Object Oriented Programming
Prof. Robert Harle

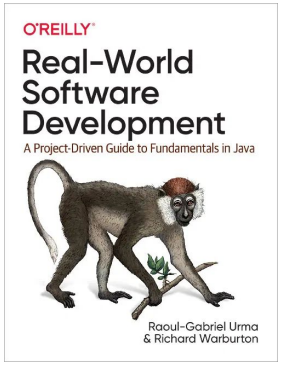
IA CST, Michaelmas 2025

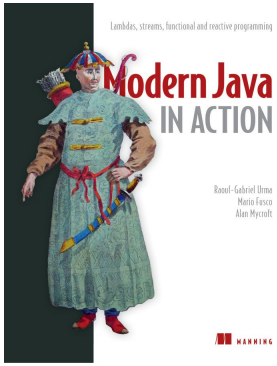
The OOP Course

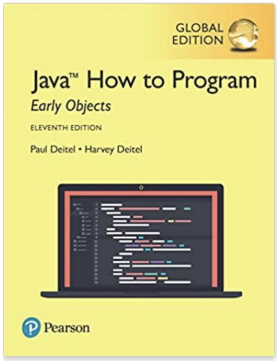
1. Intro to Java
2. Class Design and Encapsulation
3. Memory
4. Inheritance
5. Polymorphism
6. Object Lifecycle, Garbage Collection and Copying
7. Collections, Comparisons
8. Generics
9. Coupling, Errors and Exceptions
10. Design Patterns, Lambdas, Method References and Streams

Books and Resources I

- OOP Concepts
  - Look for books for those learning to first program in an OOP language (Java, C++, Python)
  - *Java: How to Program* by Deitel & Deitel (also C++)
  - *Thinking in Java* by Eckels
  - *Java in a Nutshell* (O' Reilly) if you already know another OOP language
  - Java specification book: http://java.sun.com/docs/books/jls/
  - Lots of good resources on the web
- Design Patterns
  - *Design Patterns* by Gamma et al.
  - Lots of good resources on the web







Books and Resources II

http://www.cl.cam.ac.uk/teaching/current/OOProg/

(Links out to moodle site)

Books and Resources III

Chat GPT?

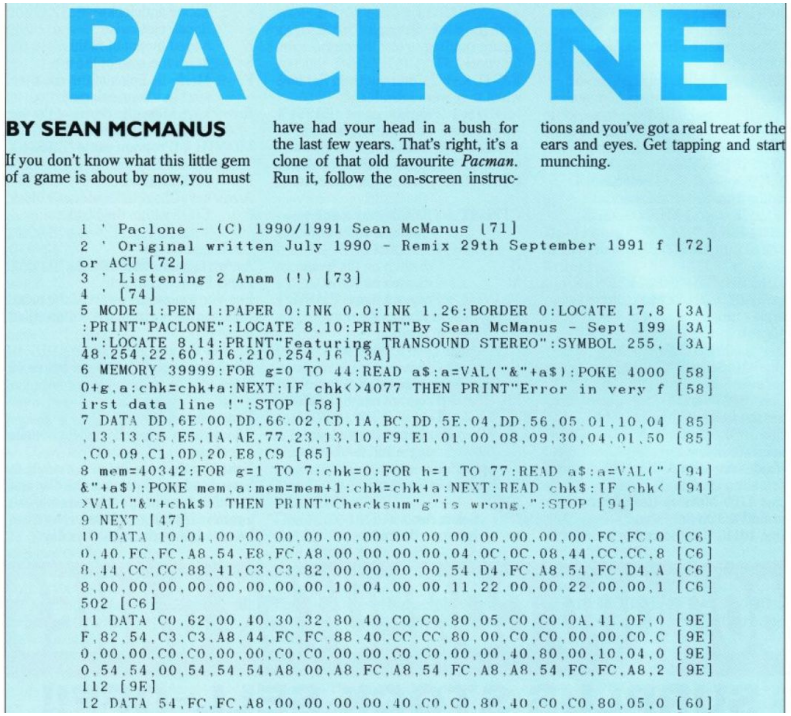
Gemini?

(Other generative AIs are available. Your home may be repossessed if you do not keep up repayments on your mortgage. Etc etc)

Motivations, Languages, OOP Intro

Motivating OOP

Battling Complexity



PACLONE

BY SEAN MCMANUS

If you don't know what this little gem of a game is about by now, you must have had your head in a bush for the last few years. That's right, it's a clone of that old favourite *Pacman*. Run it, follow the on-screen instructions and you've got a real treat for the ears and eyes. Get tapping and start munching.

```
1 ' Paclone - (C) 1990/1991 Sean McManus [71]
2 ' Original written July 1990 - Remix 29th September 1991 f [72]
or ACU [72]
3 ' Listening 2 Anam (!) [73]
4 ' [74]
5 MODE 1:PEN 1:PAPER 0:INK 0,0:INK 1,26:BORDER 0:LOCATE 17,8 [3A]
:PRINT"PACLONE":LOCATE 8,10:PRINT"By Sean McManus - Sept 199 [3A]
1":LOCATE 8,14:PRINT"Featuring TRANSOUND STEREO":SYMBOL 255, [3A]
48,254,22,60,116,210,254,16 [3A]
6 MEMORY 39999:FOR g=0 TO 44:READ a$:a=VAL("&"+a$):POKE 4000 [58]
0+g,a:chk=chk+a:NEXT:IF chk<>4077 THEN PRINT"Error in very f [58]
irst data line !":STOP [58]
7 DATA DD,6E,00,DD,66,02,CD,1A,BC,DD,5E,04,DD,56,05,01,10,04 [85]
,13,13,C5,E5,1A,AE,77,23,13,10,F9,E1,01,00,08,09,30,04,01,50 [85]
,C0,09,C1,0D,20,E8,C9 [85]
8 mem=40342:FOR g=1 TO 7:chk=0:FOR h=1 TO 77:READ a$:a=VAL(" [94]
&"+a$):POKE mem,a:mem=mem+1:chk=chk+a:NEXT:READ chk$:IF chk< [94]
>VAL("&"+chk$) THEN PRINT"Checksum"g"is wrong.":STOP [94]
9 NEXT [47]
10 DATA 10,01,00,00,00,00,00,00,00,00,00,00,00,FC,FC,0 [C6]
0,40,FC,FC,A8,54,E8,FC,A8,00,00,00,00,04,0C,0C,08,44,CC,CC,8 [C6]
8,44,CC,CC,88,41,C3,C3,82,00,00,00,00,54,D4,FC,A8,54,FC,D4,A [C6]
8,00,00,00,00,00,00,00,00,10,04,00,00,11,22,00,00,22,00,00,1 [C6]
502 [C6]
11 DATA C0,62,00,40,30,32,80,40,C0,C0,80,05,C0,C0,0A,11,0F,0 [9E]
F,82,54,C3,C3,A8,44,FC,FC,88,40,CC,CC,80,00,C0,C0,00,00,C0,C [9E]
0,00,00,C0,C0,00,00,C0,C0,00,00,C0,C0,00,00,40,80,00,10,04,0 [9E]
0,54,54,00,54,54,54,A8,00,A8,FC,A8,54,FC,A8,A8,54,FC,FC,A8,2 [9E]
112 [9E]
12 DATA 54,FC,FC,A8,00,00,00,00,40,C0,C0,80,40,C0,C0,80,05,0 [60]
```

Large software gets complicated fast

It became clear it was hard to write this code but also it was really hard to **maintain it**

In the 1960s they were searching for ways to tame this complexity

Maintainability Wishlist

- Simple to locate code responsible for a particular feature
- Simple to understand what the code does
- Simple to add or remove a new feature
- Simple to change existing behaviour
- Make it (more) difficult to introduce new bugs

Types of Languages

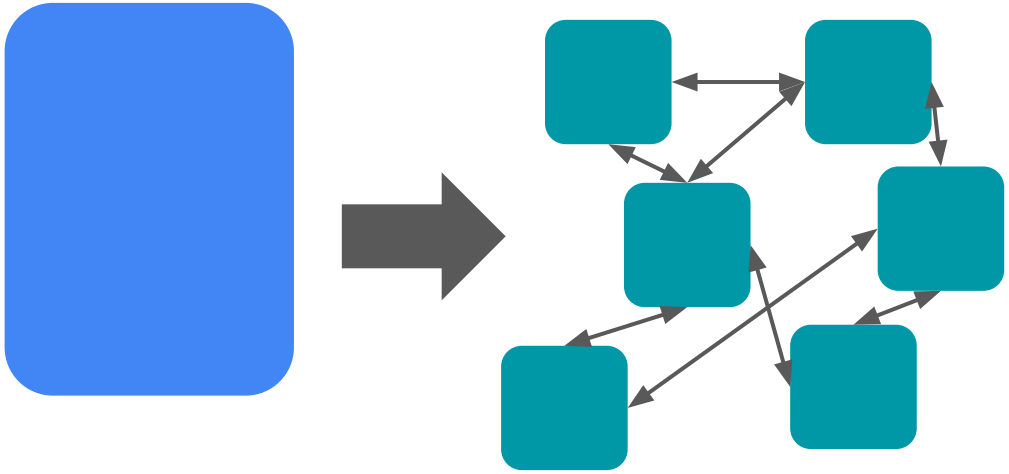
- Declarative - specify what to do, not how to do it.
  - Functional - functions at the core
  - Logic - reason about facts and rules
  - Reactive - reason about streams of data and events
  - E.g. HTML describes what should appear on a web page, and not how it should be drawn to the screen
  - E.g. SQL statements such as “select * from table” tell a program to get information from a database, but not how to do so
- Imperative – specify both what and how
  - Procedural - group code into procedures
  - OOP - group procedures and data together
  - E.g. “double x“ might be a declarative instruction that you want the variable x doubled somehow. Imperatively we could have “x=x*2” or “x=x+x”

OCaML

- **OCaML is a functional language** and therefore declarative
  - It may appear that you tell it how to do everything, but you should think of it as providing an explicit example of what should happen
  - The compiler may optimise i.e. replace your implementation with something entirely different but 100% equivalent.

What is OOP?

Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of **fields**, and code, in the form of procedures (often known as **methods**). Objects are usually defined by **classes** that group fields and methods together.



Procedural code monster

OOP objects interact to achieve same result

These objects are easier to reason about and maintain

Characteristics of OOP

1. Encapsulation
2. Abstraction
3. Inheritance
4. (Subtype) Polymorphism

We will cover these concepts in the rest of the course and see how they help develop software that can cope for changing requirements as well as improve maintainability of code (*when done properly*).

Warning #1: Languages are rarely ‘pure’

A given language is like a ‘pick n mix’ of concepts that the language creator needed for their task, or which other developers have requested

Either languages:

- become very niche/specialist (in which case they may be ‘pure’)
- grow to be general purpose (in which case they become behemoth jack-of-all-trades).

Java, C++, ...

Warning #2: OOP isn't always appropriate

- Poor choice for smaller programs where abstractions are not needed
- May not be as intuitive for reasoning / mathematical types of problems
- Often involves more boilerplate code and memory footprint in exchange for abstractions
- Requires thinking about stateful objects
- Can introduce coupling without care (e.g. bad inheritance, more dependencies). We'll explore this later.

Intro to Java

Why Java?

Java was designed as an OOP language

It remains widely used — www.tiobe.com/tiobe-index/

It is quite forgiving for beginners since it does not require manual memory management

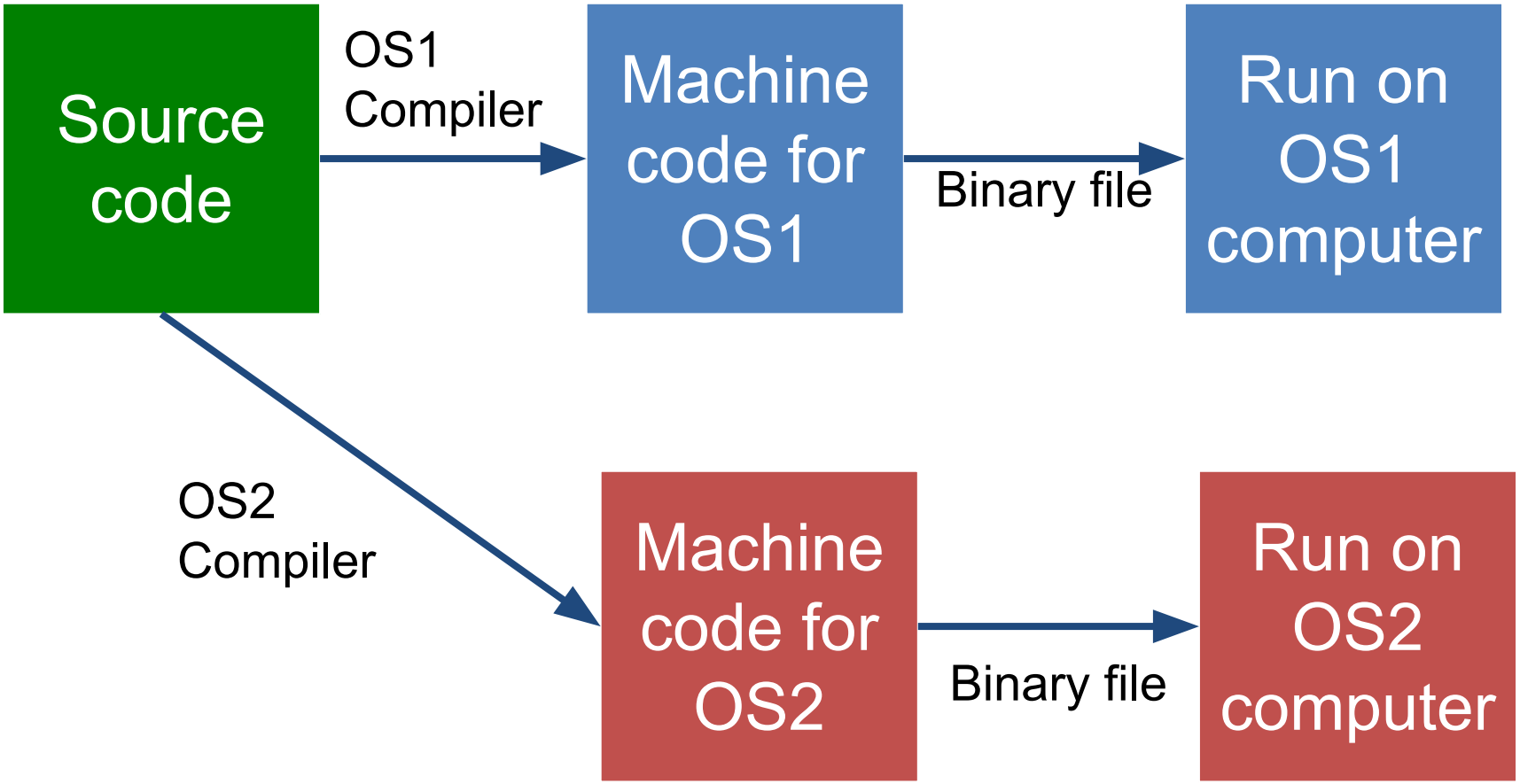
Java's Virtual Machine

- *Java* was intended as an early language to connect different devices (1990s) and was thus well placed when the web came along.
  - But many architectures were attached to the internet – how do you write one program for them all?
  - And how do you keep the size of the program small (for quick download)?

- Could use an interpreter (→ Javascript). But:
  - High level languages not very space-efficient
  - The source code would implicitly be there for anyone to see, which hinders commercial viability.

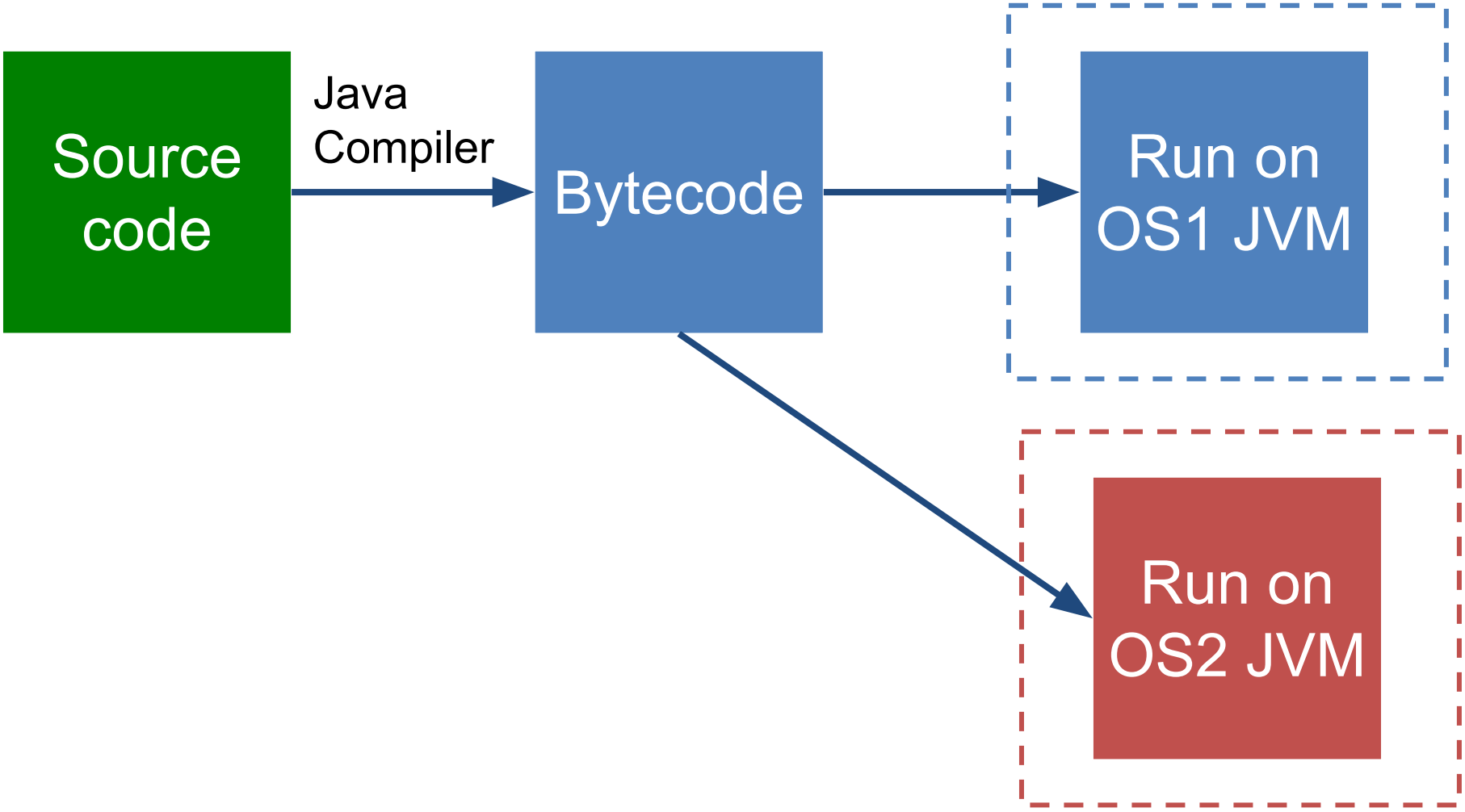
- Went for a clever hybrid interpreter/compiler

Traditional Model



(Might also need to adapt the source code to work with each architecture)

Java Model



The JVM (Java Virtual Machine) is a piece of software that converts Java bytecode (an architecture agnostic machine code) to local machine code

Pros/cons

- + Bytecode is compiled so not easy to reverse engineer
- + The JVM ships with tons of libraries which makes the bytecode you distribute small
- + The toughest part of the compile (from human-readable to computer readable) is done by the compiler, leaving the computer-readable bytecode to be translated by the JVM (→ easier job → faster job)
- - Still a performance hit compared to fully compiled ("native") code (although the gap closes all the time)

BTW Python also does this

Although it's less explicit than with Java, python will generate .pyc files that are the Python bytecode for your program

(note Java bytecode and Python bytecode are not the same)

In both cases you explicitly invoke the virtual machine to run your program

```
java YourProgram
```

```
python your_program.py
```

Java Hello world

In HelloWorld.java:

```
public class HelloWorld {
    public static void main {
        System.out.print("Hello world");
    }
}
```

In terminal:

```
> javac HelloWorld.java
> java HelloWorld
```

Java Hello world

In HelloWorld.java:

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

In terminal:

```
> javac HelloWorld.java
> java HelloWorld
```

There is one class (unit of code) per file

The class name and the filename **must** match (capitals and all)

Java Hello world

In HelloWorld.java:

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

In terminal:

```
> javac HelloWorld.java
> java HelloWorld
```

Scoping (of functions, of loops, etc) is handled by explicit braces

In python this is done with whitespace

In Java you can butcher the whitespace and get away with it (but don't)

Java Hello world

In HelloWorld.java:

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

In terminal:

```
> javac HelloWorld.java
> java HelloWorld
```

main is function that Java knows to execute automatically

It takes an array of Strings that are the arguments (if any) given on the command line when running

It returns void (think None in python)

Java Hello world

In HelloWorld.java:

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

In terminal:

```
> javac HelloWorld.java
> java HelloWorld
```

Java has grouping of classes called packages

Here, there is a package System. It contains a subpackage out. It contains a function println which prints a supplied line to screen. "Hello world" is the supplied line

Java Hello world

In HelloWorld.java:

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

In terminal:

```
> javac HelloWorld.java
> java HelloWorld
```

Every statement ends with an explicit semi-colon

Java Hello world

In HelloWorld.java:

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

In terminal:

```
> javac HelloWorld.java
> java HelloWorld
```

This compiles the Java bytecode and stores it in a file HelloWorld.class

Java Hello world

In HelloWorld.java:

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

In terminal:

```
> javac HelloWorld.java
> java HelloWorld
```

This executes the bytecode on the JVM (the java program)

Note we specify the class name to run and **not** the .class or .java files

Java will look in HelloWorld.class to see if there is a main method to execute. If not it will fail.

Practicalities

Java Developer's Kit (JDK)

- Contains javac and other useful tools you need
- Download and install the latest openJDK on your system

Integrated Dev Environment

- With built-in debugger to explore what is happening
- I recommend **IntelliJ IDEA**

(Note there is also the JRE — Java Runtime Enironment — this is a subset of the JDK that contains the JVM but ***not*** the compile tools like javac

jshell

When you download the Java JDK you get `jshell` for free

This is a REPL (Read-Eval-Print Loop) like you've used with Python

Great to start out with **but** not the 'normal' way to interact with Java

Representing State (Data)

State

Modelling interaction requires a notion of states that can be observed and changed.

For example:

Watching movies on netflix (state of your user account)

Making bank transactions (state of your bank balance)

Eating food (state of your body)

For keeping track of state we have variables…

Variables in Java

Values (data) are stored in memory and are referred to using variables in code

`int myVar = 10;` creates an integer value 10 named myVar

Java is **strongly typed** — you have to explicitly assign types to variables*. Those types may be built-in **primitive types** or **reference types** (more on this later):

```
double amount;          // declare 'amount' as a double
int count = 0;          // variable of primitive type
String courseName = "OOP"; // courseName refers to
                           // String reference type
```

** subject to the `var` discussion in a few slides*

Aside: Naming conventions

In Python we use snake_case

- course_name = ‘OOP’
- num_lectures = 10
- assign_marks()

In Java, we use CamelCase

- String courseName = “OOP”;
- int numLectures = 10;
- void assignMarks() {...}

Primitive Types in Java

"Primitive" types are the built in ones. They are building blocks for more complicated types that we will be looking at soon.

- `boolean` – 1 bit (true, false)
- `char` – 16 bits as an **unsigned** integer (0 to 65,535)
- `byte` – 8 bits as a signed integer (-128 to 127)
- `short` – 16 bits as a signed integer
- `int` – 32 bits as a signed integer
- `long` – 64 bits as a signed integer
- `float` – 32 bits as a floating point number
- `double` – 64 bits as a floating point number

Reference Types in Java

Any type that isn't a primitive is a reference type

- `boolean` – 1 bit (true, false)
- `char` – 16 bits as an **unsigned** integer (0 to 65,535)
- `byte` – 8 bits as a signed integer (-128 to 127)
- `short` – 16 bits as a signed integer
- `int` – 32 bits as a signed integer
- `long` – 64 bits as a signed integer
- `float` – 32 bits as a floating point number
- `double` – 64 bits as a floating point number

Immutable to Mutable State

ML

```
- val x=5;
> val x = 5 : int
- x=7;
> val it = false : bool
- val x=9;
> val x = 9 : int
```

Java

```
int x=5;      // Create variable and assign value
x=7;          // (Re) assign value

int x=9;      // Compile fail - attempt
              // to redeclare x
```

Type conversion

Variables of one type can be **promoted** or **narrowed** to another type, where it is appropriate to do so.

