
- **Narrowing conversion** is checked at runtime, if the type does not match will cause type errors.

```
B b1 = new C()  
B b2 = new B();  
  
C c1 = (C) b1; // ok  
C c2 = (C) b2; // type error
```

### Shadowing

If a variable in the subclass has the same name as the superclass, we can specify where the variable comes from.

```
// running in C  
// these are all different variables  
this.x = 10;  
((B) this).x = 20;  
((A) this).x = 30;
```

#### Note

This is very ugly, please avoid at all cost.

## Overriding

### Definition

**Annotations** are not part of the language, but hints to the compiler so it can perform stricter checks.

Use the `@Override` annotation to make sure it overrides a function from superclass.

```
@Override
public void myFunc();
```

### Definitions

- An **abstract class** forces a subclass to override abstract methods.
- An **abstract method** has no implementation.

- In UML, an abstract class has *italic* name if drawn in computer, { curly braces } if drawn by hand.
- Interfaces just have the word “interface” in an UML diagram.

Interface groups classes that implement a set of methods.

## Inheritance vs Composition

- Inheritance represents an *is-a* relationship.
- Composition represents a *has-a* relationship.

Implementation of a stack with inheritance  
(inappropriate)

```
class Stack extends Vector;
```

A better implementation uses composition.

```
class Stack {
    private Vector internal;
}
```

---

## Polymorphism

- **Subtyping polymorphism** is where many kinds of objects provide the same method.
- **Parametric polymorphism** uses generics.
- **Ad-hoc polymorphism** uses overloading.

**Static polymorphism** decides the function to run at runtime.

```
Person p = new Student();
p.dance(); // this runs Person::dance()
```

Java applies static polymorphism on state and static methods.

**Dynamic polymorphism** decides at runtime when we know the child’s type.

```
Person p = new Student();
p.dance(); // this runs Student::dance()
```

Java applies dynamic polymorphism on methods.

### Note

Dynamic polymorphism has a runtime overhead.

Polymorphism allows:

- Less changes needed to add features.
- So less likely to make mistakes.

## Multiple Inheritance

Inheriting from multiple classes causes name clash called the **diamond problem**.

There will be loads of state and method name clashes.

What should happen if we call `plumbtrician.doJob()`?

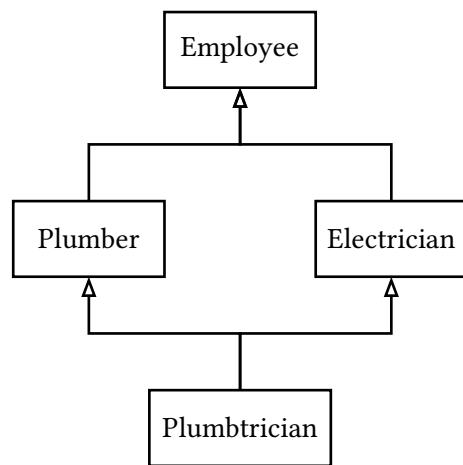
In Java

- Classes can have at most 1 direct parent.
- But multiple interfaces are allowed: interfaces don't have implementations, it just says the class must provide the methods.

If the interfaces require the same method that's fine.

### Note

Interfaces can have **default implementations**, so new functions can be added without breaking existing classes (that implements the interface).



## Method Resolution

When there is a name clash, the resolution priority from highest to lowest is

1. Class methods
2. Interface methods
3. Methods from superclass

To call a default method from an interface, write

```

class MyClass implements InterA, InterB {
    public void myMethod() {
        InterA.super.myMethod(); // calls InterA implementation
        InterB.super.myMethod(); // calls InterB implementation
    }
}
  
```

## OOP Principles

### OCP: Open to extension, Close to modification

- Make classes easy to add new behaviour.
- But hard to change existing behaviour.

### LSP: Liskov Substitution Principle

Subtypes must be behaviourally substitutable to the parent type without negative side effects.

E.g. code below is not behaviourally substitutable for the interface it is implementing.

```

class MyClass implements InterA {
    public void myMethod() {
        throw new UnsupportedOperationException("myMethod");
    }
}
  
```

---