For example:

If you do arithmetic on different types, Java *implicitly* **promotes** one argument to the *widest* range.
E.g. 2.0 * 3 results in 6.0 as a double

You can also *explicitly* ***narrow*** the type through a cast (could be dangerous)
E.g. (int) 6.4 results in 6

Local variable type inference

A recent addition to Java is the ability to infer types on local variables:

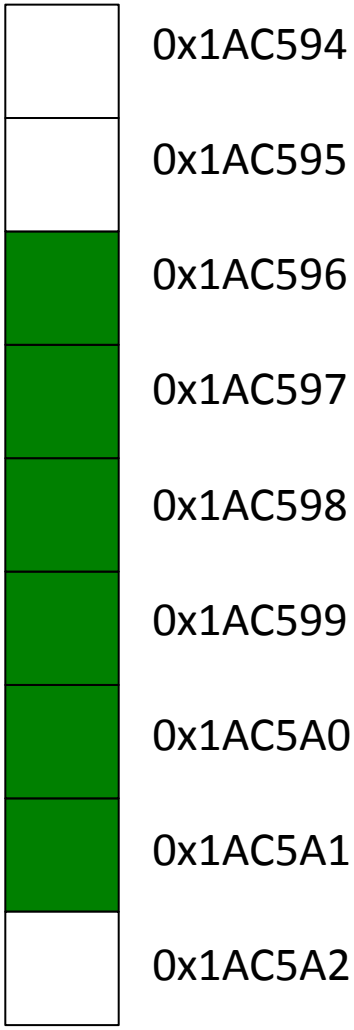
```
var courseName = "Java";
var data = getData();
var data = new ArrayList<Map<String, Integer>>();
```

Can make code more readable when used correctly. But use good judgement - some things aren't helped:

```
var x = 7; // Is this an int? short? char?
```

Arrays

```
// Both of these are valid
byte[] arraydemo = new byte[6];
byte   arraydemo2[] = new byte[6];
```




```
jshell> int f, g[], h;
f ==> 0
g ==> null
h ==> 0

jshell> int[] f,g,h;
f ==> null
g ==> null
h ==> null
```

Naming variables

You tend to write code once but read the same code many more times.

Optimise for maintenance and readability.

Java convention: use Camel Case with an initial lowercase letter

E.g. fontColour, age, xComponent but *not* variableToHoldAge, etc.

Representing Behaviour (Functions)

Function Prototypes

- Functions are made up of a prototype and a body
  - Prototype specifies the function name, arguments and *possibly* return type
  - Body is the actual function code

```
fun myfun(a,b) = …;

int myfun(int a, int b) {...}
```

Actually *procedures*

- More correctly, *functions* are like mathematical functions: they take inputs and provide an output
- Now we have **procedures**: these can manipulate state **outside** of the function (a 'side effect'), and may have no return type at all

```
int x = 1;

void demo() {
    x = x+1;
}

demo();  // x is 2
demo();  // x is 3
```

```
int x = 1;

int demo(int a) {
    x = x+1;
    return a+x;
}
demo(1);  // returns 3
demo(1);  // returns 4
```

Overloading Functions

- Same function name
- Different arguments
- Possibly different return type

```
int myfun(int a, int b) {…}
float myfun(float a, float b) {…}
double myfun(double a, double b) {...}
```

- But <u>not</u> just a different return type

```
int myfun(int a, int b) {…}
float myfun(int a, int b) {…}
```

X

Objects and Classes
(Behaviour and state grouped together)

Objects

- An **object is a bundle of state and behaviour**
- The state of an object is defined through its **fields**
- The behaviours of an object is defined through its **methods** (OOP speak for function/procedure)
- "Invoking a method" means executing the associated behaviour of a specific object

Classes

- A class is a blueprint/template for a specific type of object
- A class defines both type and implementation
  - Type: where can the object be used
  - Implementation: how the object does things
- The methods of a class can be seen as an API

In Java, **all** source code is contained in classes (this isn't a requirement of OOP, although it's common)

Java Class Structure

```
import java.util.ArrayList; // 0. import statements
public class Vector2D { // start of the class definition
// 1. constants
// 2. fields
// 3. constructors
// 4. methods
} // end of the definition
// no code allowed here!
```

Every class must be in its own file, called <classname>.java

Loose Terminology

State

Fields

Instance Variables

Properties

Variables

Members

Behaviour

Functions

Methods

Procedures

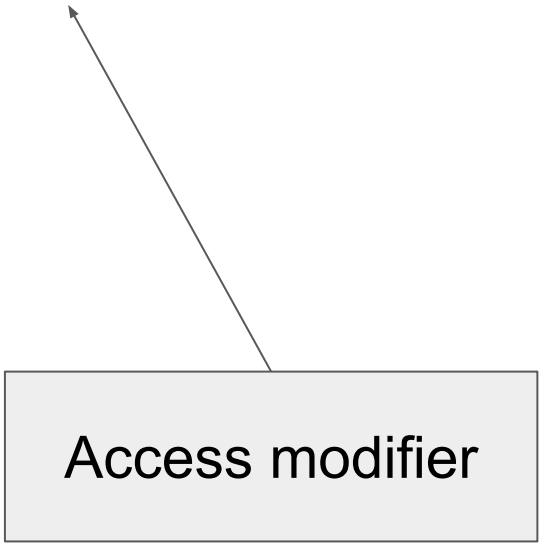
Naming Classes

Class names are often nouns, and use camel case with a **capital** letter at the start

E.g. Vector2D, ScreenWriter, …

Declaring fields

```
public class Vector2D {

    public double x;
    public double y;

}
```



Declaring a constructor

```
public class Vector2D {

    public double x;
    public double y;


    public Vector2D(double x, double y) {
        this.x = x; // the this keyword
                    // refers to the object itself
        this.y = y;
    }
}
```

If you don't give any constructor, Java creates an empty constructor for you that does the minimum. It would be equivalent to:

```
public Vector2D() {    }
```

Note!

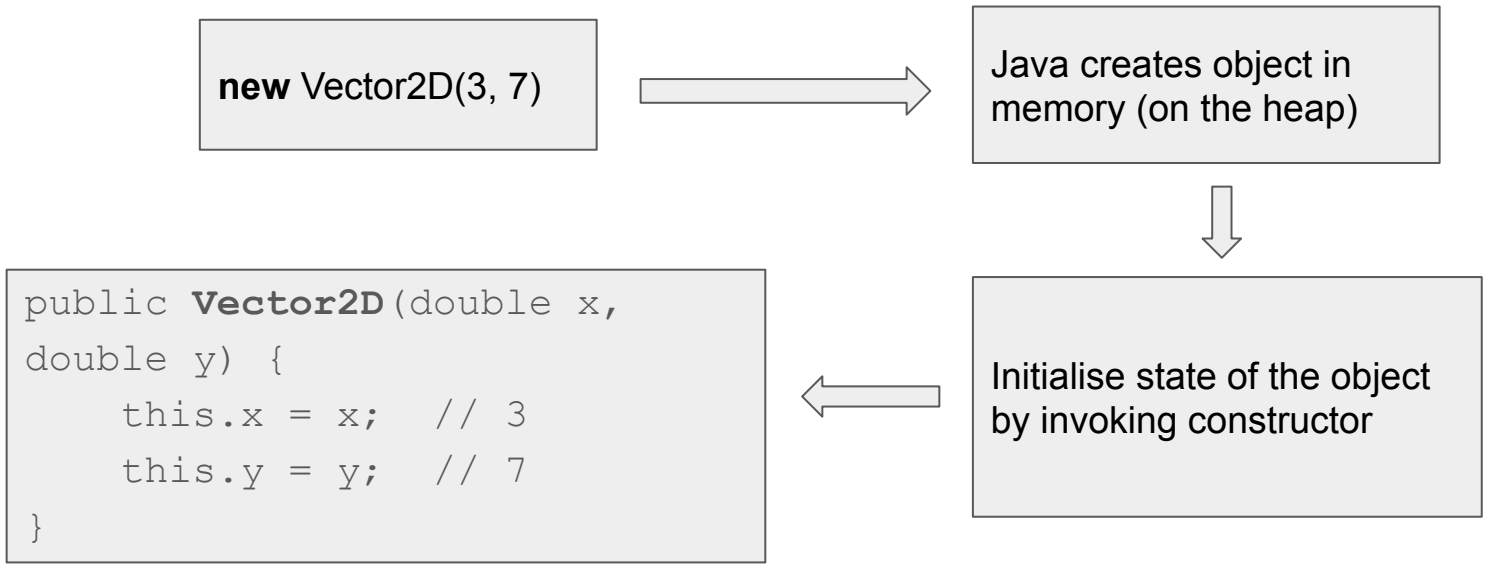
```
public Vector2D(double x, double y) {
    …
}
```

1. Constructors don't return **anything**, not even void. (Why?)
2. Constructors have the same name as the class

Creating Objects

We use the `new` keyword plus a constructor to create an object

```
Vector2D myVector = new Vector2D(3, 7);
```



Overloading Constructors

```
public class Vector2D {

    public double x;
    public double y;

    public Vector2D(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public Vector2D() {
        this.x = 0;
        this.y = 0;
    }
}
```

```
Vector2D vector1 = new Vector2D(3, 7);
Vector2D vector2 = new Vector2D();
```

As long as the signatures differ, you can have as many constructors as you like

Parameterised Classes

- ML's polymorphism allowed us to specify functions that could be applied to multiple types

```
> fun self(x)=x;
val self = fn : 'a -> 'a
```

- In Java, we can achieve something similar through *Generics*; C++ through *templates*
  - Classes are defined with placeholders (see later lectures)
  - We fill them in when we create objects using them

```
LinkedList<Integer> = new LinkedList<Integer>()

LinkedList<Double> = new LinkedList<Double>()
```

You can create those too…

```
public class Vector2D<T> {

   public T x;
   public T y;


      public T getX() {
      return x;
   }

   // etc
}
```

We'll have lots more to say on these later

Using `static` in your classes

Static fields

- A **static** field is created only once in the program's execution, despite being declared as part of a class

```
public class ShopItem {
   private float mVATRate;
   private static float sVATRate;
   ....
}
```

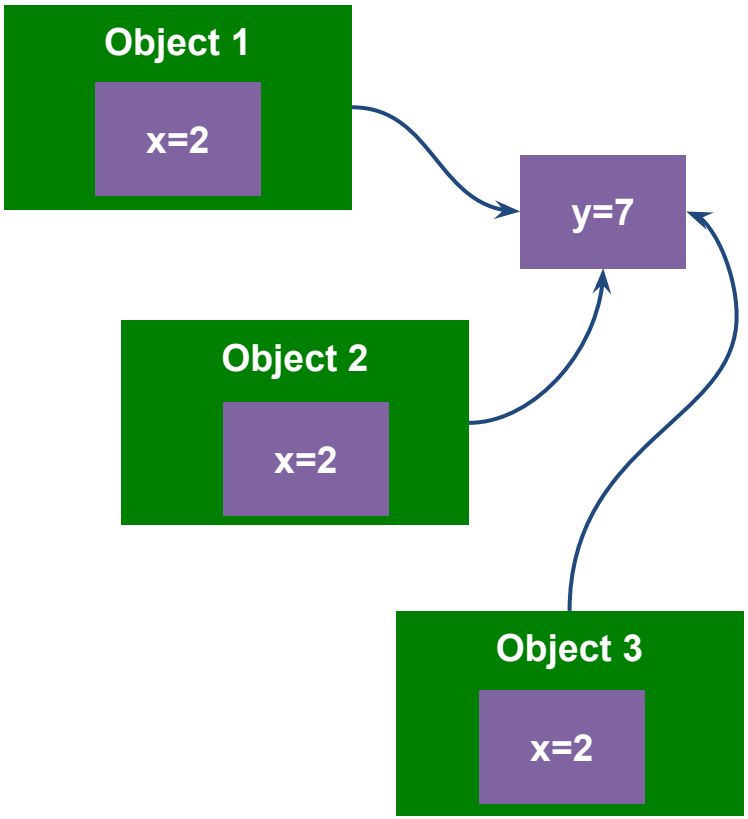
One of these created every time a new ShopItem is instantiated. Nothing keeps them all in sync.

Only one of these created ever. Every ShopItem object references it.

Static fields

```
public class Whatever {
      public float x = 2;
      public static float y = 7;
}
```

Only one instance of the field is created and every object uses that one instance



- Pros:
  - Auto synchronised across instances
  - Space efficient
- Cons:
  - Makes code harder to understand (best for `final` constants)

Static Methods

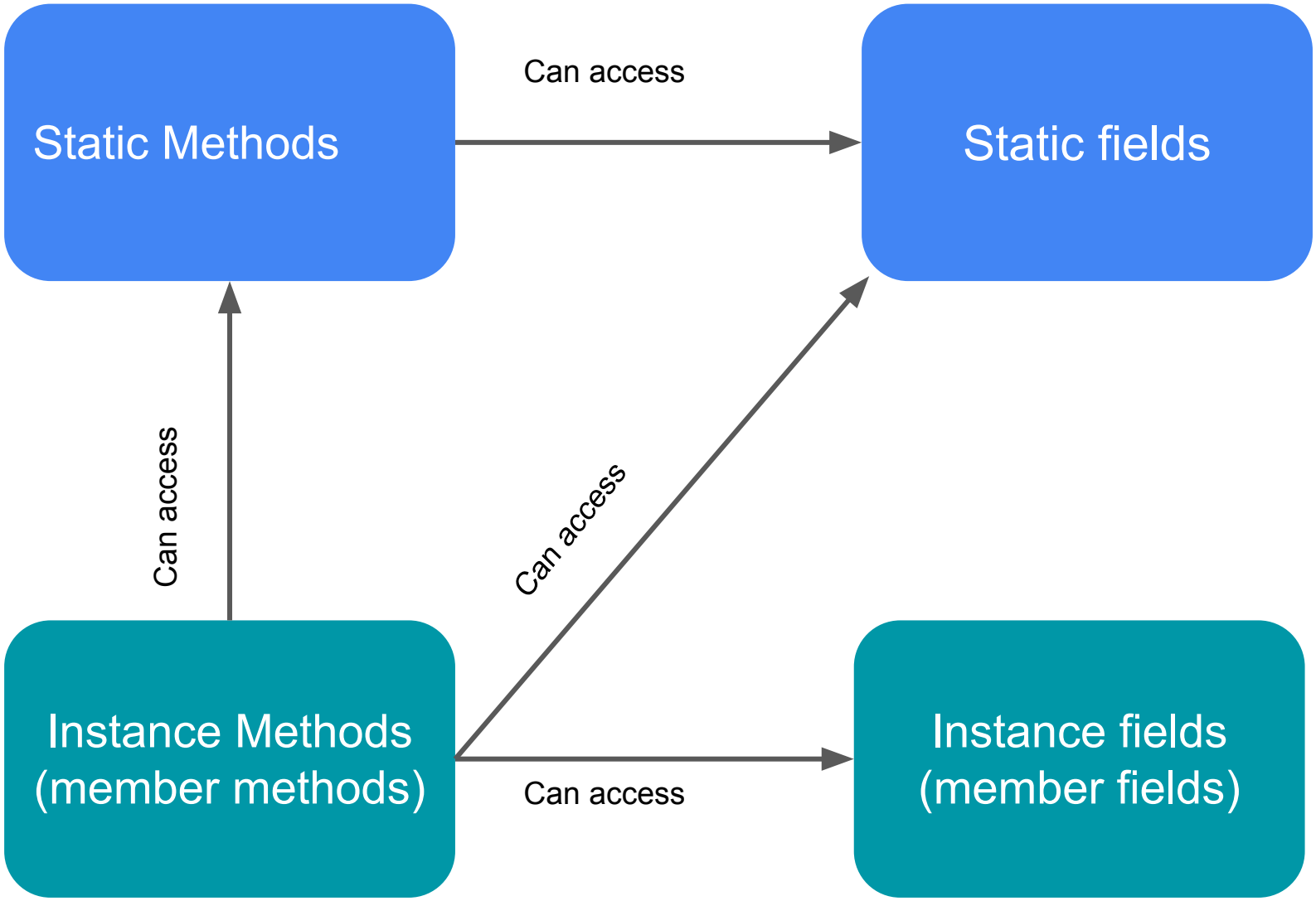
Methods that don't 'belong' to an object, but make sense in the class

```
public class Maths {
  public float sqrt(float x) {…}
  public double sin(float x)
{…}
  public double cos(float x)
{…}
}..
Maths mathobject = new Math();
mathobject.sqrt(9.0);
…
```

vs

```
public class Maths {
  public static float sqrt(float x)
{…}
  public static float sin(float x) {…}
  public static float cos(float x)
{…}
}..
Maths.sqrt(9.0);
…
```

Static Methods



Static Methods

- Easier to debug (only depends on static state)
- Self documenting
- Groups related methods in a Class without requiring an object
- The compiler can produce more efficient code since no specific object is involved
- Enables a readable factory method pattern
  - LocalDate.now()
  - List.of()

Class Design and Encapsulation

OK, you can make classes. But how do you decide what goes in a given class?

Rookie error: God/Monster class

Just have one class for each project you do

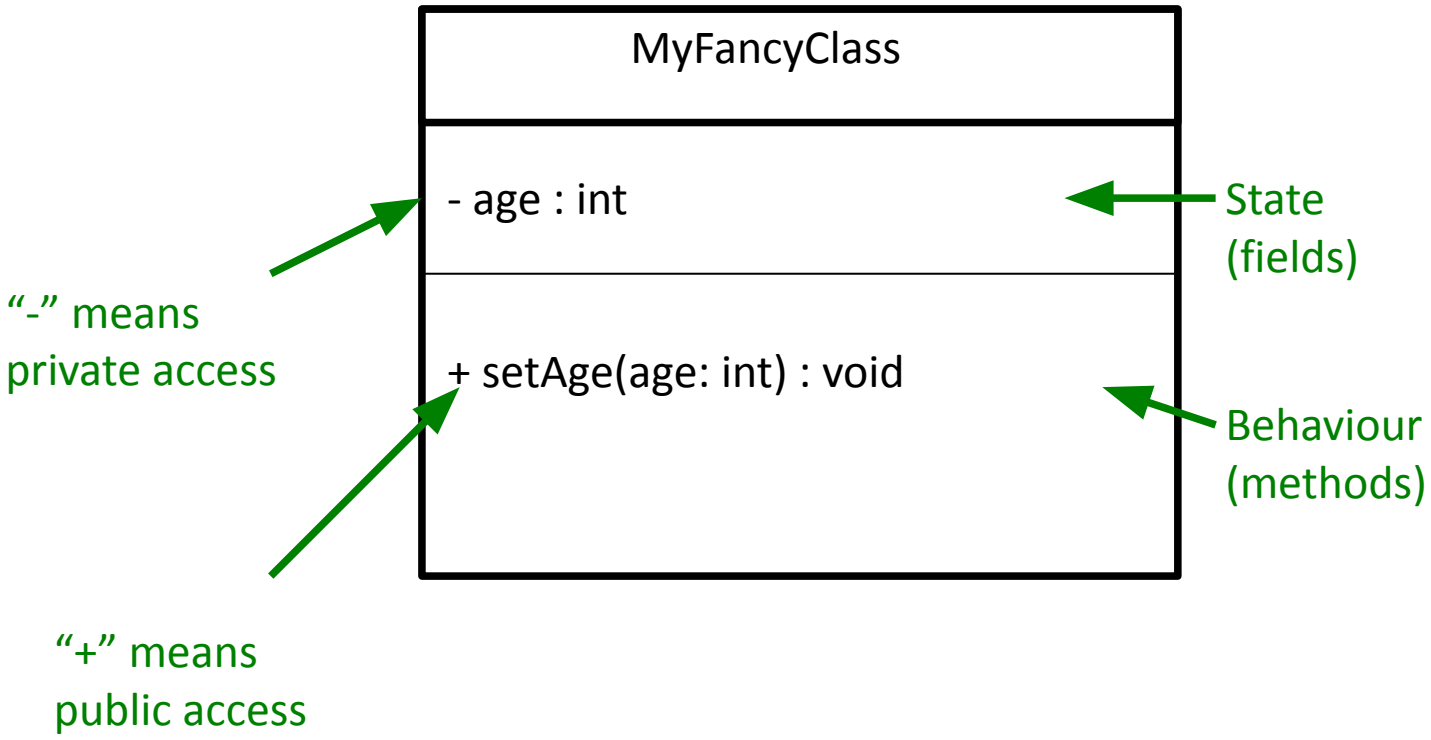
Correct Classes

- We want our class to be a grouping of **conceptually-related** state and behaviour
- One popular way to group is using grammar
  - Noun → Object
  - Verb → Method
  - "A simulation of the Earth's orbit around the Sun"

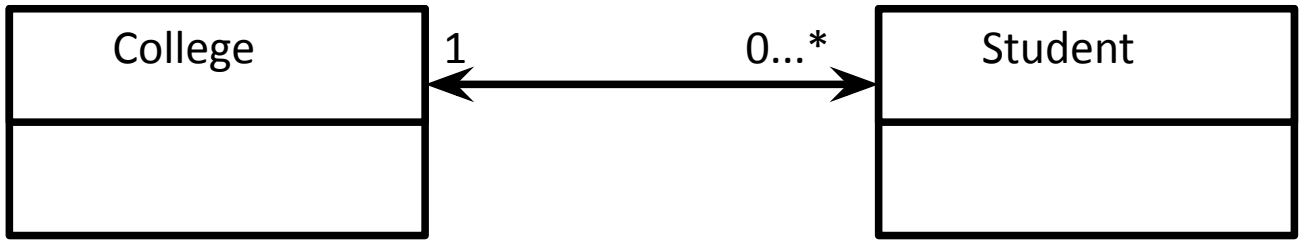
Correct Classes

- We want our class to be a grouping of **conceptually-related** state and behaviour
- One popular way to group is using grammar
  - Noun → Object
  - Verb → Method
  - "A simulation of the Earth's orbit around the Sun"

UML: Representing a Class Graphically

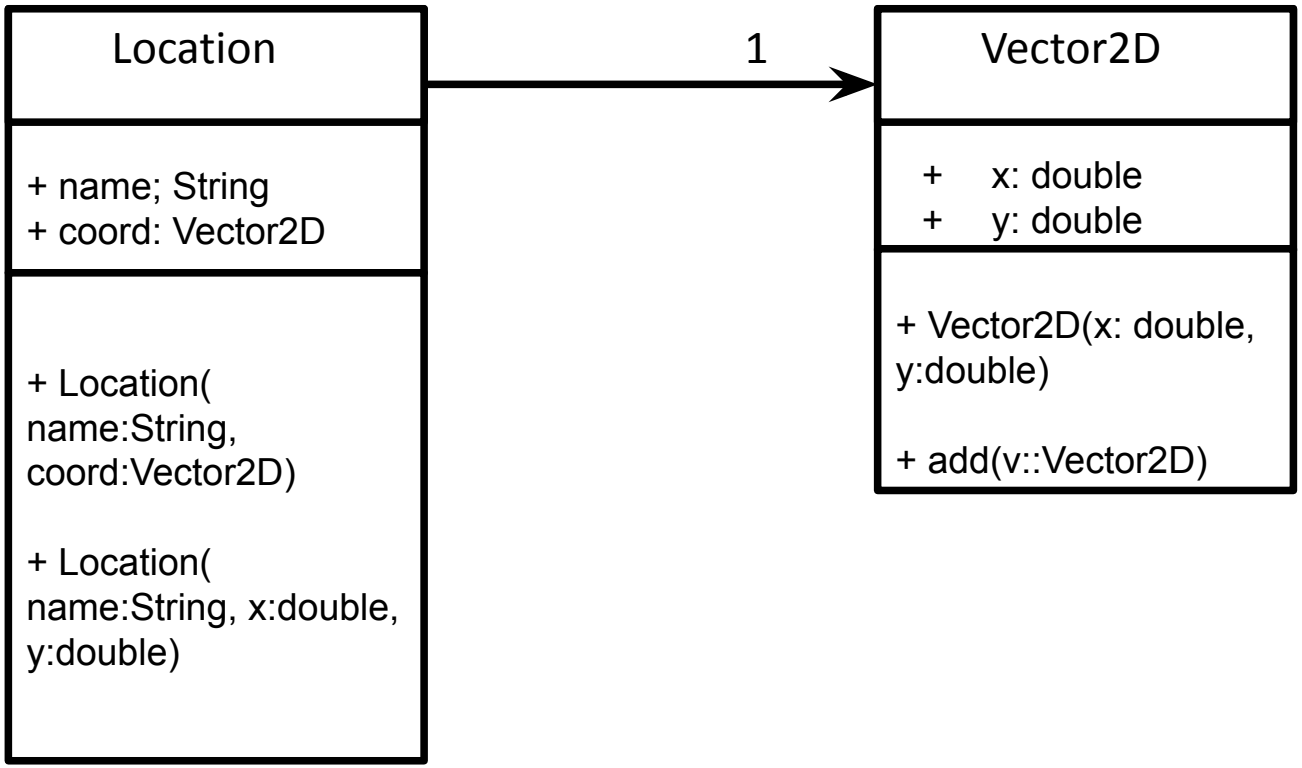


The has-a Association



- Arrow going left to right says "a College has zero or more students"
- Arrow going right to left says "a Student has exactly 1 College"
- What it means in real terms is that the College class will contain a variable that somehow links to a set of Student objects, and a Student will have a variable that references a College object.
- Note that we are only linking *classes*: we don't start drawing arrows between or to primitive types.

Example from code



SRP and Cohesion

Modularity and Code reuse

- You've long been taught to break down complex problems into more tractable sub-problems.
- Each class represents a sub-unit of code that (if written well) can be **developed**, **tested** and **updated** independently from the rest of the code.
- Indeed, two classes that achieve the same thing (but perhaps do it in different ways) can be swapped in the code
- Properly developed classes can be used in other programs without modification.
- Java also has the notion of **packages** to group together classes that are conceptually linked

How do we maximise the chance our classes are reused?

Single Responsibility Principle (SRP)

A class has responsibility over a single functionality.

There is only one single reason for a class to change

SRP Violation

```
public class CompanyIncomeStatement {
      public GUI gui;
      void drawSummaryOnScreen() {
      }
      double calculateOperatingIncome() {
      }
}
```

Combines **Model** concern (operating income) and **View** concern (draw on screen)

I may want the functionality around income in my program, but have no interest in using the graphical display of it. So either I import loads of dead code or i don't use your class…

Aside: Model-View-Controller

A lot of interactive programs are designed around the Model-View-Controller (MVC) concept

The idea is you keep very clear boundaries between:

Model: The code that stores and manipulates the underlying state/data

View: The code that deals with how to draw the state to screen

Controller: The code that sequences everything together, handling input such as clicks, updating the view code when it needs to be updated.

SRP Benefits

1. The class is **easier to understand** because there's only a small number of self explanatory methods and fields

2. The class is **easier to maintain** because changes are isolated

3. **Easier to re-use** because it doesn't contain unnecessary responsibilities

Cohesion

- How to reason about the quality of your code?
- **Cohesion** measures how strongly grouped the responsibilities of a class are
- Code easier for others to locate, understand and use

In other words, **how related are the methods compared to the intention of the class?**

Good Ways to Get Cohesion

- **Functional**
  - methods are grouped because solving a defined task
  - e.g. CSV parsing
- **Informational**
  - methods are grouped because worked on a same domain object
  - e.g. maths libraries
- **Sequential**
  - outputs of methods becomes input of other methods
  - e.g. text processing pipeline

Less Good Ways to Get Cohesion

- **Logical**
  - grouping functionalities that sound like it fits in a similar category but are different (e.g. grouping XML and JSON parsing)
- **Utility classes**
  - multiple different concerns
- **Temporal/Procedural**
  - parts of a module are grouped because follow an execution pattern/particular time (e.g. data validation, data storage, notification)

Cohesion Summary

Level of Cohesion	Pro	Con
Functional (high cohesion)	Easy to understand	Can lead to overly simplistic classes
Informational (medium cohesion)	Easy to maintain	Can lead to unnecessary dependencies
Sequential (medium cohesion)	Easy to locate related operations	Encourage violation of SRP
Logical (medium cohesion)	Provide a form of high level categorisation	Encourage violation of SRP
Utility (low cohesion)	Simple to put in place	Harder to reason about the responsibility of the class
Temporal (low cohesion)	-	Harder to understand and use individual operations

OOP Principle 1: Encapsulation

Core Idea

We want to provide an appropriate API to the object that only exposes what it needs to

Supporting encapsulation means supporting tools to **hide** things away that a user of the class shouldn't need to even know about

Encapsulation Example

```
public class Student {
   public int age;

  public static void main(String[] args) {
     Student s = new Student();
     s.age = 21;

     Student s2 = new Student();
     s2.age=-1;

     Student s3 = new Student();
     s3.age=10055;
  }
}
```

By exposing the age field, we can't stop something stupid being set there

Encapsulation Example

```
public class Student {
  private int age;

  public boolean setAge(int a) {
    if (a>=0 && a<130) {
     age=a;
     return true;
    }
    return false;
  }

  public int getAge() {return age;}

  public static void main(String[] args) {
   Student s = new Student();
   s.setAge(21);
  }
}
```

By hiding the age away (making it private), we prevent this problem

To make it useful we provide a method to set age that incorporates sanity checks

Encapsulation

Encapsulation allows us to decouple the API (set of methods an object supports) from the underlying state so we can e.g. change that implementation

```
class Location {
  private float x;
  private float y;

  float getX() {return x;}
  float getY() {return y;}

  void setX(float nx) {x=nx;}
  void setY(float ny) {y=ny;}
}
```

```
class Location {

  private Vector2D v;

  float getX() {return v.getX();}
  float getY() {return v.getY();}

  void setX(float nx) {v.setX(nx);}
  void setY(float ny) {v.setY(ny);}
}
```

Access Modifiers

	Can be accessed by			
Modifier	**Class**	**Package**	**Subclass**	**Everyone**
`public`	✅	✅	✅	✅
`protected`	✅	✅	✅	❌
no modifier	✅	✅	❌	❌
`private`	✅	❌	❌	❌

Aka Information Hiding

Another name for encapsulation is **information hiding** or even implementation hiding in some texts.

Idea: Classes expose a clean interface that allows full interaction with it, but which exposes nothing about its internal state or how it manipulates it.

Remember, you can always make a private member public, but not vice-versa!

Immutability

Immutability

M**utable objects** (i.e. those where their state can be changed during execution) can be very powerful,, but they also increase complexity and can be a **common source of bugs**

Our access modifiers give us a way to make an **immutable class,** whereby you can set the internal data on initialisation but not change it…

Immutable class definition

```
public class Vector2D {
     private final int x;
     private final int y;

     public Vector2D(int x,
                     int y) {
         this.x = x;
         this.y = y;
     }

     public int getX() {
       return x;}

     public int getY() {
       return y;}

}
```

`private` means no one using the class can directly get at the data

`final` says the value can't be changed once it has been set (it's a belt-and-braces thing here)

Not providing any setters means anyone using the class can't set anything

The constructor allows the values to be set

*(You can also make the **class** final: this signals intent and prevents a world where you subclass and provide access through the subclass)*

Although…

Java 16 introduced a **`record`** type which generates this:

```
public class Vector2D {
     private final int x;
     private final int y;

     public Vector2D(int x, int y) {
         this.x = x;
         this.y = y;
     }

     public int getX() {return x;}

     public int getY() {return y;}
}
```

From this:

```
public record Vector2D(int x, int y) {}
```

(actually, it generates more for you: equals(), hashcode(), etc. We just haven't got to that yet)

Creating an immutable class

- Make all fields `final`, avoid `setXXX` methods
- Or, in Java 16+, use `record` types
- Optionally, provide change-factory methods
  Common but not universal to use `withXXX` names for these. E.g

`String.toUpper(),LocalDate.withYear(2022)`

Immutable benefits

1. Reduce the scope for bugs
2. Can be thread-safe (more on this next year!)
3. Easier to reason about

Immutable classes are everywhere in the JDK

- `Integer, Double, BigDecimal...`
- `String`
- `LocalDate, LocalTime …`
- `UUID`
- `Optional`
- `Enums (usually & idiomatically)`
  - `Enums can compare with == instead of .equals()`

Memory, pointers, references

Primitives, References and Memory

Primitives

What is the value of a variable?

For **primitive** types (int etc) it may seem obvious: the value is whatever it is e.g. 1, 2.0, etc

In memory, primitives are stored directly (we'll see how shortly)

```
int amount = 100
```

Objects

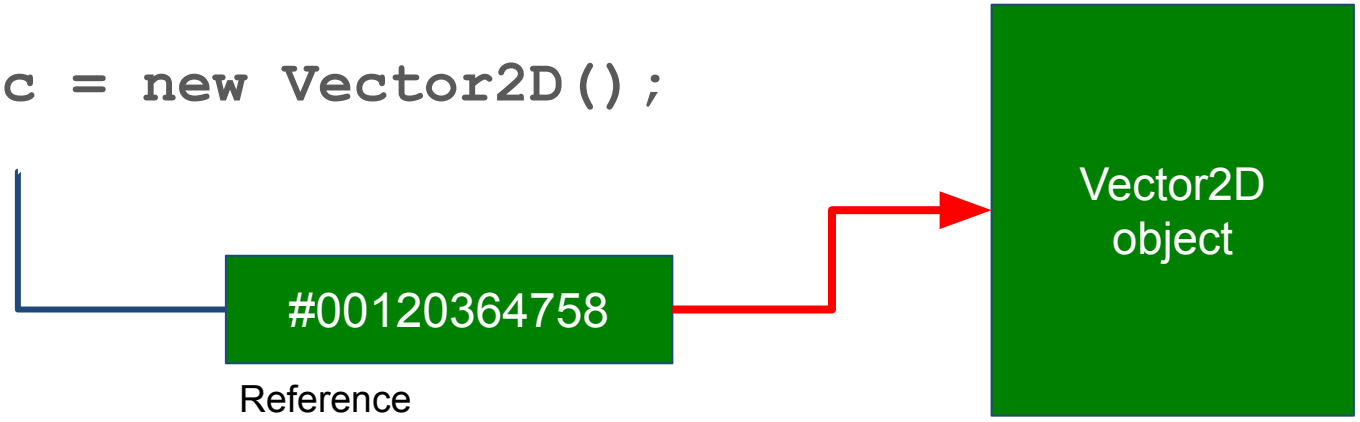
What is the value of a variable?

For an **object** (a custom types, Vector2D etc) the value of the associated variable is a **memory address** for where to find all the data associated with that object.

We say that the value of the variable is a **reference**

We'll see why shortly, but this gives some interesting results…

```
Vector2D vec = new Vector2D();
```



Values are copied on assignment

Values are either primitives or references. They are copied on assignment.

```
int amount = 100;
```


```
int amountCopy = amount;
```

Values are copied on assignment

Values are either primitives or references. They are copied on assignment.

```
int amount = 100;
```

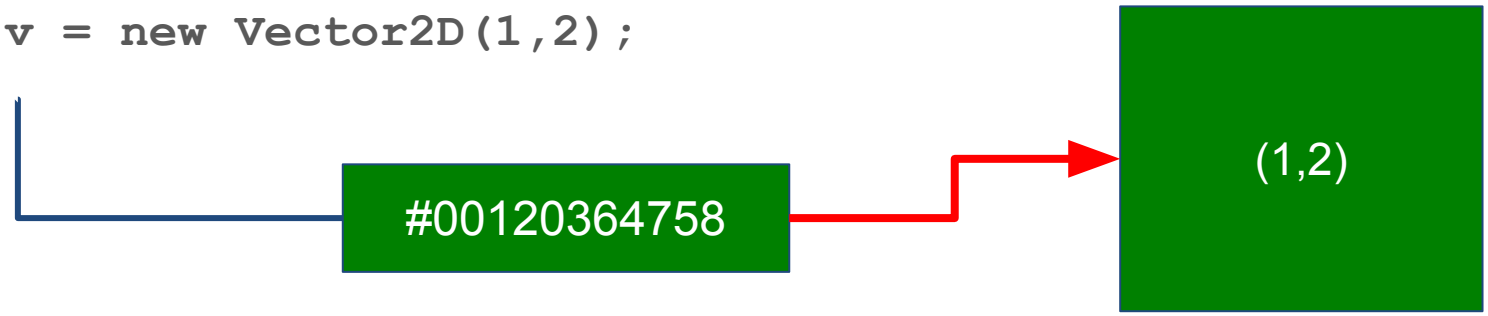

```
int amountCopy = amount;
```


```
amountCopy = 200;
```

Values are copied on assignment

Values that are references also copy **BUT NOT THEREFORE THE OBJECTS**

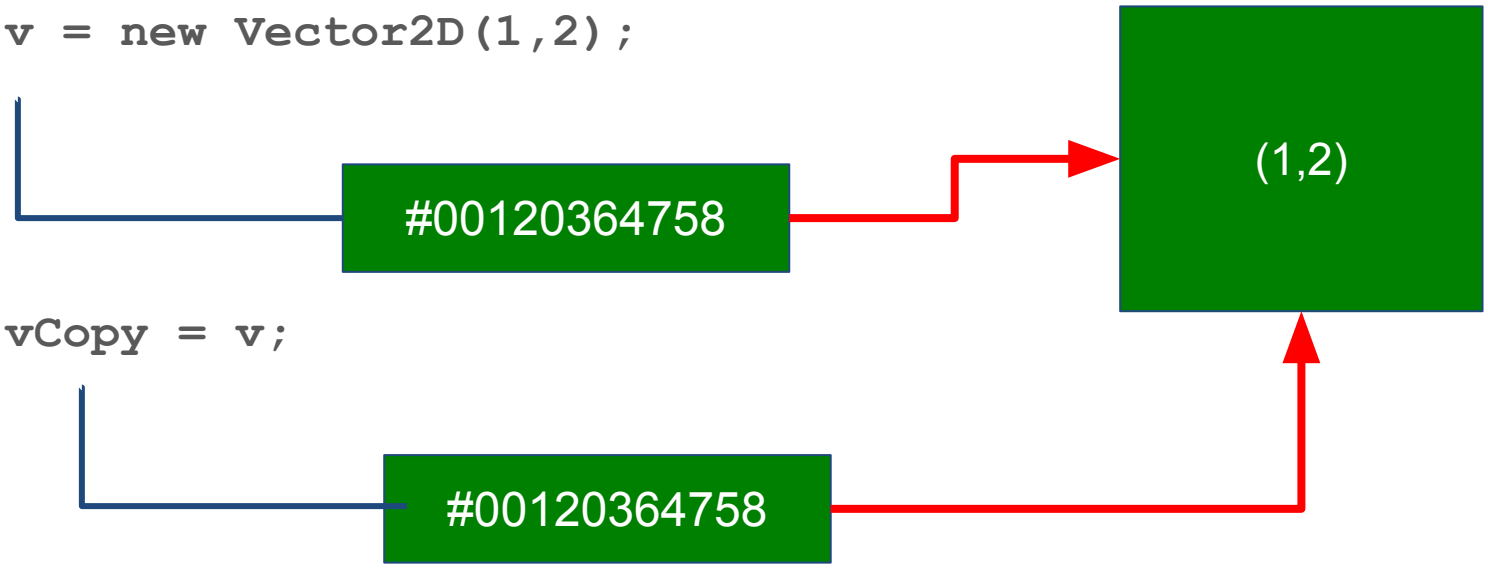
```
Vector2D v = new Vector2D(1,2);
```



Values are copied on assignment

Values that are references also copy **BUT NOT THEREFORE THE OBJECTS**

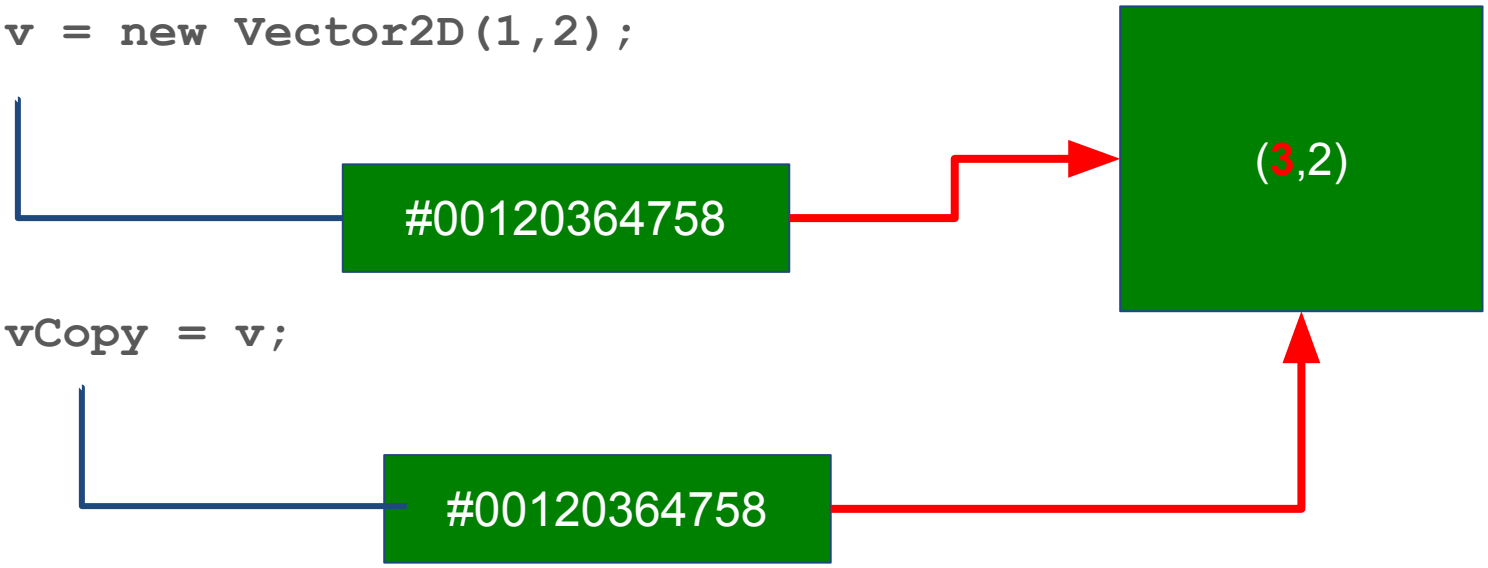
```
Vector2D v = new Vector2D(1,2);
```



Values are copied on assignment

Values that are references also copy **BUT NOT THEREFORE THE OBJECTS**

```
Vector2D v = new Vector2D(1,2);
```



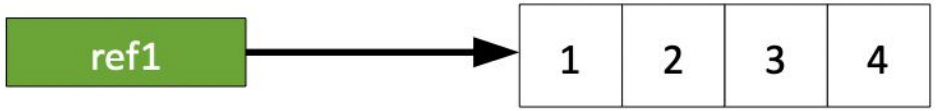

```
Vector2D vCopy = v;
```

```
vCopy.setX(3);
```

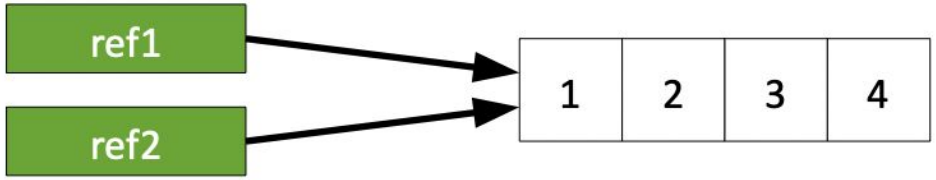
Example

```
int[] ref1 = null;
```

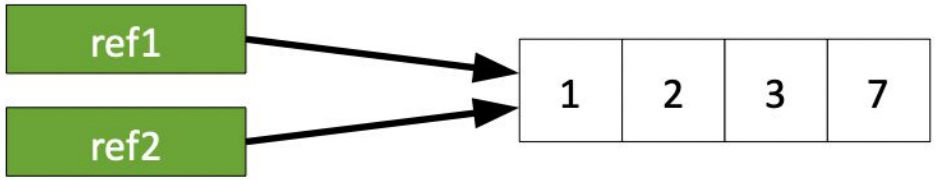

```
ref1 = new int[]{1,2,3,4};
```



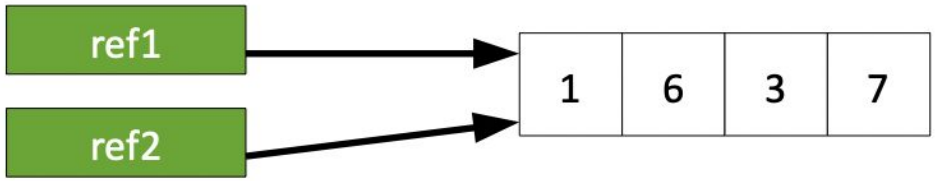

```
int[] ref2 = ref1;
```




```
ref1[3]=7;
```




```
ref2[1]=6;
```



Where stuff goes: The stack and heap abstractions

Memory

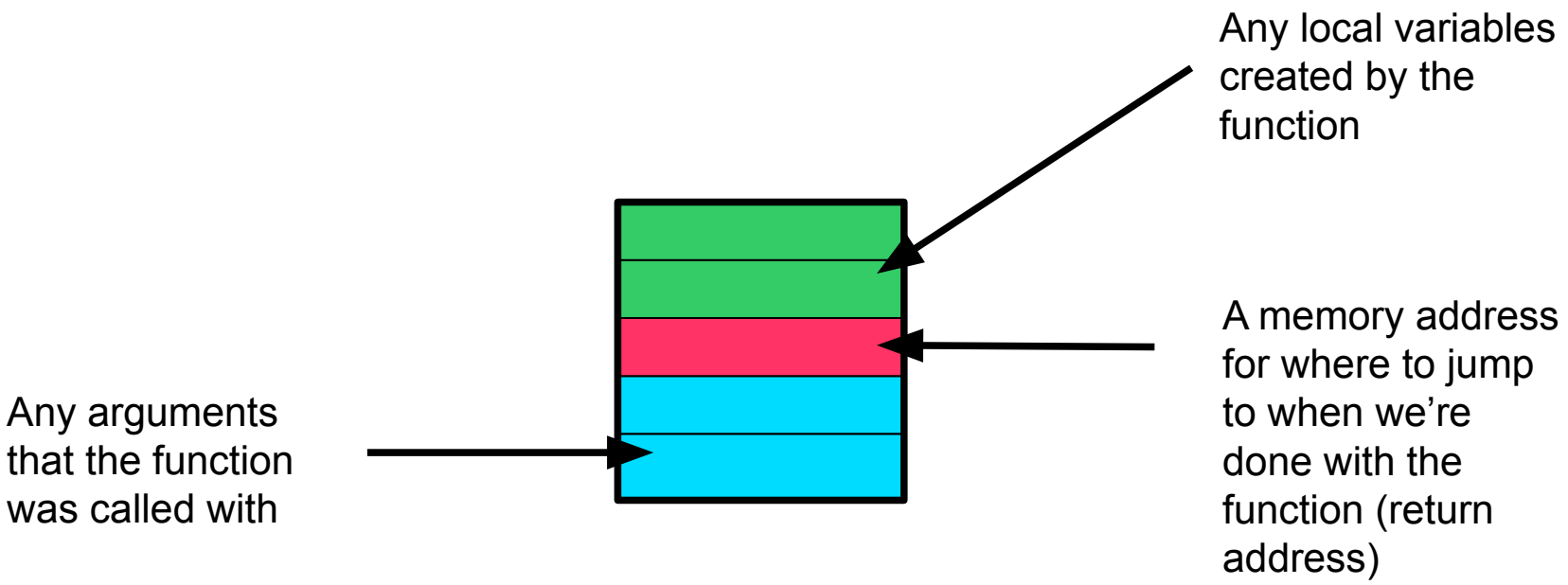
How do we organise all the code and data for our program in memory?

This is actually really complex (and covered in Operating Systems) but there is an intermediate abstraction that gives programmers a mental model that is widely used (beyond Java)

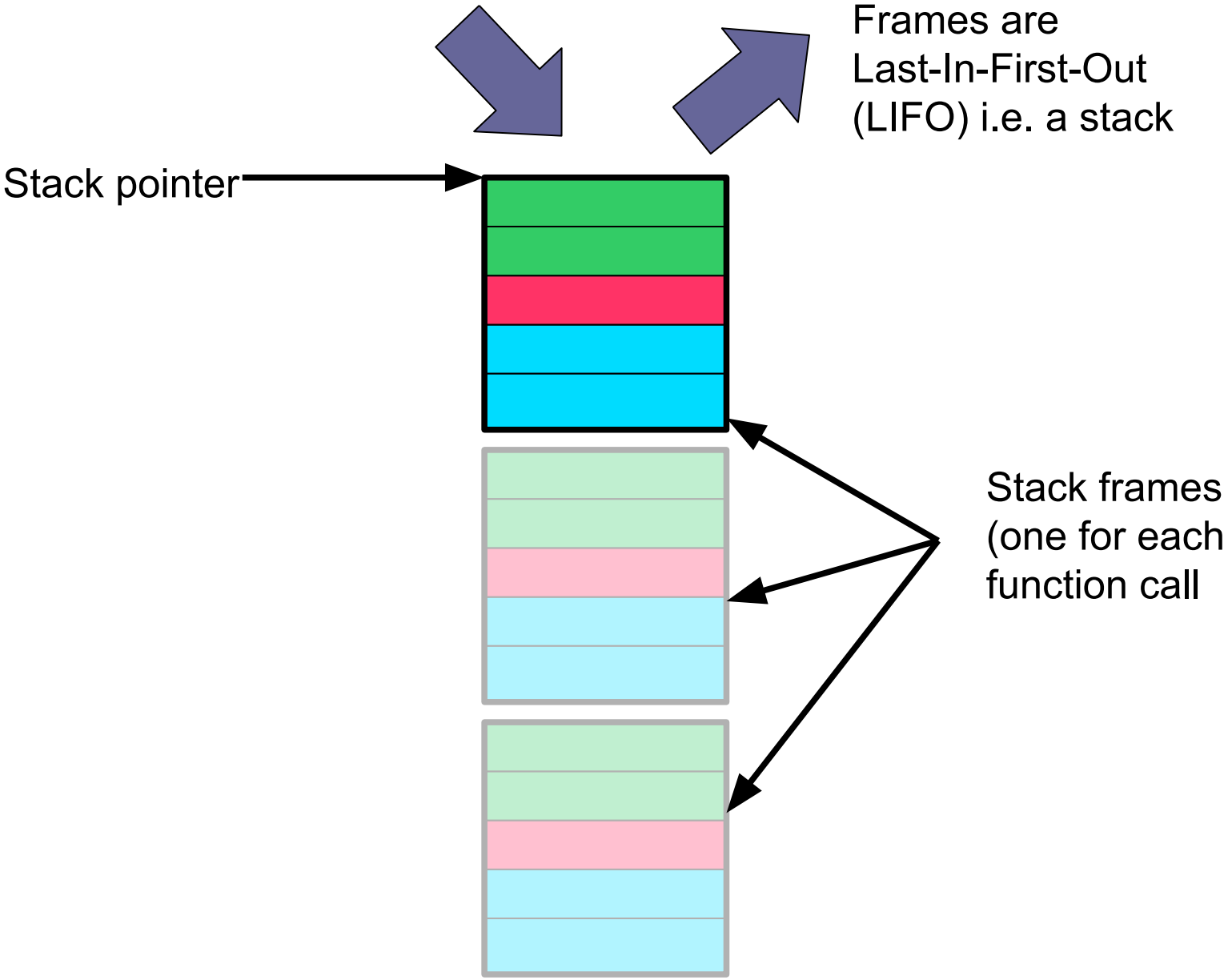
Let's look at what happens when you call a function/method/procedure/…

Simple visualisation of a function's data

When we call a function/method, we need to store three things:

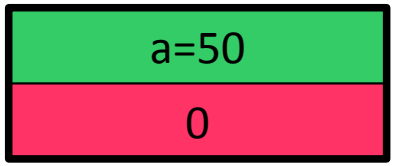


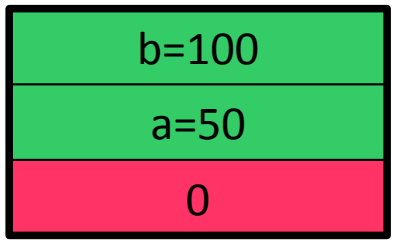
The Call Stack

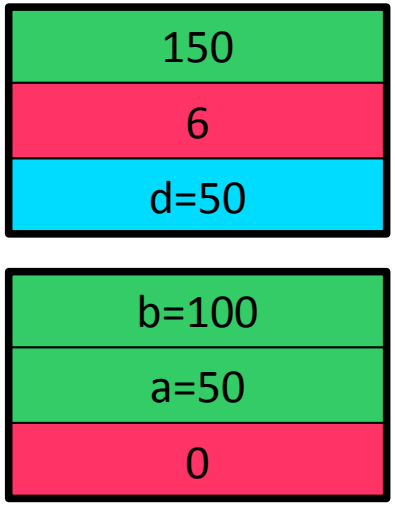


The Call Stack: Example

```
1    int twice(int d) return 2*d;
2    int triple(int d) return 3*d;
3    int a = 50;
4    int b = twice(a);
5    int c = triple(a);
6    ...
```

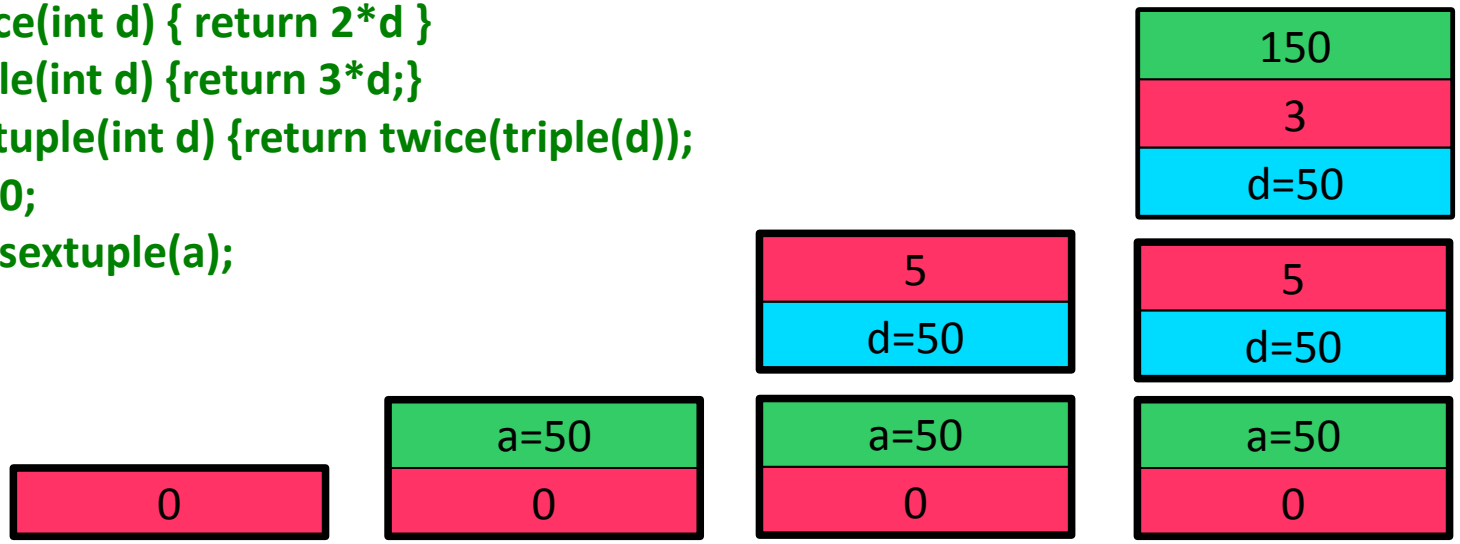


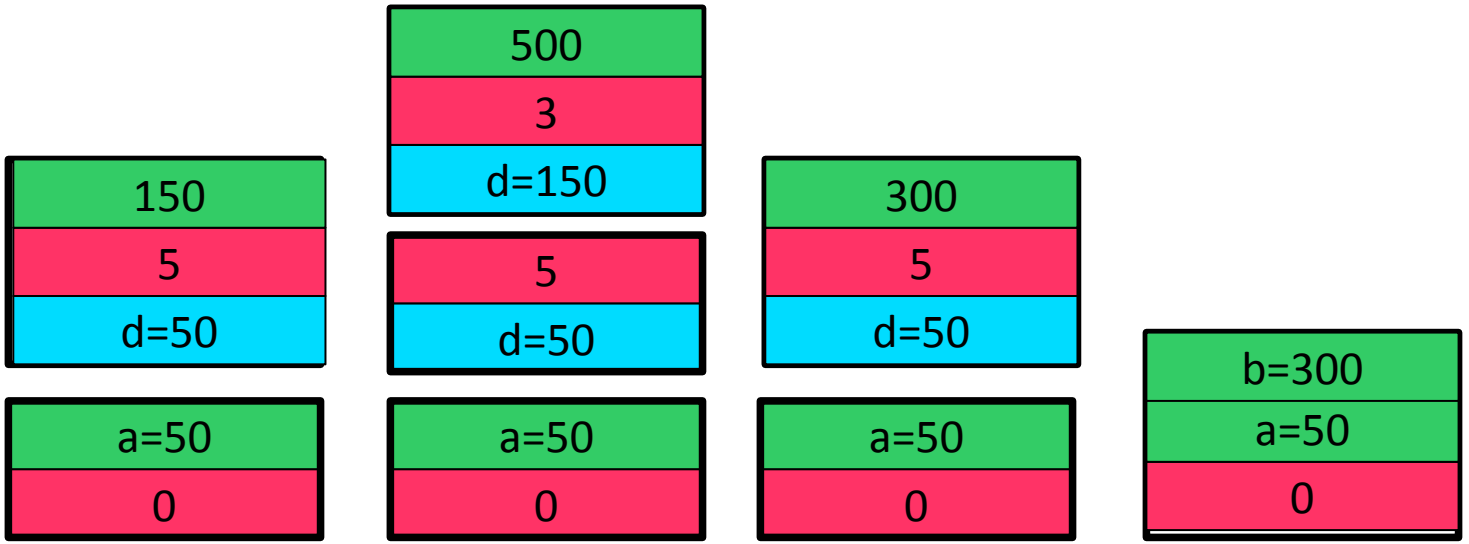




Nested Functions

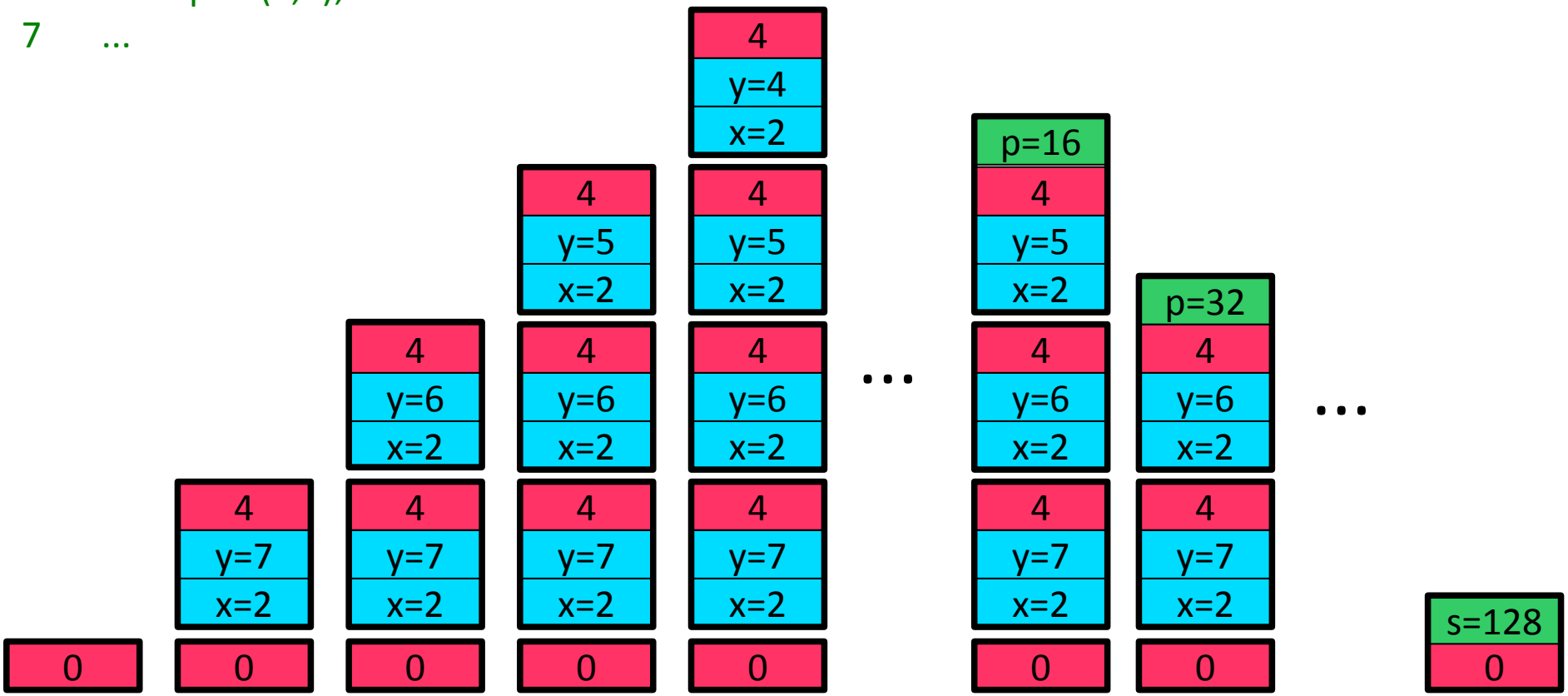
```
1    int twice(int d) { return 2*d }
2    int triple(int d) {return 3*d;}
3    int sextuple(int d) {return twice(triple(d));
4    int a=50;
5    int b = sextuple(a);
6    ...
```





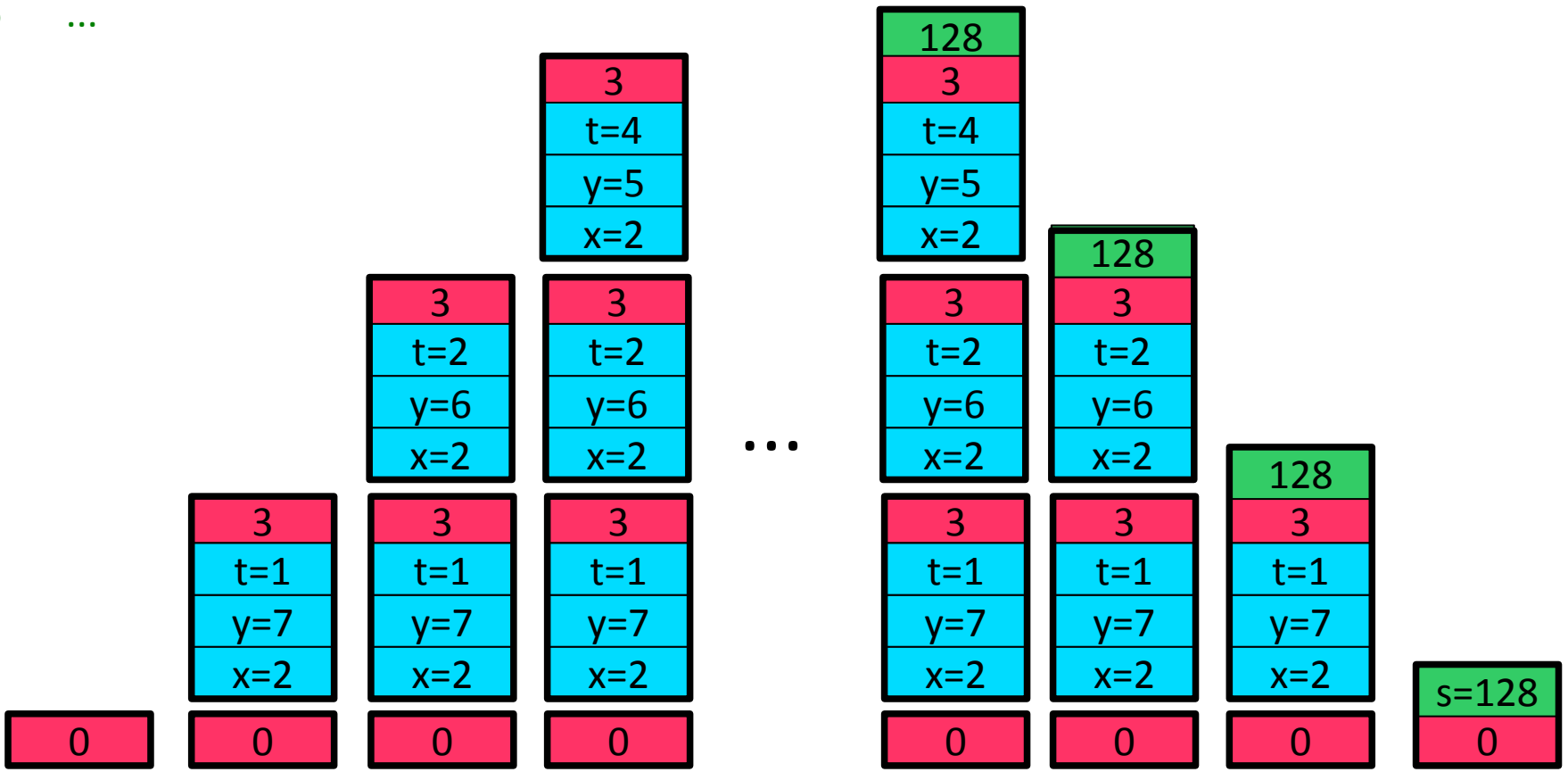
Recursive Functions

```
1    int pow (int x, int y) {
2        if (y==0) return 1;
3        int p = pow(x,y-1);
4        return x*p;
5    }
6    int s=pow(2,7);
7    ...
```



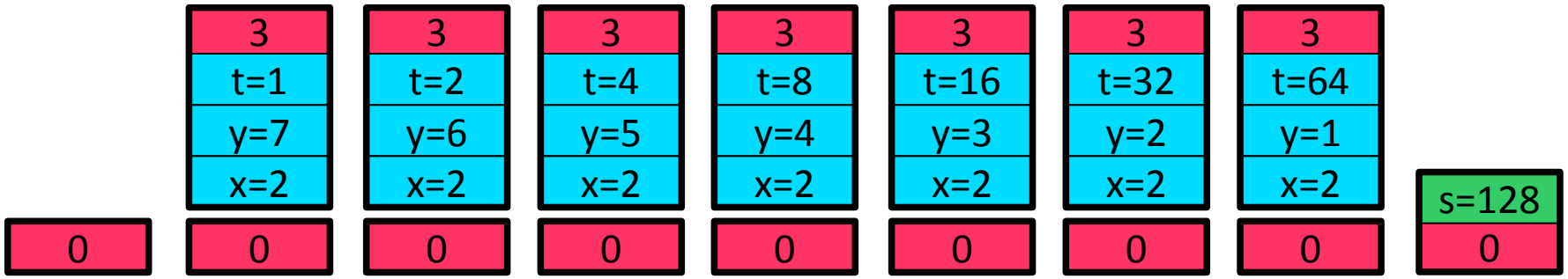
Tail-Recursive Functions I

```
1    int pow (int x, int y, int t) {
2        if (y==0) return t;
3        return pow(x,y-1, t*x);
4    }
5    int s = pow(2,7,1);
6    ...
```



Tail-Recursive Functions II

```
1    int pow (int x, int y, int t) {
2        if (y==0) return t;
3        return pow(x,y-1, t*x);
4    }
5    int s = pow(2,7,1);
6    ...
```



Scope

Note that when a stack frame is ‘popped’ (the function has finished), all the variables it created will be deleted.

We talk about those variables having ‘local **scope’**.

Static variables have global scope

Member variables/fields have the scope defined by their owning object

Objects aren't primitives

So the call stack keeps track of our function calls and variables and it does so efficiently (no wasted space)

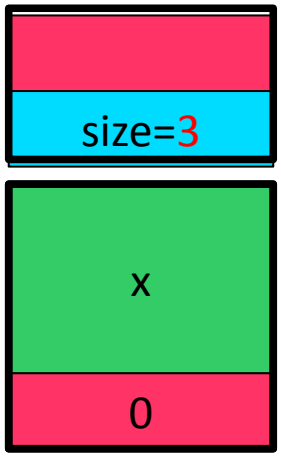
But what happens if something on the stack changes size? This would mess things up considerably (you'd have to move everything and update return addresses…yuk)

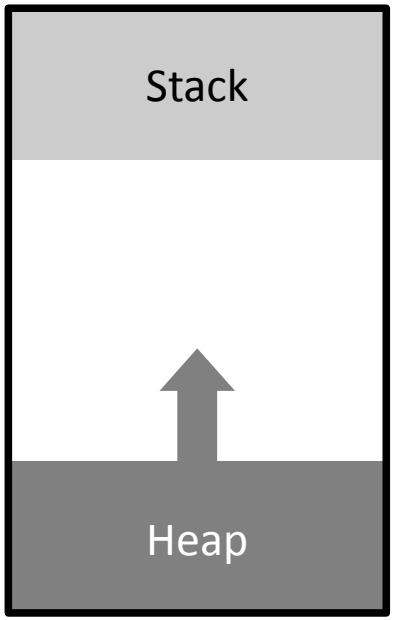
Primitives are good here: no size changes possible.

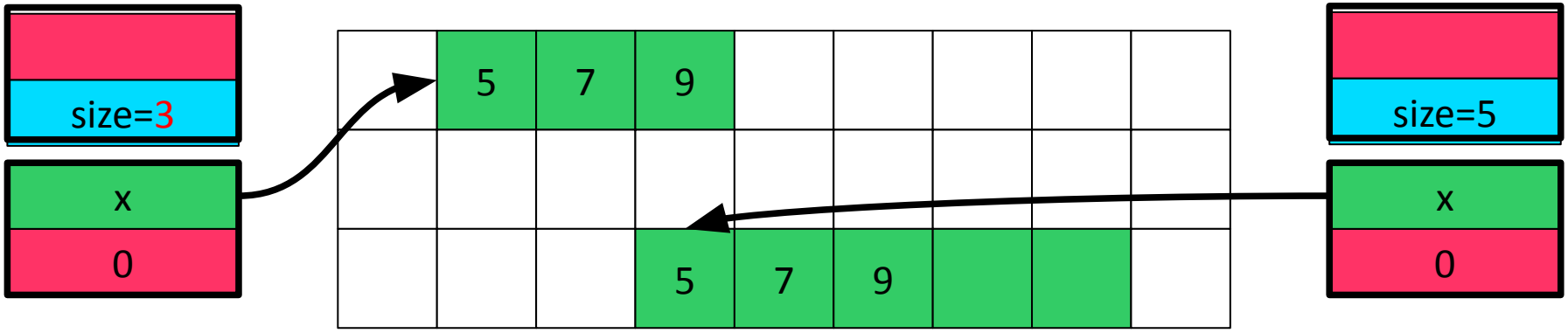
For objects, we store ***references*** on the stack (which are just memory addresses which are known size numbers) and make them point to a different section of memory…

The Heap

```
int[] x = new int[3];
public void resize(int size) {
    int tmp=x;
    x=new int[size];
    for (int=0; i<3; i++)
        x[i]=tmp[i];
}
resize(5);
```







The Heap

The heap is (as its name suggests) more of a mess. There will be gaps between objects.

But it gives us the flexibility to do what we need to do

Pointers vs References

Pointers

You may have come across pointers in other languages (e.g. C, C++, etc). They are variables that hold integer values that are interpreted as memory addresses.

Thus these are the same concept as a reference, but are a bit more ‘raw’...

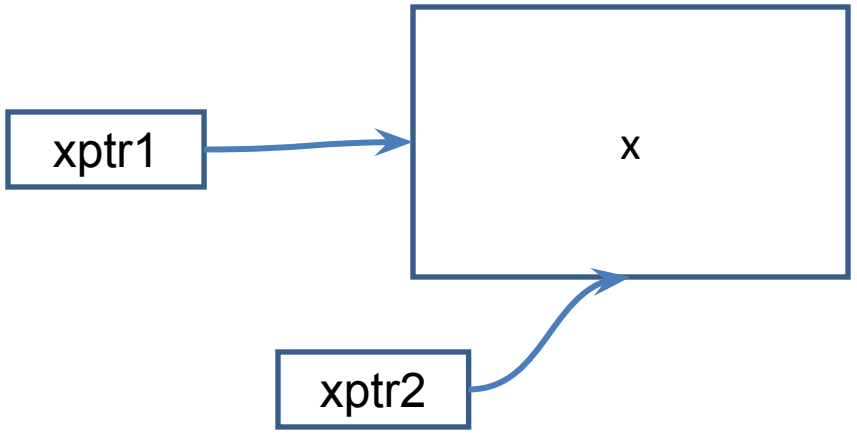
Manipulating memory directly allows us to write fast, efficient code, but also exposes us to bigger risks

- Get it wrong and the program 'crashes' .

Pointers: Box and Arrow Model

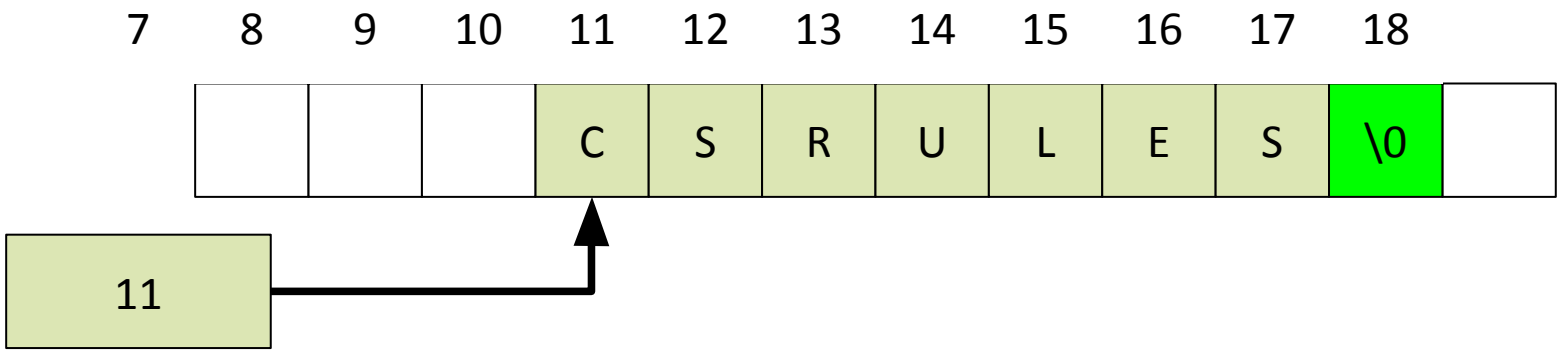
- A pointer is just the memory address of the first memory slot used by the variable
- The pointer type tells the compiler how many slots the whole object uses

```
// C++
int x = 72;
int *xptr1 = &x;
int *xptr2 = xptr1;
```



Example: Representing Strings I

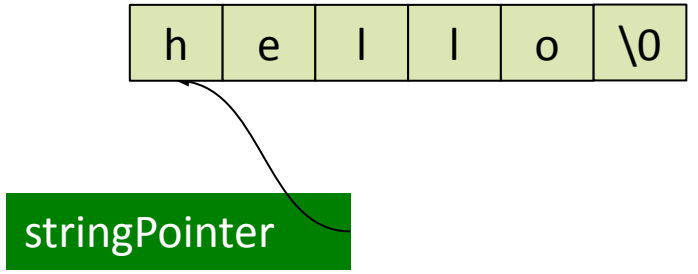
- A single character is fine, but a text string is of variable length – how can we cope with that?
- We simply store the start of the string in memory and require it to finish with a special character (the NULL or terminating character, aka '\0')
- So now we need to be able to store memory addresses → use pointers



- We think of there being an array of characters (single letters) in memory, with the string pointer pointing to the first element of that array

Example: Representing Strings II

```
char letterArray[] = {'h','e','l','l','o','\0'};

char *stringPointer = &(letterArray[0]);

printf("%s\n",stringPointer);

letterArray[3]='\0';

printf("%s\n",stringPointer);
```



References

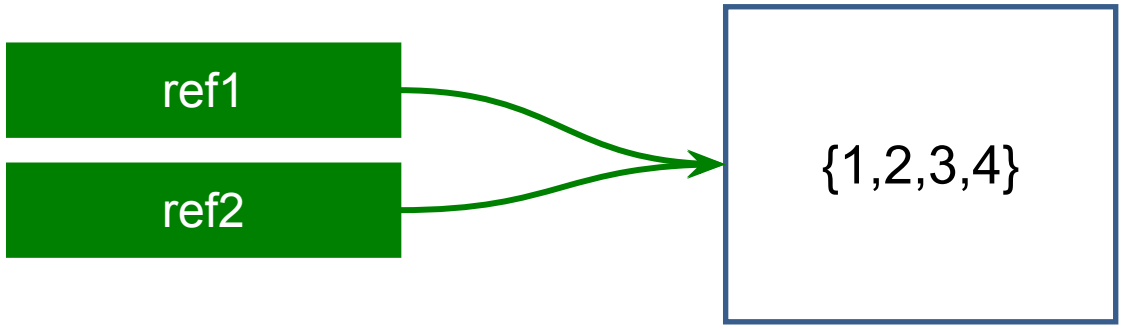
- Pointers are useful but dangerous
- References can be thought of as restricted pointers
  - Still just a memory address
  - But the compiler limits what we can do to it
- C, C++: pointers *and* references
- Java: references *only*
- ML: references *only*

References vs Pointers

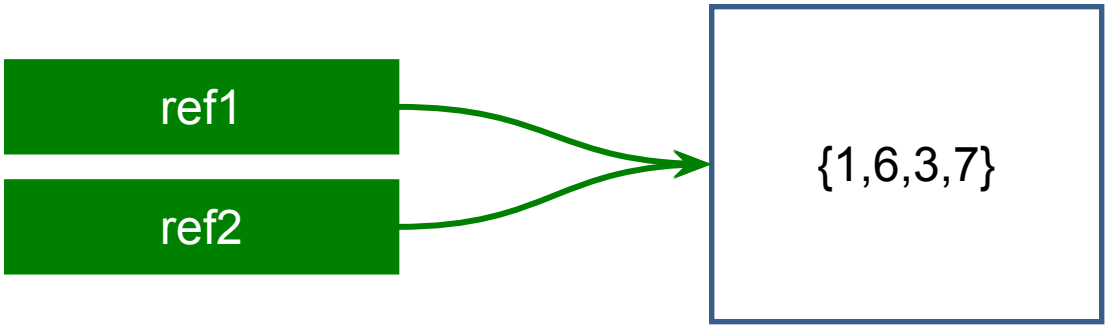
	Pointers	References
Represents a memory address	Yes	Yes
Can be arbitrarily assigned	Yes	**No**
Can be assigned to established object	Yes	Yes
Can be tested for validity	**No**	Yes

References Example (Java)

```
int[] ref1 = null;
ref1 = new int[]{1,2,3,4};
int[] ref2 = ref1;
```




```
ref1[3]=7;
ref2[1]=6;
```



Why not have references to primitives?

A reference is just a memory address - typically a `long`

If we referred to all primitives using references, we'd be doubling our memory usage (reference size ~= primitive size). Object sizes are typically >> reference size

But there are cases where you might want to reference a primitive directly…

Pass-by-value

```
void changeVal(int x) {
   x = 2*x;
}
```

…

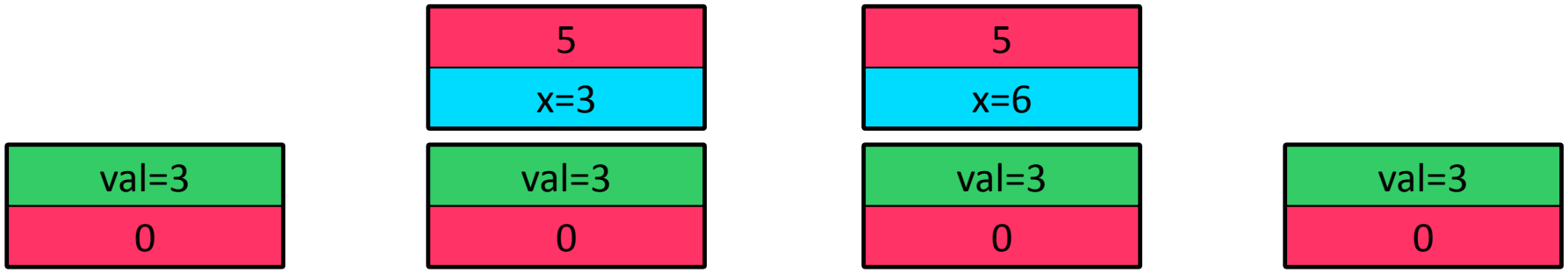
```
int val = 3;
changeVal(val);
// val is 3 still
```

When we call changeVal with argument val, we **copy** the value into the stack frame.

We can change that copy's value, but it won't affect the original

This approach of copying is known as **pass-by-value, and it's all Java offers**

Pass-by-value



x is a copy of val

Pass-by-value

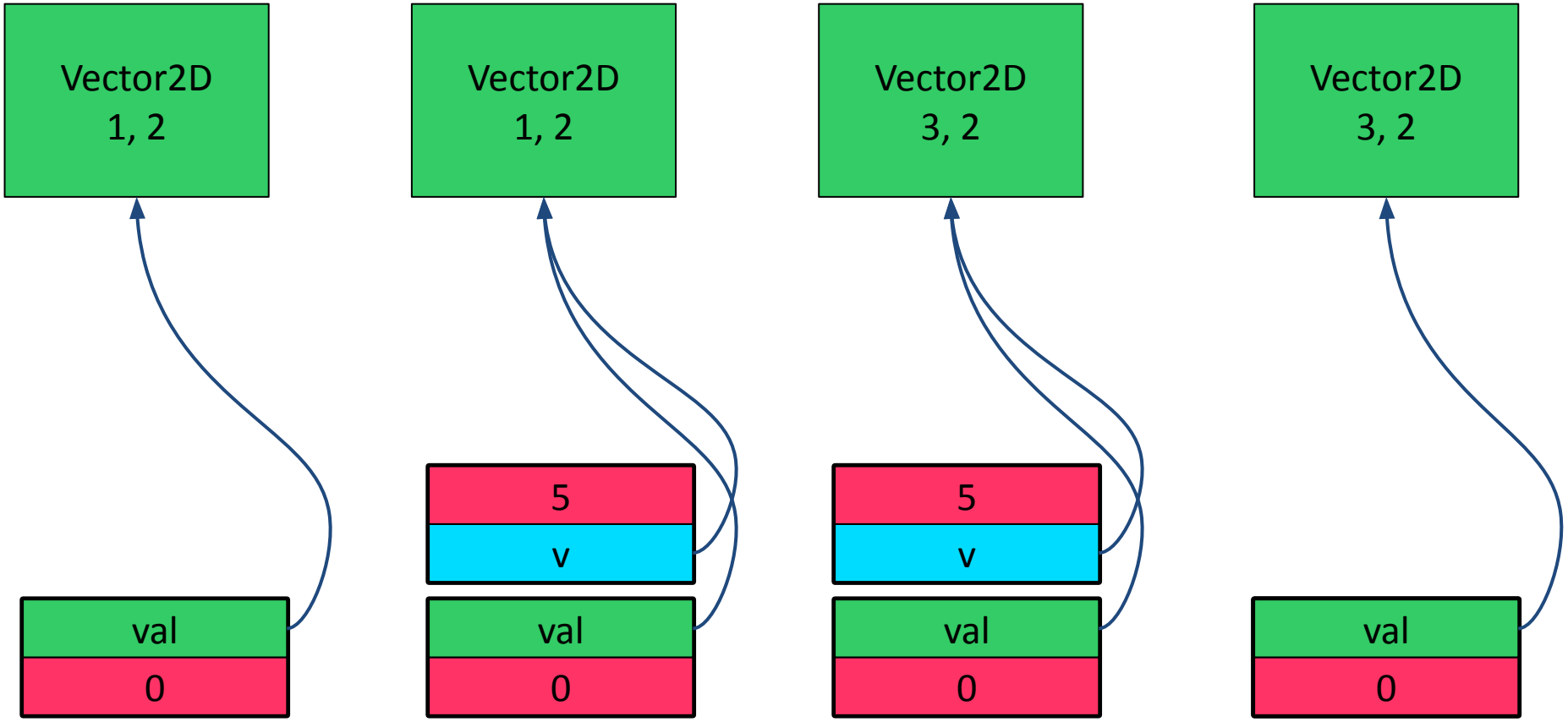
```
void changeVal(vector2D v) {
   v.setX(3);
}

...

Vector2D val = new
  Vector2D(1,2);
changeVal(val);
```

Java applies pass by value to references too, meaning it takes the reference value (a memory address) and copies it into the stack frame. Thus `v` is a reference to the same object as `val`.

Because v points to the same object, you can make changes to the object by accessing the reference

Again, All Java offers is pass by value (but for references the value is a memory address)

Pass-by-value



v is a copy of val and therefore the same memory address

Pass-by-reference (**not** Java)

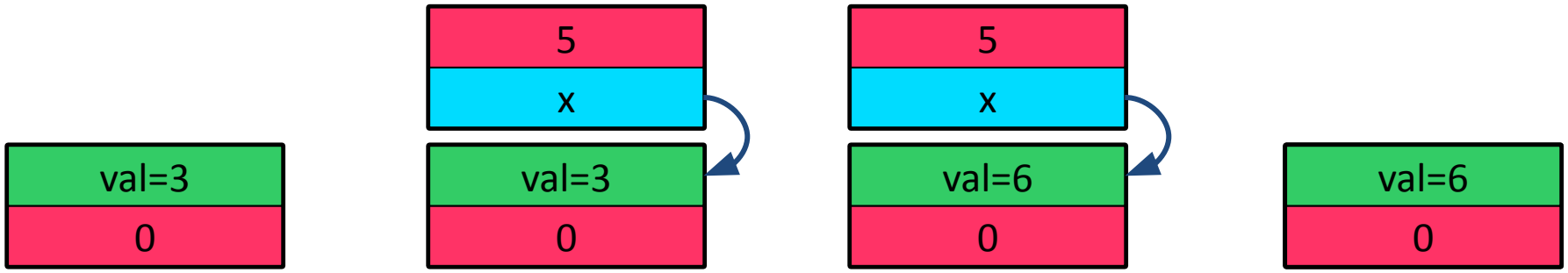
```
void changeVal(int &x) {
   x = 2*x;
}

...

int val = 3;
changeVal(val);
// val is 6!
```

The example is C++, where the & symbol means to get the memory address (pointer) to the thing that is offered

So in this example x is a reference to val and the update actually works

Java does not support this.

Pass-by-reference (**not** Java)



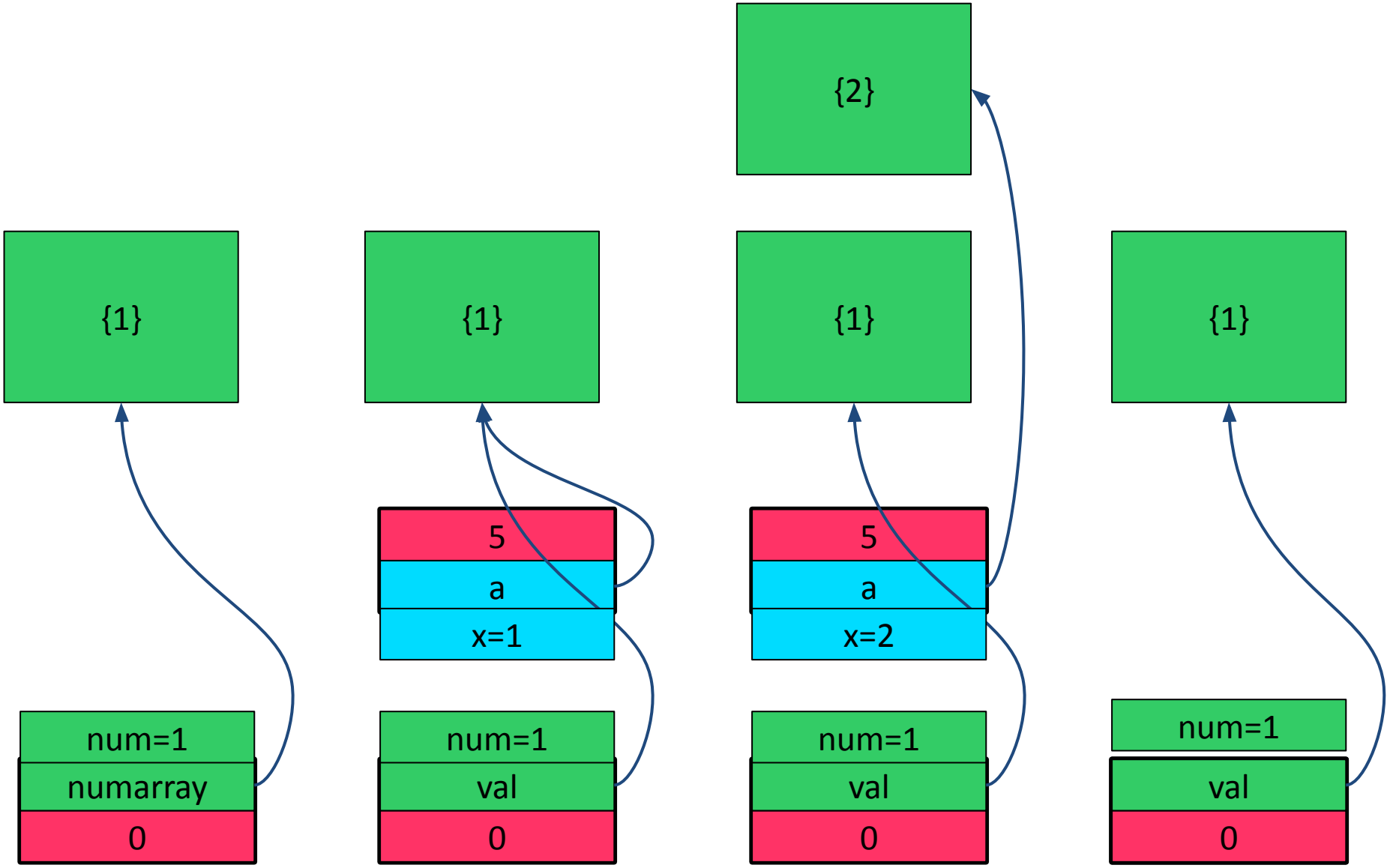
x is a reference
to val

Check...

```
public static void func(int x, int[] a) {
        x=1;
        x=x+1;
        a = new int[]{1};
        a[0]=a[0]+1;
}

public static void main(String[] arguments) {
        int num=1;
        int numarray[] = {1};

        func(num, numarray);
        System.out.println(num+" "+numarray[0]);
}
```

A. “1 1”
B. “1 2”
C. “2 1”
D. “2 2”

Explanation



Inheritance

Inheritance I

```
class Student {
   public int age;
   public String name;
   public int grade;
}

class Lecturer {
   public int age;
   public String name;
   public int salary;
}
```

- There is a lot of duplication here
- Conceptually there is a hierarchy that we're not really representing
- Both Lecturers and Students are people (no, really).
- We can view each as a kind of specialisation of a general person
  - They have all the properties of a person
  - But they also have some extra stuff specific to them

(I should not have used public variables here, but I did it to keep things simple)

Inheritance II

```
class Person {
   public int age;
   public String name;
}

class Student extends Person {
   public int grade;
}

class Lecturer extends Person {
   public int salary;
}
```

- We create a *base class* (Person) and add a new notion: classes can *inherit* properties from it
  - Both state and functionality
- In java the `extends` keyword is used to inherit from a class
- We say:
  - Person is the *superclass* of Lecturer and Student
  - Lecturer and Student *subclass* Person

Loose terminology

`Person`	`Student`
Parent class	Child class
Superclass	Subclass
Base class	Derived class

What is Inheritance?

When a class inherits from another class it:

- It inherits its **type**. So Lecturer is-a Person.
- incorporates (“inherits”) all the **attributes and behaviours** from that class.
- can **directly** access the public & protected members of that class (but *not* the private)
- can redefine some inherited behaviour, or add new attributes and behaviour.

What gets Inherited?

A subclass inherits from its parent classes:

- Type
- Fields
- Methods

`private` superclass fields cannot be accessed directly* by the subclass

* this is worded carefully. Private superclass fields are within the subclass, but they are not directly accessible. Most people consider this restriction to mean they aren't "inherited". You can still get at them **iff** there are inherited public/private methods that give you access to them

The magic `Object` Class

All classes inherit from `Object` in Java. i.e.

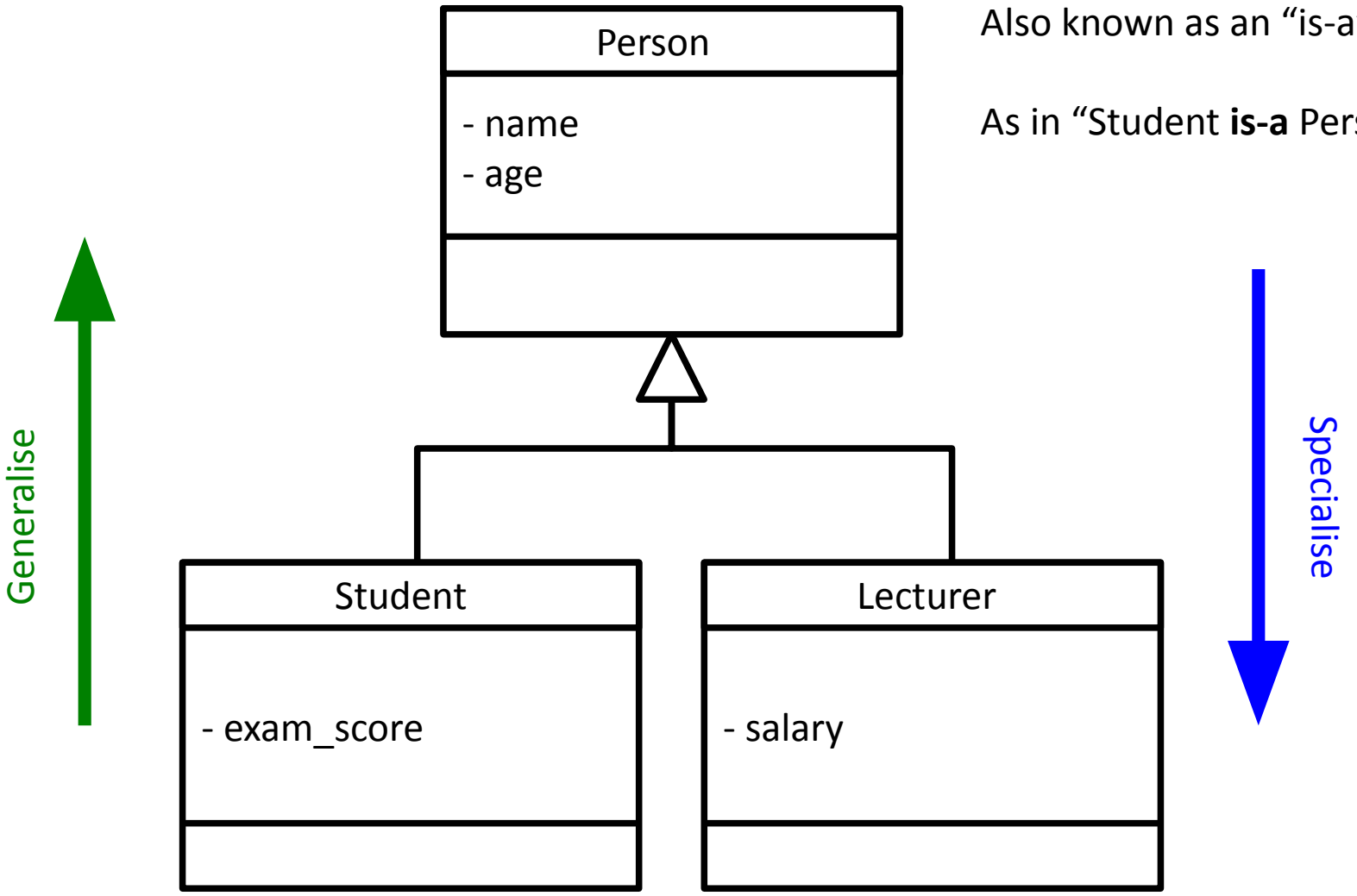
```
public class MyClass
```

→

```
public class MyClass extends Object
```

Because it's always true, we never bother drawing it on UML or writing it in code. But it's there, and we'll look at some of the things it provides later.

Representing Inheritance Graphically



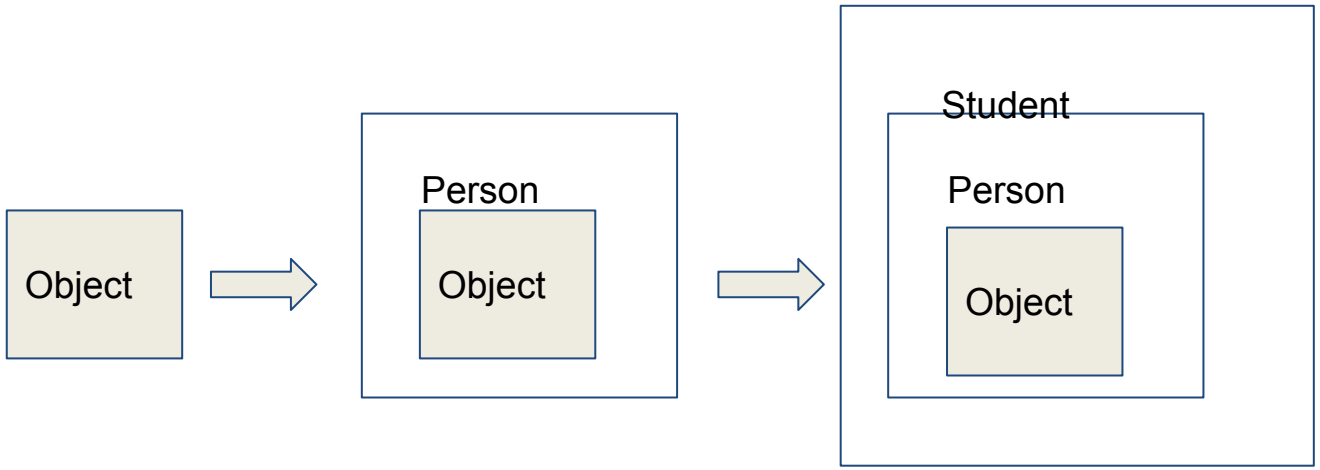
Also known as an “is-a” relation

As in “Student **is-a** Person”

Constructors and Inheritance

To build an object of the `Student` class

- First you have to build the foundation (`Object`)
- Then the 1st subclass (`Person`)
- Then the 2nd subclass (`Student`)



Constructor Chaining: super()

When you construct an object of a type with parent classes, we call the constructors of all of the parents in sequence. This is done **implicitly** - Java compiler inserts call to `super()`

What if your classes have explicit constructors that take arguments? You need to explicitly chain and use `super( )` in constructor code to invoke superclass constructor.

Constructor Chaining

```
public class Person {
    protected String name;
    protected int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}


public class Student extends Person  {
    private long studentId;

    public Student(String name, int age, long studentId)
{
        super(name, age);    // means:  Person(name, age)
        this.studentId = studentId;
    }
}
```

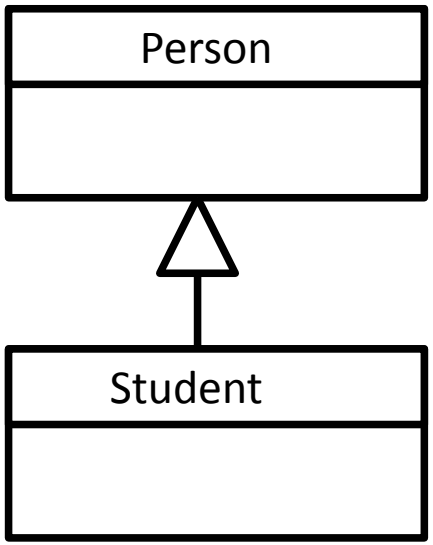
Casting

- We know we can *type cast* between numeric types

```
int i = 7;
float f = (float) i;   // f==7.0
double d = 3.2;
int i2 = (int) d;     // i2==3
```

- With inheritance it is reasonable to type cast an object within the inheritance tree...

Widening



- Student is-a Person
- Hence we can use a Student object anywhere we want a Person object
- Can perform *widening* conversions (up the tree)

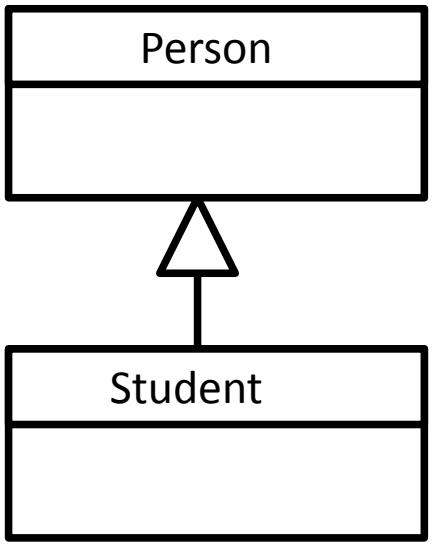
```
Student s = new Student();

Person p = (Person) s;
```

Explicit cast

```
public void print(Person p) {...}

Student s = new Student();
print(s);
```

Implicit cast

Narrowing



- Narrowing conversions move down the tree (more specific)
- Need to take care...

```
Person p = new Person();

Student s = (Student) p;
```

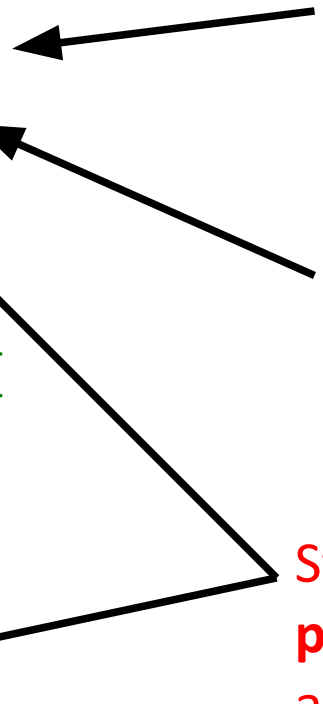
FAILS. Not enough info
In the real object to represent a Student

```
Student s = new Student();
Person p = (Person) s;
Students s2 = (Student) p;
```

OK because underlying object really is a Student

Fields and Inheritance

```
class Person {
   public String mName;
   protected int mAge;
   private double mHeight;
}

class Student extends Person {

  public void do_something() {
    mName=”Bob”;
    mAge=70;
    mHeight=1.70;
  }

}
```



Student inherits this as a public variable and so can access it

Student inherits this as a protected variable and so can access it directly

Student inherits this but as a **private** variable and so cannot access it directly (so this code won’t compile)

Fields and Inheritance: Shadowing

```
class A {   public int x; }

class B extends A {
   public int x;
}

class C extends B {
  public int x;

  public void action() {
     // Ways to set the x in C
     x = 10;
     this.x = 10;

     // Ways to set the x in B
     super.x = 10;
     ((B)this).x = 10;

     // Ways to set the x in A
     ((A)this.x = 10;
  }
}
```

Don’t write code like this.

Ever.

I mean it.

Methods and Inheritance: Overriding

- We might want to require that every Person can dance. But the way a Lecturer dances is not likely to be the same as the way a Student dances...

```
class Person {
  public void dance() {
    jiggle_a_bit();
  }
}
```

Person defines a 'default' implementation of dance()

```
class Student extends Person {
  public void dance() {
    twerk();
  }
}
```

Student overrides the default

```
class Lecturer extends Person {
}
```

Lecturer just inherits the default implementation and jiggles

Use `@Override` notation for code clarity

Abstract Classes

Abstract Classes and Methods

- Sometimes we want to force a subclass to implement a method but there isn't a convenient default behaviour to put in the parent
- An **abstract** method is used in a base class to do this
- It has no implementation whatsoever

```
abstract class Person {
  public abstract void dance();
}

class Student extends Person {
  public void dance() {
    twerk();
  }
}

class Lecturer extends Person {
  public void dance() {
    jiggle_a_bit();
  }
}
```

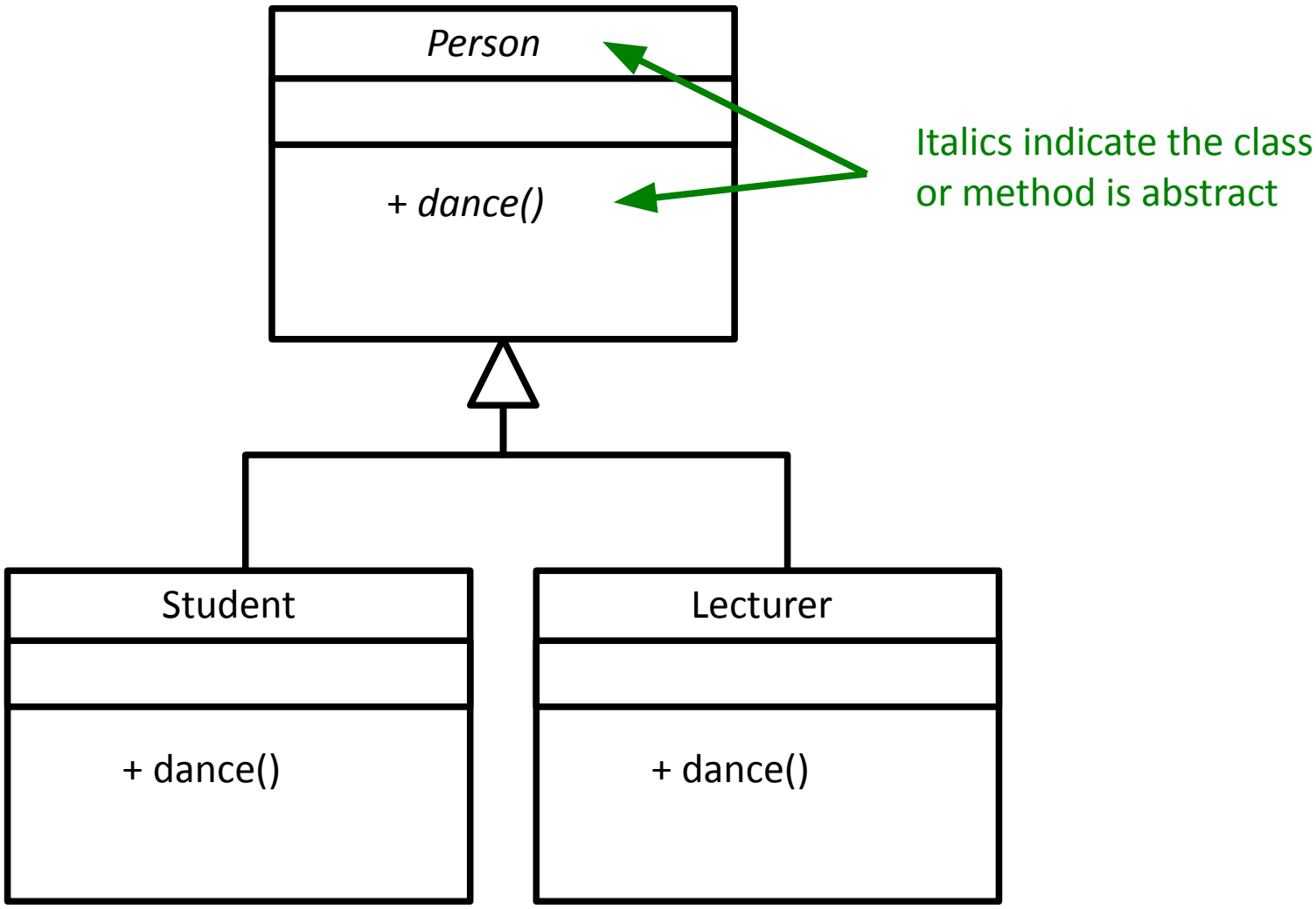
Abstract Classes

- Note that I had to declare the class abstract too. This is because it has a method without an implementation so we can't directly instantiate a Person.

```
public abstract class Person {
   public abstract void dance();
}
```

- All state and non-abstract methods are inherited as normal by children of our abstract class
- Interestingly, Java allows a class to be declared abstract even if it contains no abstract methods! (Why?)

Representing Abstract Classes



Interfaces

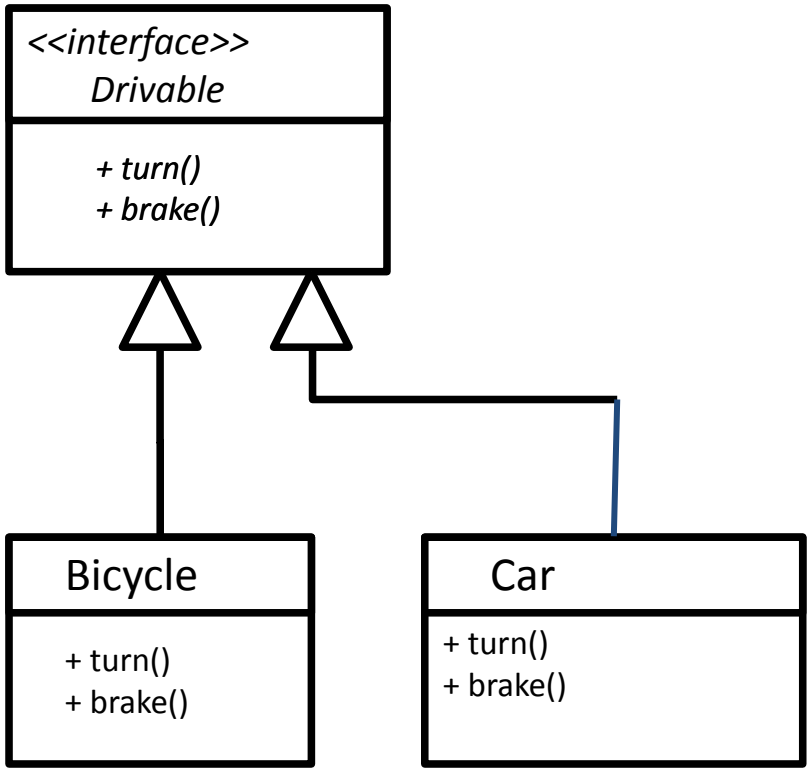
Abstracting Further: Interfaces

- An interface is a contract that groups together required methods

- [For now] It's a collection of exclusively abstract functions that force classes that inherit from them to provide concrete implementations

- Interfaces are declared using **`interface`** rather than `class`
- Interfaces are used via **`implements`** rather than `extends`

Abstracting Further: Interfaces




```
interface Drivable {
   public void turn();
   public void brake();
}

class Bicycle implements Drivable {
   public void turn() {...}
   public void brake() {… }
}

class Car implements Drivable {
   public void turn() {...}
   public void brake() {… }

}
```

Type is inherited via extends **and** implements

```
interface Drivable {
   public void turn();
   public void brake();
}

abstract class Vehicle implements Drivable {
   public abstract void turn();
   public void brake() {… }
}

class Car extends Vehicle {
   public void turn() {...}
   public void brake() {… }

}
```

- Vehicle inherits the Drivable type. i.e. Vehicle is-a Drivable
- Car inherits Vehicle's type**s** i.e. Car is-a Vehicle and is-a Drivable
- You can use `instanceof` to check this if you want to investigate

Abstract class or interface?

Feature	Abstract Class	Interface
(Abstract) Methods	✓	✓
Behaviour	✓	✓
Class can Implement Multiple	×	✓
Instance Variables	✓	×
Protected/Package Scoped Methods	✓	×
(**Java 9**) Private methods	✓	✓
Static methods	✓	✓

Inheritance gone wrong

When Inheritance goes wrong

When inheritance goes wrong, it's usually because it's being used when there isn't a good is-a relationship

A common error is to relate classes using inheritance ("is-a") when you should use composition ("has-a") and vice versa

Surprisingly this mistake is easier to make than you might think. Even the JDK has issues…

JDK Stack

```
class Stack extends Vector {
    public Object pop() { … }
    public void push(Object o) { }
}
```

Inheritance

This is bad because a Stack is not a Vector (which is the original ArrayList in Java). The result is Stack suddenly has methods that don't make sense - e.g. add(int index, E element)

JDK Stack as it should have been

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

Composition

Here, we decide Stack has-a Vector. In this way we can use the implementation in Vector without exposing methods that aren't relevant

Polymorphism, Multiple Inheritance, Coupling

(Subtype) Polymorphism

Polymorphism

Poly - morph = many forms

Polymorphism in OOP means that many kinds of objects can provide the same method, and we can invoke that method without knowing which kind of object will perform it. We typically refer to this characteristic as **subtyping** polymorphism.

Other forms of polymorphism previously too:

- Parametric polymorphism (i.e. generics)
- Ad-hoc polymorphism (i.e. overloading)

Polymorphic Methods

```
Student s = new Student();
Person p = (Person)s;
p.dance();
```

- Assuming Person has a default dance() method, what should happen here??

- General problem: when we refer to an object via a parent type and both types implement a particular method: which method should it run?

Static Polymorphism

- **Static** polymorphism
  - Decide at compile-time
  - Since we don't know what the true type of the object will be, we just run the parent method
  - Type errors give compile errors

```
Student s = new Student();
Person p = (Person)s;
p.dance();
```

- Compiler says “p is of type Person”
- So p.dance() should do the default dance() action in Person

Static Polymorphism

```
Person p = null;

if (inputParam == 1) p = (Person) new Student();
else p = (Person) new Lecturer();

p.dance(); // the implementation from Person class runs
```

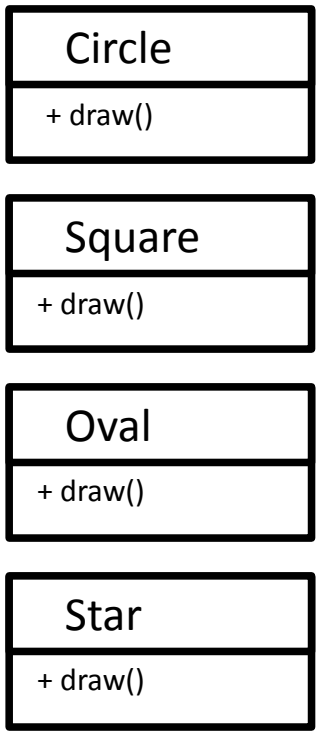
Dynamic Polymorphism

- **Dynamic** polymorphism
  - Run the method in the child
  - Must be done at run-time since that's when we know the child's type
  - Type errors cause run-time faults (crashes!)

```
Student s = new Student();
Person p = (Person)s;
p.dance();
```

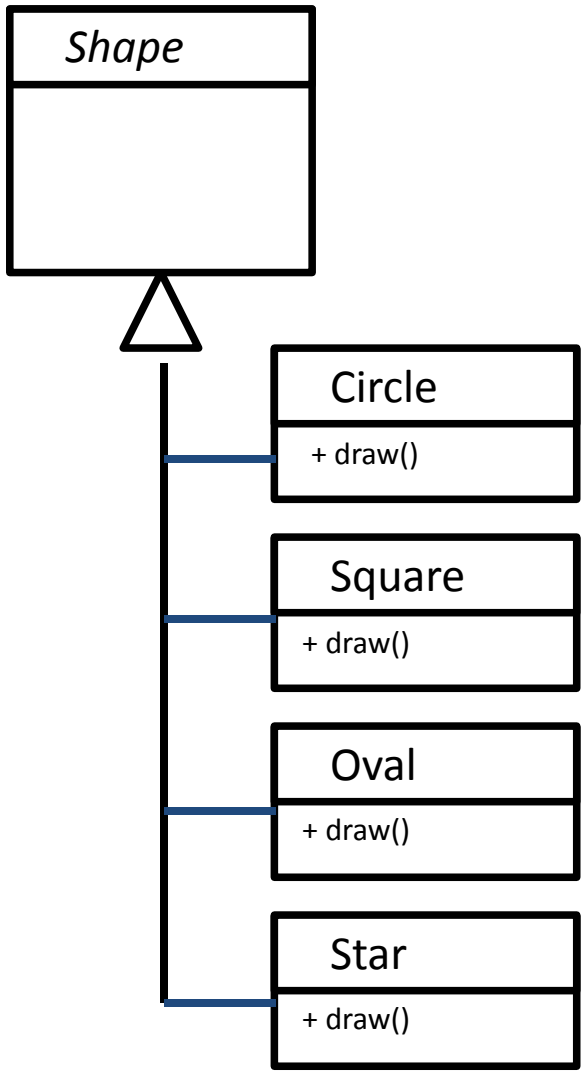
- Compiler looks in memory and finds that the object is really a Student
- So p.dance() runs the dance() action in Student

The Canonical Example I



- A drawing program that can draw circles, squares, ovals and stars
- It would presumably keep a list of all the drawing objects
- **Option 1**
  - Keep a list of Circle objects, a list of Square objects,...
  - Iterate over each list drawing each object in turn
  - What has to change if we want to add a new shape?

The Canonical Example II

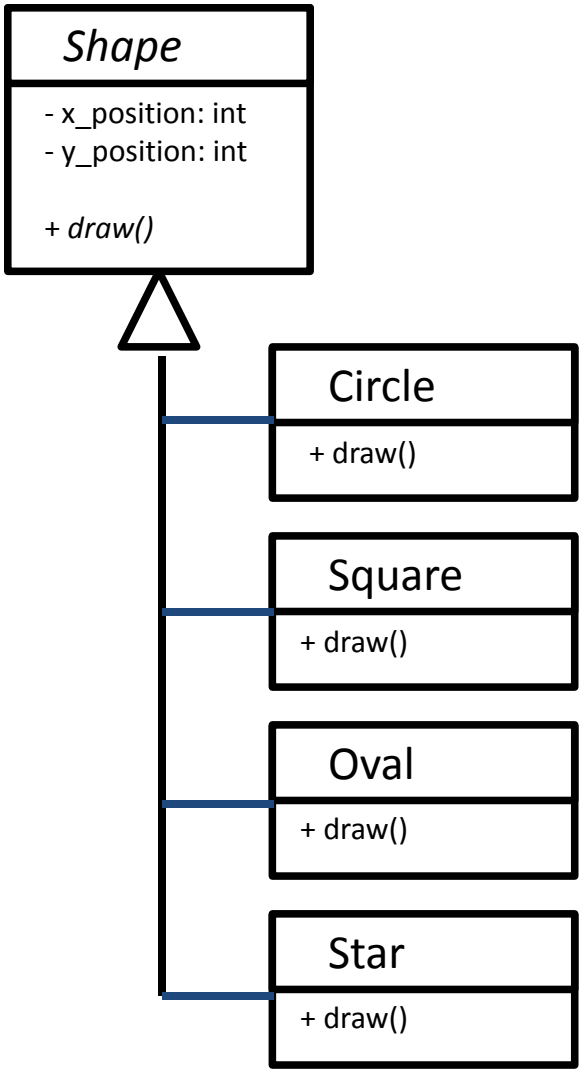


- **Option 2**
  - Keep a single list of Shape references
  - Figure out what each object really is, narrow the reference and then draw()

```
for every Shape s in myShapeList
  if (s is really a Circle)
    Circle c = (Circle)s;
    c.draw();
  else if (s is really a Square)
    Square sq = (Square)s;
    sq.draw();
  else if…
```

  - What if we want to add a new shape?

The Canonical Example III



- **Option 3 (Polymorphic)**
  - Keep a single list of Shape references
  - Let the compiler figure out what to do with each Shape reference

```
For every Shape s in myShapeList
  s.draw();
```

  - What if we want to add a new shape?

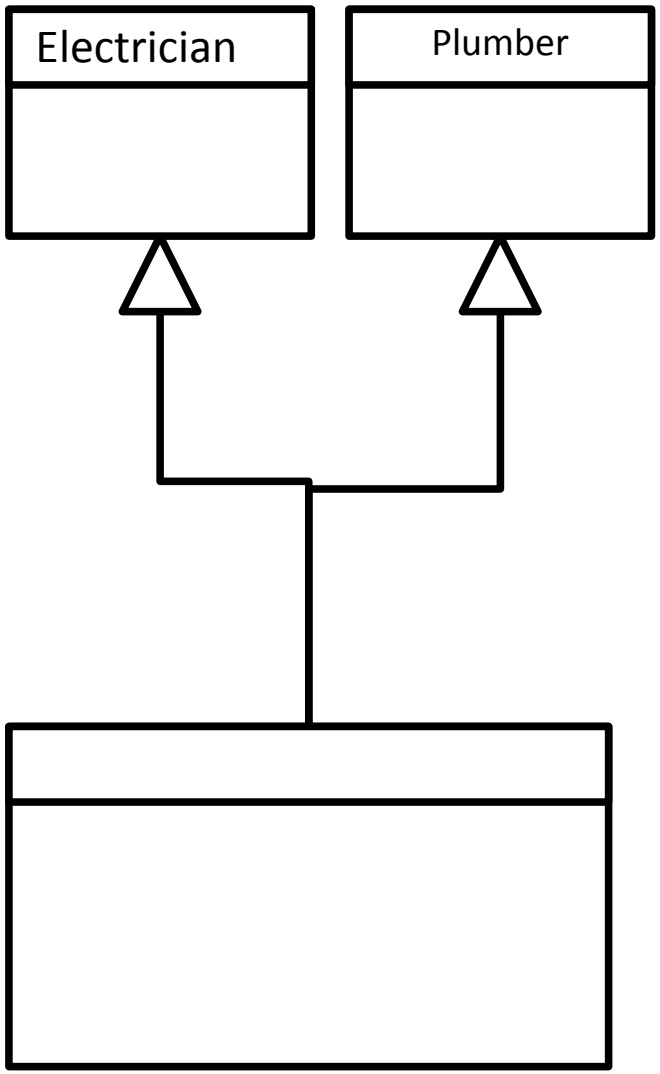
Implementations

- Java
  - All methods are dynamic polymorphic.
- Python
  - All methods are dynamic polymorphic.
- C++
  - Only functions marked *virtual* are dynamic polymorphic

- Polymorphism in OOP is an extremely important concept that you need to make sure you understand...

Multiple Inheritance

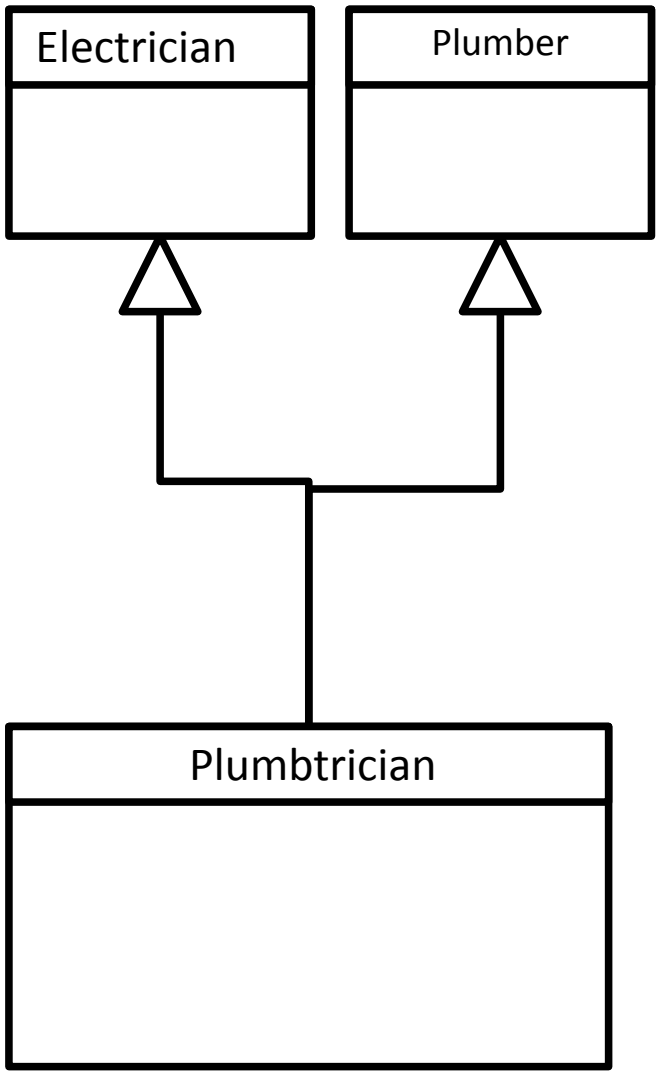
Multiple Inheritance



- Imagine you work for a construction firm and you have software to track your tradespeople.
- You already have an Electrician class and a Plumber class
- You've been asked to deal with a new employee who is qualified to do both!
- Your solution: multiple inheritance!

(Health warning: Java doesn't support this)

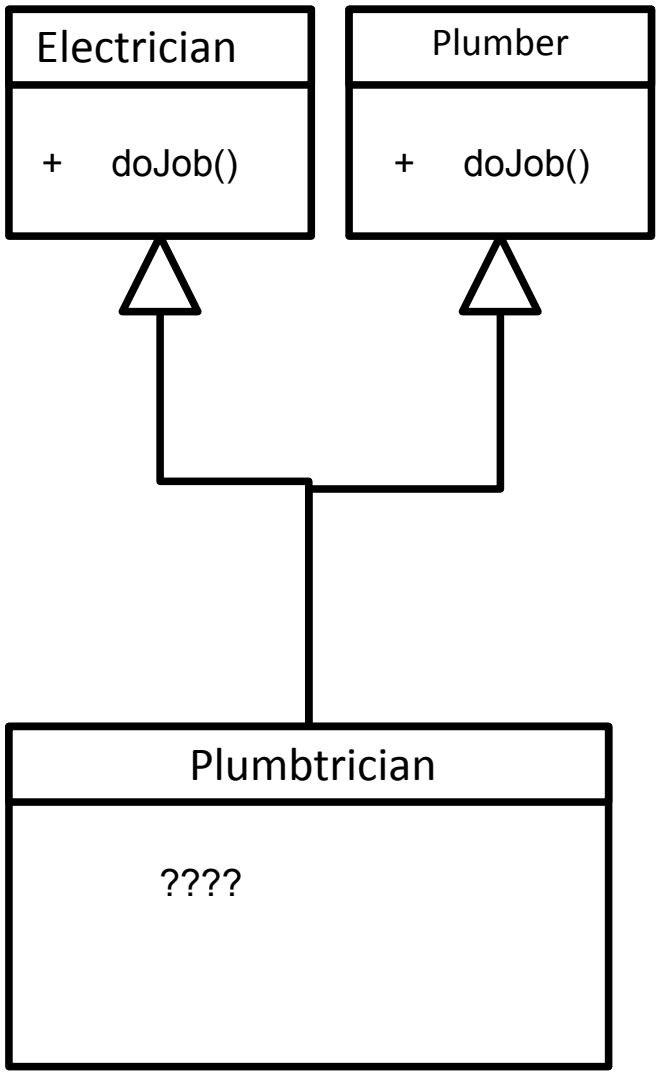
Multiple Inheritance



- Imagine you work for a construction firm and you have software to track your tradespeople.
- You already have an Electrician class and a Plumber class
- You've been asked to deal with a new employee who is qualified to do both!
- Your solution: multiple inheritance!

(Health warning: Java doesn't support this)

Multiple Inheritance

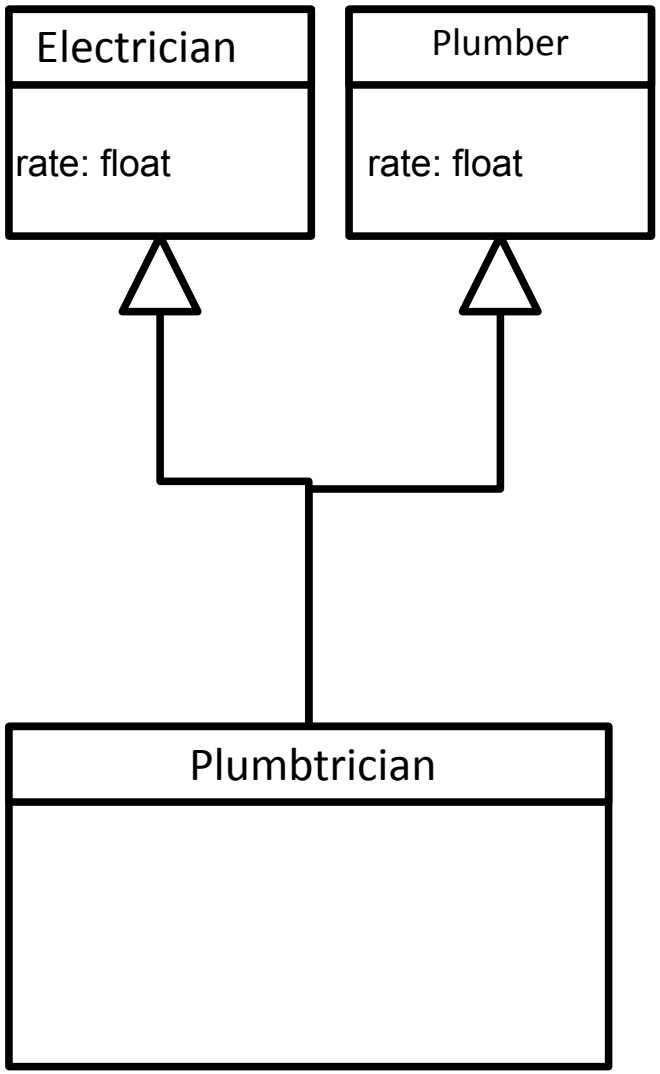


Multiple inheritance of behaviour

- Introduces a new problem: name clashes
- Which doJob() should Plumbtrician inherit?
- There are various ways to handle this. Generally it's fixable, but can add complexity to your code

(Health warning: Java doesn't support this)

Multiple Inheritance

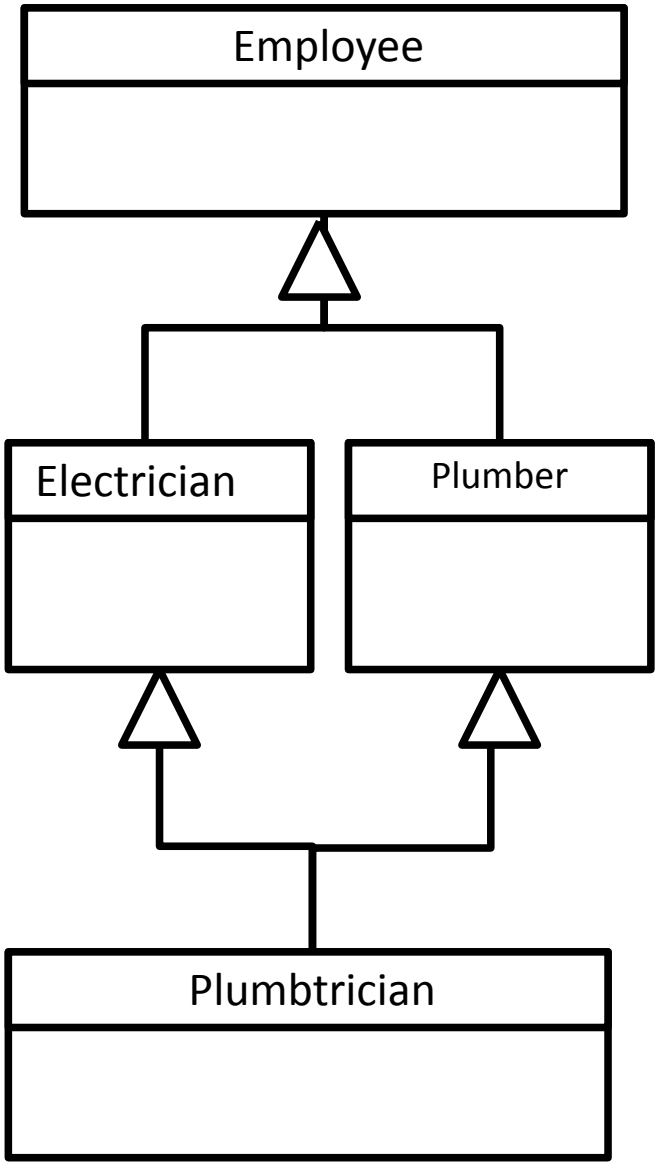


Multiple inheritance of state

- State is particularly problematic, because you can end up inheriting multiple states with the same name
- Need nasty syntax like Electrician::rate and Plumber::rate → have to change it everywhere in your code!
- Whatever you do your code ends up being less readable and might confuse and lead to bugs

(Health warning: Java doesn't support this)

Diamond problem



- The ‘dreaded diamond’ is particularly annoying, since it guarantees loads of state and behaviour name clashes

(Health warning: Java doesn’t support this)

So is multiple inheritance just evil?

No (despite what the internet may tell you). It's a **tool** that can be useful.

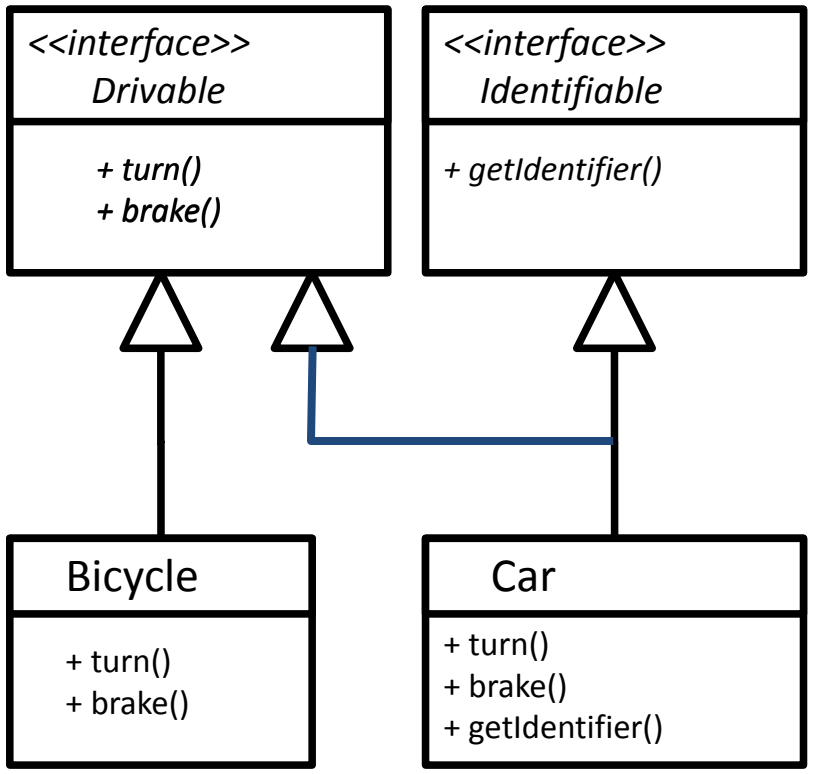
But, like all tools, you need to know when to use it and what the consequences might be

For the most part, multiple inheritance adds complexity and sometimes ambiguity to your code. Sometimes it's the result of not creating a correct class hierarchy, sometimes (often) the same effect can be achieved more neatly using other methods, and sometimes it's the neatest way to do it (in truth, rarely)

Java limits how far you can go with it…

Java's Take on it

- Classes can have at most **one** **direct** parent. Period.
- But we can allow multiple **interfaces** to be **directly** inherited




```
interface Drivable {
   public void turn();
   public void brake();
}

interface Identifiable {
   public void getIdentifier();
}

class Bicycle implements Drivable {
   public void turn() {...}
   public void brake() {… }
}

class Car implements Drivable, Identifiable {
   public void turn() {...}
   public void brake() {… }
   public void getIdentifier() {...}
}
```

i.e.

- A given class can extend up to one direct parent but multiple direct interfaces
- Name clashes in abstract functions is OK since there was no implementation in the first place!
- There's no state to have name clashes on! (caveat: see next few slides)

```
public class Lecturer extends Person implements
  GeniusAbility, NinjaAbility, … { }
```

Just for fun…

```
interface Collection<E> extends Iterable<E>

abstract class AbstractCollection<E> implements Collection<E>

abstract class AbstractList<E> extends AbstractCollection<E>
         implements List<E>

public class ArrayList<E> extends AbstractList<E>
        implements List<E>,
                   RandomAccess,
                   Cloneable,
                   Serializable
```

Except....

- Early Java had interfaces that were completely abstract, so this multiple inheritance was solved by the previous rules

Except....

- Early Java had interfaces that were completely abstract, so this multiple inheritance was solved by the previous rules
- But Java 8 added **default methods** to interfaces (Java 8)

```
public interface SomeInterface {
  default void some method() {
    System.out.println("Oh no…")
  }
}
```

- ***Why??!!***

API evolution

Imagine having thousands of classes implementing your interface (definitely true for the JDK interfaces).. What happens if you decide that you want to add a new feature to the API?

If you just add it to the interface, you will break every class that implements it

You could inherit from it and add a subinterface. But APIs do evolve - having a huge inheritance hierarchy would really suck

Default methods solve this: you can add an implementation that the The other classes will just inherit, and put the functionality you need in your new class

Cool! But, it does give us multiple inheritance of behaviour headaches (not state)…

Resolution Rules for Method Clashes

1. Classes always win

2. Otherwise, subinterfaces win. The method with the same signature in most specific interface is selected

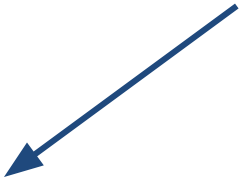
3. If the choice is still ambiguous, the class inheriting must override the method and be explicit

(Note you have to know these resolution rules to know what Java will actually do. That does not result in readable, easily-maintained code :-/)

Principles for good OOP

Open-Closed Principle (OCP)

Easy to **add** new behaviour



Make your classes **open to extension** but **closed to modification**

Hard to **change** existing behaviour

OCP example

```
// Original

public class Order {
    private List<Item> items;

    public Order(List<Item> items) {
        this.items = items;
    }

    public double calculateTotal() {
       // Compute
    }
}
```

Goal: add discount ability

OCP example

```
// OCP Violation

public class Order {
    private List<Item> items;
    private double discount;

    public Order(List<Item> items, double discount) {
        this.items = items;
        this.discount = discount;
    }

    public double calculateTotal() {
        // Compute with discount
    }
}
```

We modified the original and this carries a high risk of breaking things

OCP example

```
// OCP Fixed

public interface Discount {
    double applyDiscount(double total);
}

public class DiscountedOrder extends Order {
    private Discount discount;

    public DiscountedOrder(List<Item> items, Discount discount) {
        super(items);
        this.discount = discount;
    }

    @Override
    public double calculateTotal() {
        // Compute wih discount
    }
}
```

Inheritance used to extend while original is untouched

Liskov Substitution Principle

- Concerned with subtyping and inheritance
- **Subtypes must be behaviourally substitutable for their base types without negative side effects**
- If don't adhere to it, leads to clunky code and corner cases

```
public interface Persistable {
  void load();
  void save();
}
```

Example Violation

```
class ApplicationSettings implements Persistable { … }

class UserSettings implements Persistable { … }


class AdminSettings implements Persistable {

    void load() { … }

    void save() { throw new NotImplementedException(); }

}
```

You can't use substitute a Persistable for an AdminSettings without getting a negative effect (an exception thrown)

Example Violation

```
static void saveAll(List<Persistable> resources) {
for(Persistable r: resources) {
   if (r instanceof AdminSettings) { continue; }
   r.persist();
}
```

This is nasty code that suggests we have the wrong abstraction

Example Violation

```
public interface Loadable {
    void load();
}

public interface Persistable {
    void save();
}
```

This solves it

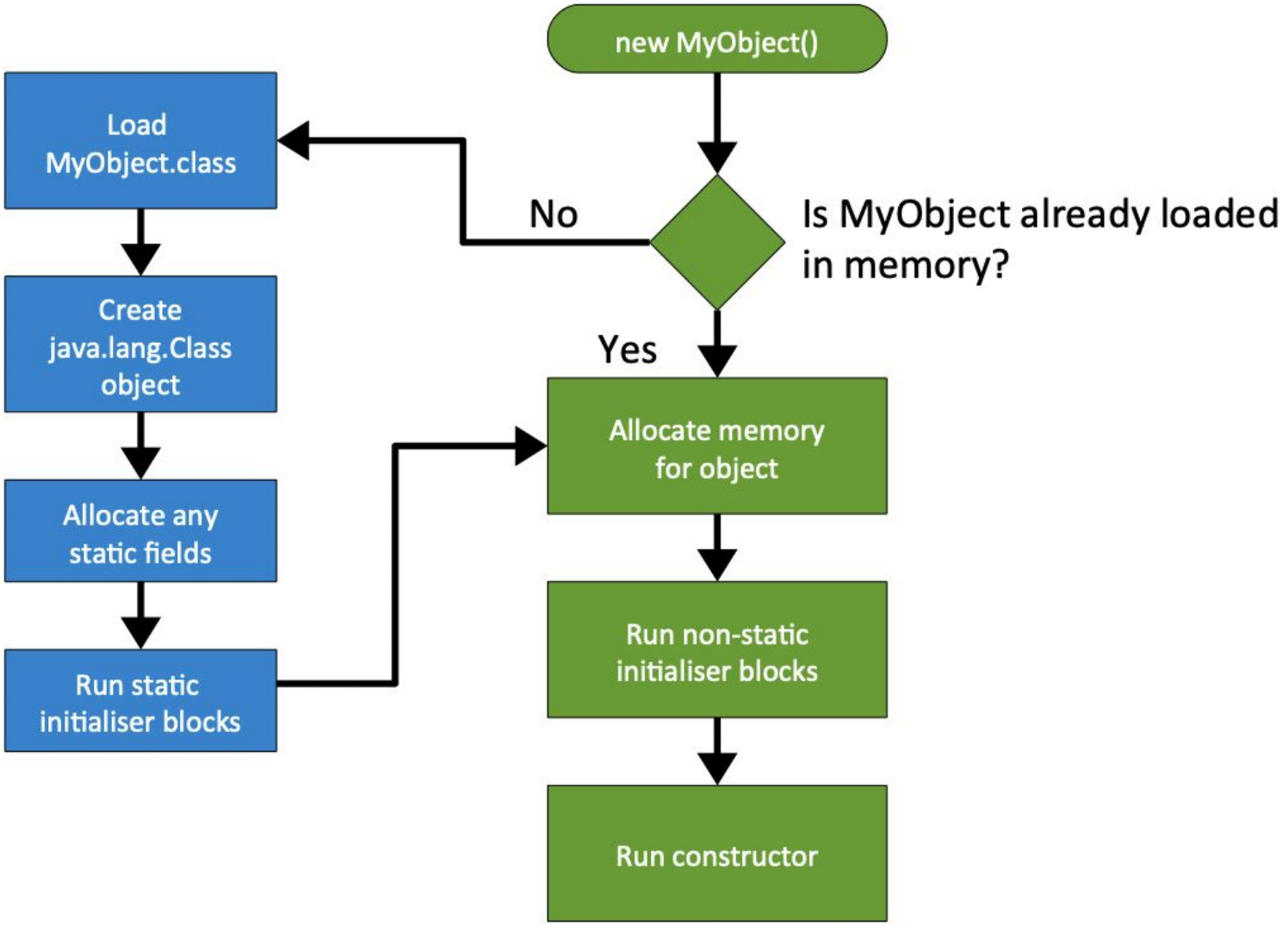
JDK LSP Violation!

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    default void remove() {
      throw new UnsupportedOperationException("remove");
    }
}
```

Object Life Cycle, Garbage Collection, Copying

Creating objects

Object Creation



Initialisation

```
public class Blah {
  private int mX = 7;
  public static int sX = 9;

  {
     mX=5;
   }

   static {
     sX=3;
   }

  public Blah() {
    mX=1;
    sX=9;
  }
}
```

```
Blah b = new Blah();
Blah b2 = new Blah();
```

1. Blah loaded
2. sX created
3. sX set to 9
4. sX set to 3
5. Blah object allocated
6. mX set to 7
7. mX set to 5
8. Constructor runs (mX=1, sX=9)
9. b set to point to object

10. Blah object allocated
11. mX set to 7
12. mX set to 5
13. Constructor runs (mX=1, sX=9)
14. b2 set to point to object

Deleting objects

Cleaning Up

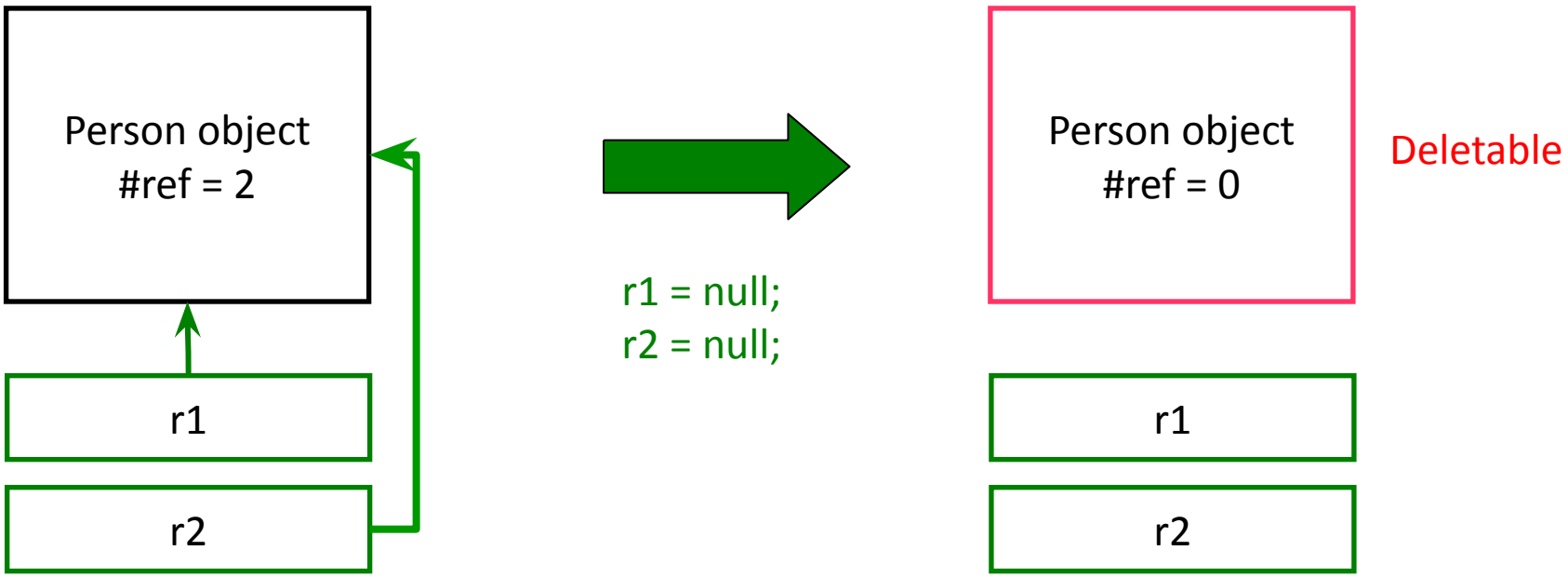
- A typical program creates lots of objects, not all of which need to stick around all the time

- **Approach 1:**
  - Allow the programmer to specify when objects should be deleted from memory
  - Lots of control, but what if they forget to delete an object?
    - A “memory leak”

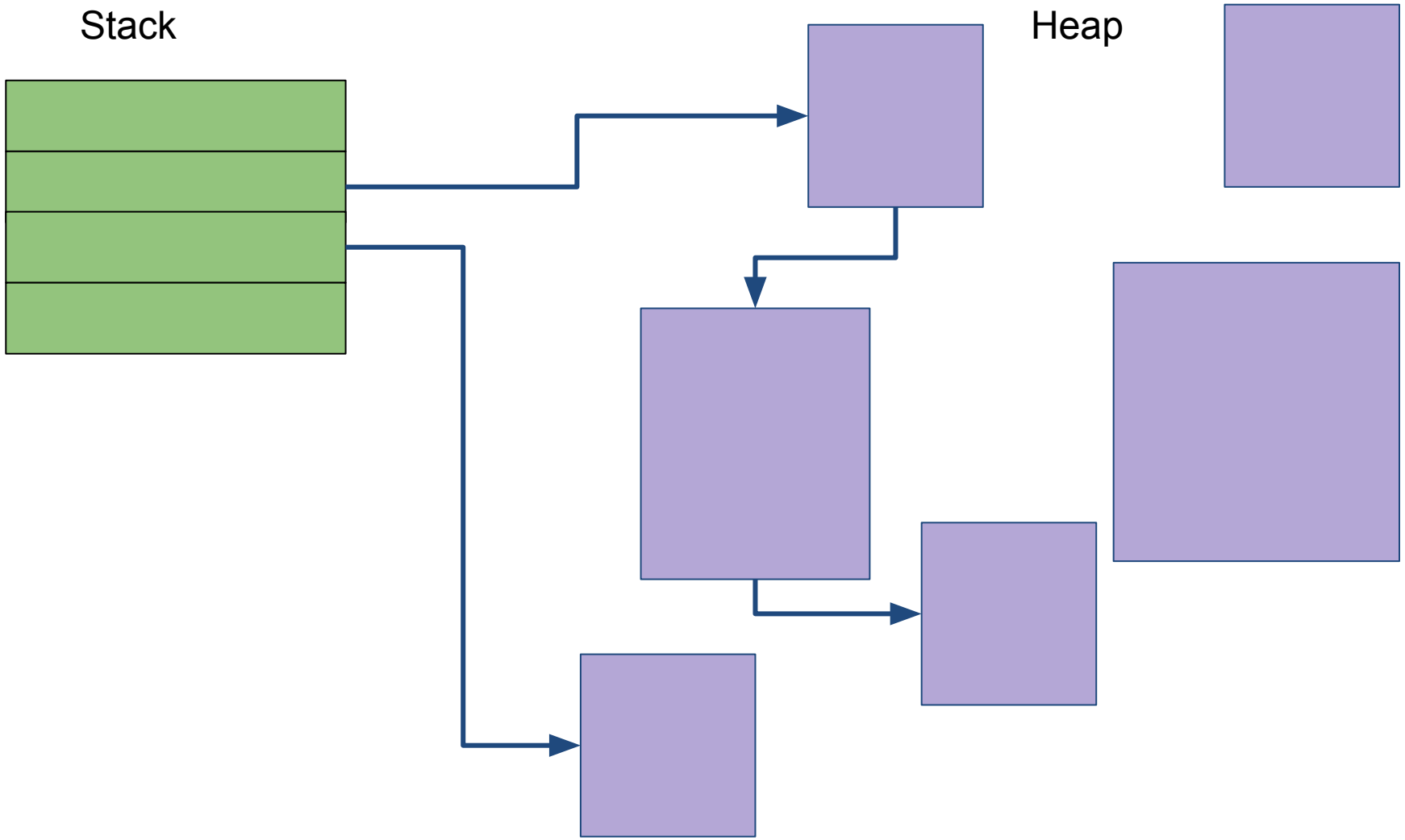
- **Approach 2:**
  - Delete the objects automatically (**Garbage collection**)
  - But how do you know when an object will never be used again and can be deleted??

Cleaning Up (Java) I

- Java *reference counts*. i.e. it keeps track of how many references point to a given object. If there are none, the programmer can't access that object ever again so it can be deleted

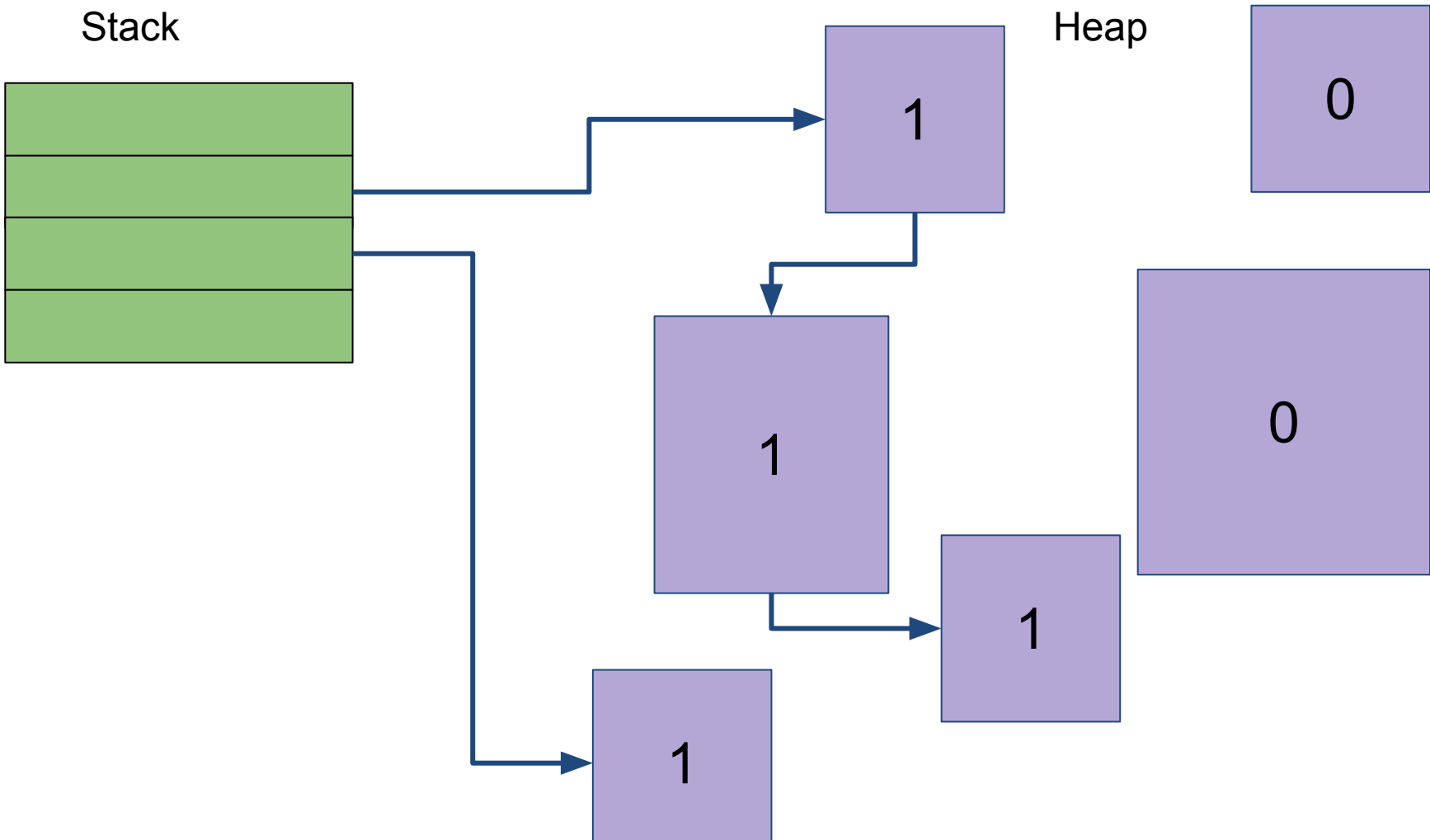


Mark and Sweep



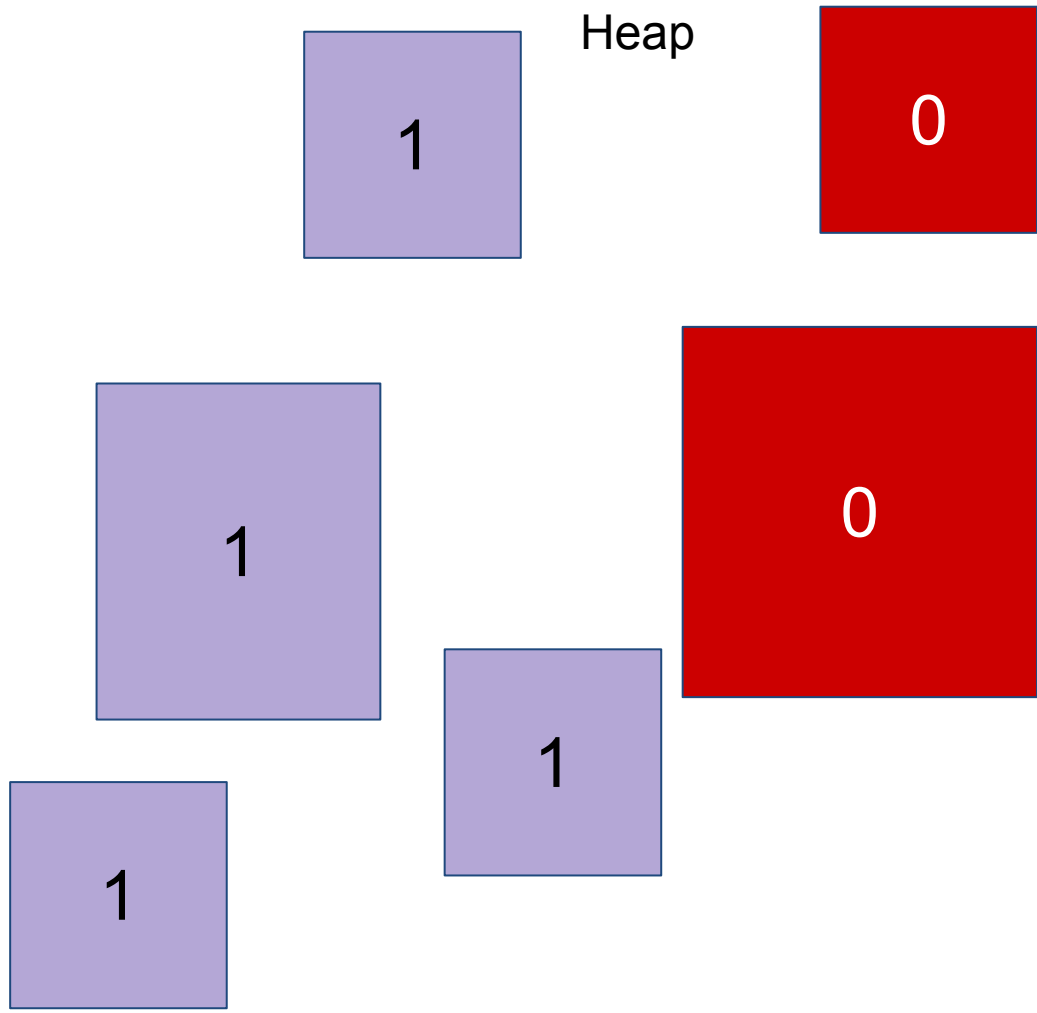
Mark and Sweep

Step 1: (Mark) Starting at each stack reference, follow references of everything reachable. Mark each object you find



Mark and Sweep

Step 2: (Sweep) Traverse all objects on the heap. If they are marked, unmark them. If they are not marked, queue them for deletion



Deletion and Compaction

Java supports multiple different GCs, which take different approaches to the next phase.

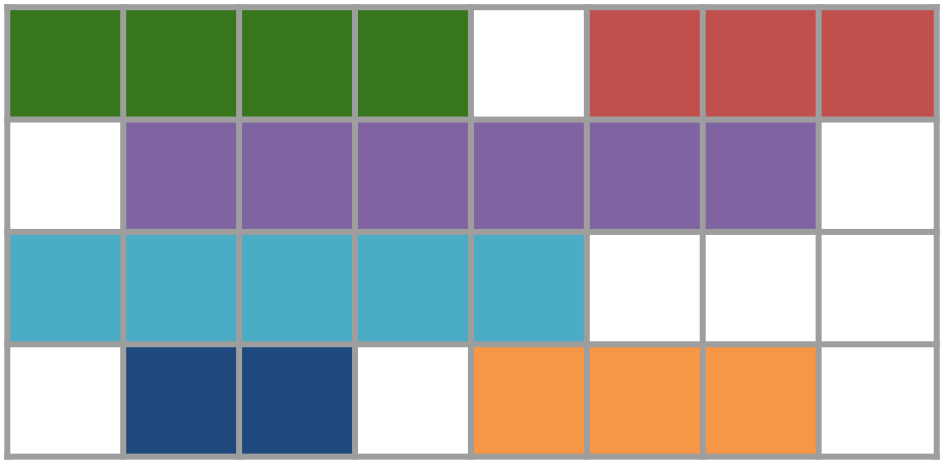
Delete immediately. Simplest approach. But if there's lots to delete, can take a while. Unpredictable pauses.

Delete next time. Queue for deletion if we've already spent too long deleting in this round.

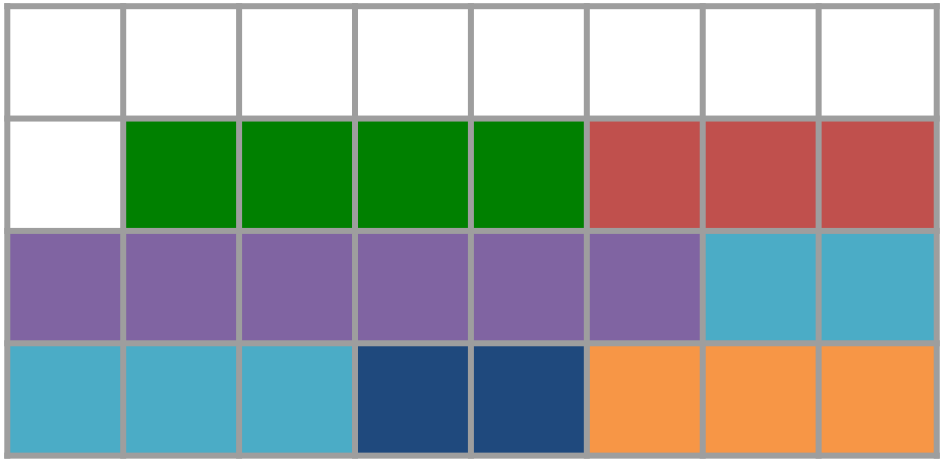
Don't delete. In some cases, we could decide not to delete (either ever, or until we hit a critical point such as running out of memory).

After any deletion, the GC can also decide to **compact**: rearrange the surviving objects in memory to reduce gaps. Means updating the references too of course.

Compacting



Over time, heap deletions and creations leaves tiny chunks of available memory → inefficient use



Moving objects around allows us to pool the available memory. More efficient but all the refs need to be updated!

The Collector isn’t Free

The work the GC does is clearly work that takes away from the main program you are running.

We have different strategies and GC algorithms for different scenarios

Heap Division

Core observation: the majority of objects actually don't last long

All objects are created in **Eden**
If they survive a few GCs, they are promoted to **Survivors**
If they survive a few more GCs, they are promoted to **Tenured**

The GC runs frequently on Eden, less so on Survivors and much less so on Tenured

Different GCs

You can actually select the GC you want. These are common:

Serial GC. A 'stop-the-world' GC where the program stops executing entirely while the GC runs. Simple, but not great for responsive programs (must tolerate short pauses). Tiny implementation though, so gets used for embedded applications.

Parallel GC. Another 'stop-the-world' GC. But runs the collection from multiple concurrent threads to be faster

Garbage first (G1). The modern default. The GC monitors memory *concurrently* (while the app still runs), doing as much as it can. Uses short stop-the-world events to do the deletions, creating regions in memory and prioritising based on how much needs to be done.

Epsilon GC. Don't do anything at all (a no-op GC). Useful if you *know* your program will use constant memory

Destructors

- Most OO languages have a notion of a destructor too
  - Gets run when the object is destroyed
  - Allows us to release any resources (open files, etc) or memory that we might have created especially for the object

C++

```
class FileReader {
  public:

    // Constructor
    FileReader() {
      f = fopen("myfile","r");
    }

    // Destructor
    ~FileReader() {
      fclose(f);
    }

  private :
    FILE *file;
}
```

```
int main(int argc, char ** argv) {

  // Construct a FileReader Object
  FileReader *f = new FileReader();

  // Use object here
  ...

  // Destruct the object
  delete f;

}
```

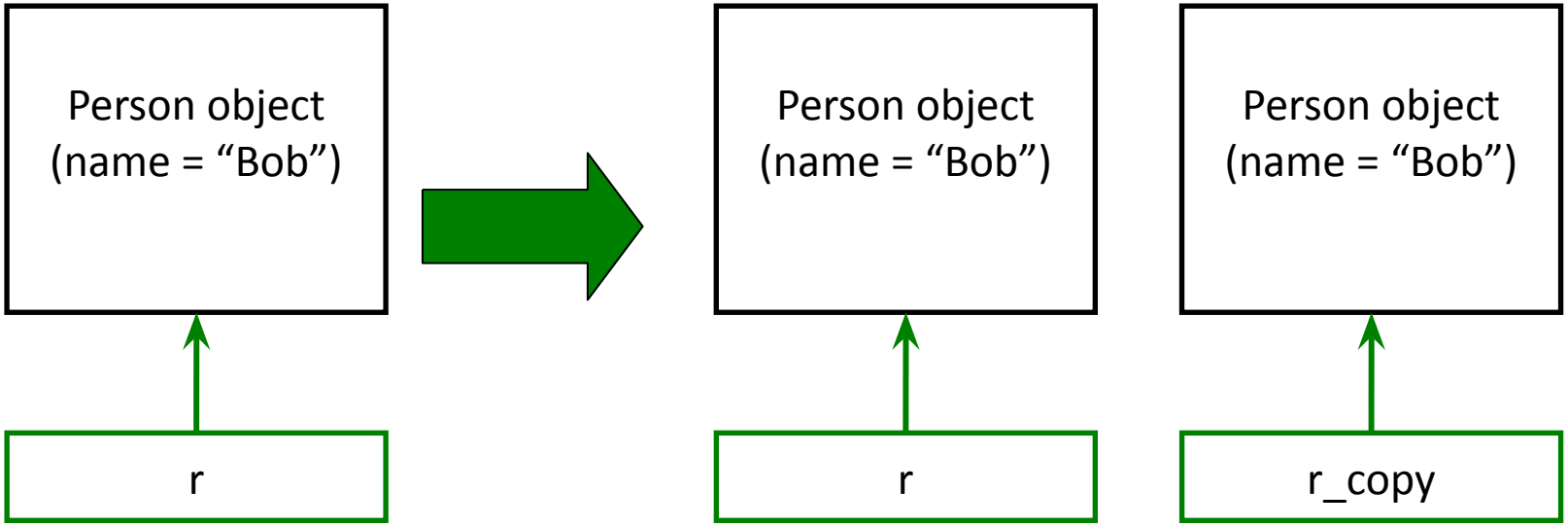
Java's finalise()

- Java has a method finalize in Object that was meant to be a destructor
- But it can only run when the GC actually deletes the object, which may be never, or certainly isn't easy to predict!
- Mostly became too problematic and it is now deprecated (i.e. don't use it).
- May be able to use `try-with-resources` (Lecture 10)

Copying objects

Object Copying

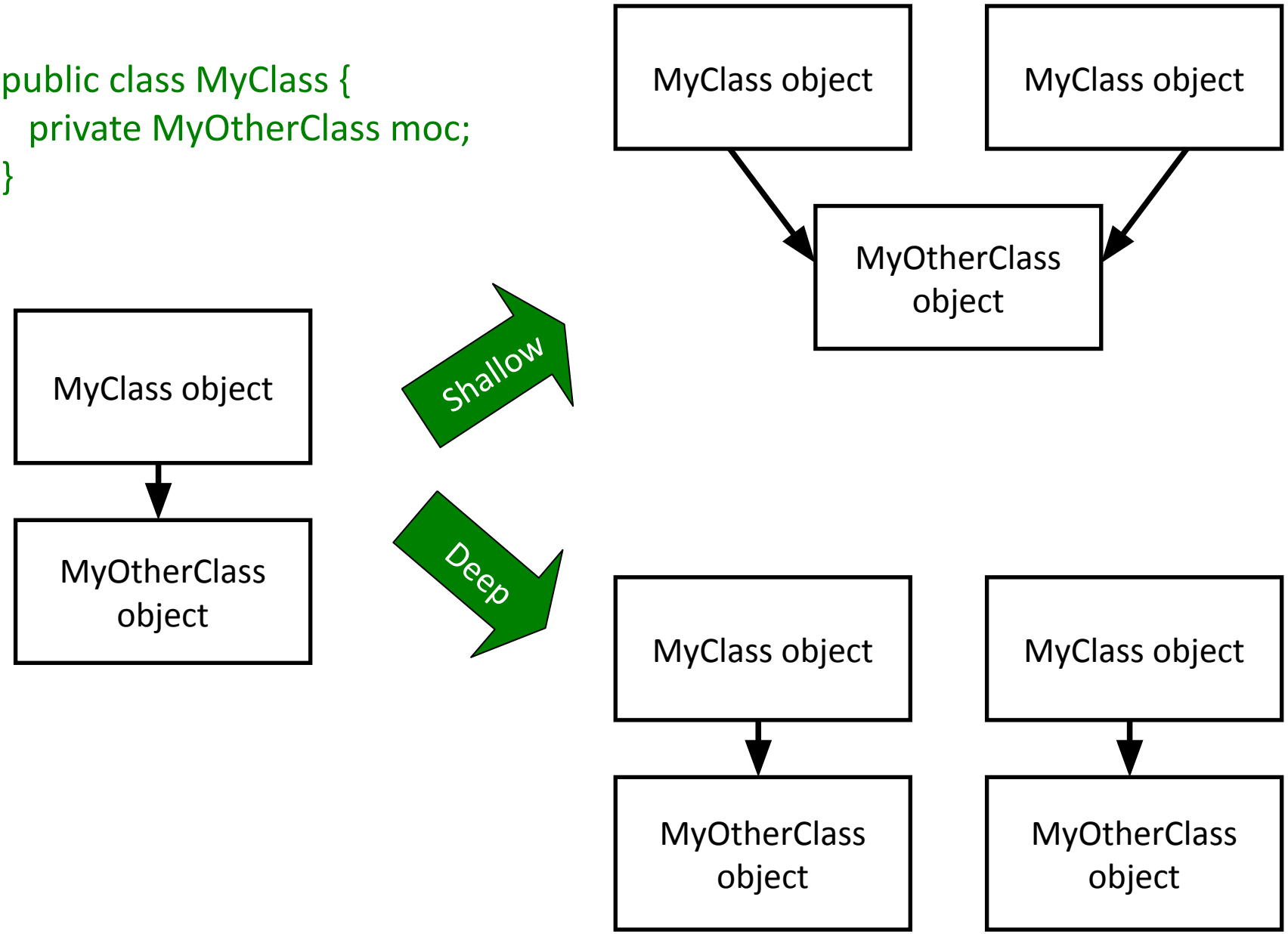
- Sometimes we really do want to copy an object



- Aka ‘cloning’

Shallow and Deep Copies

```
public class MyClass {
   private MyOtherClass moc;
}
```



Copy Constructors

Most programmers prefer to define a copy constructor that takes in an object of the same type and manually copies the data

```
public class Vehicle {
    private int age;
    private double vx;
    private double vy;

    …

    public Vehicle(Vehicle v) {
        this.age=age;
        this.vx = vx;
        this.vy = vy;
    }
}
```

Copy Constructors

- Now we can create copies by:

```
Vehicle v = new Vehicle(5, 0.f, 5.f);

Vehicle vcopy = new Vehicle(v);
```

- This is a neat approach, but:
  - deep copying can be hard
  - inheritance makes it hard:

```
Car c = new Car(5, 0.f, 5.f);  // Copy constructor is not inherited

Vehicle v = (Vehicle)c;  // If we need to copy this later, how do we
                         //  know to call new Car and not new Vehicle?
```

Java *tried* to solve this with clone()

- Every class in Java ultimately inherits from the **Object** class
  - This class contains a clone() method so we just call this to clone an object, right?
  - This can go horribly wrong if our object contains reference types (objects, arrays, etc)

Health warning: most java programmers recommend against using clone(), but it's instructive to at least discuss it here

Java Cloning

- So do you want shallow or deep?
  - The default implementation of clone() performs a **shallow** copy
  - But Java developers were worried that this might not be appropriate: they decided they wanted to know for sure that we'd thought about whether this was appropriate

- Java has a **Cloneable** interface
  - If you call clone on anything that doesn't extend this interface, it fails

Clone Example I

```
public class Velocity {
  public float vx;
  public float vy;
  public Velocity(float x, float y) {
     vx=x;
     vy=y;
  }
};

public class Vehicle {
  private int age;
  private Velocity vel;
  public Vehicle(int a, float vx, float vy) {
     age=a;
     vel = new Velocity(vx,vy);
  }
};
```

Clone Example II

```
public class Vehicle implements Cloneable {
  private int age;
  private Velocity vel;
  public Vehicle(int a, float vx, float vy) {
     age=a;
     vel = new Velocity(vx,vy);
  }

  @Override
  public Object clone() {
     return super.clone();   // shallow: won't clone the vel object
  }

};
```

Clone Example III

```
public class Velocity implement Cloneable {
      ....
     public Object clone() {
        return super.clone();
     }
};

public class Vehicle implements Cloneable {
  private int age;
  private Velocity v;
  public Student(int a, float vx, float vy) {
      age=a;
      vel = new Velocity(vx,vy);
  }

  public Object clone() {
      Vehicle cloned = (Vehicle) super.clone();  // start with a shallow copy
      cloned.vel = (Velocity)vel.clone();  // add any deep copies you need
      return cloned;
  }
};
```

Cloning Arrays

- Arrays have build in cloning but the contents are only cloned *shallowly*

```
int intarray[] = new int[100];
Vector3D vecarray = new Vector3D[10];

...

int intarray2[] = intarray.clone();
Vector3D vecarray2 = vecarray.clone();
```

Overall

- Cloning is messy in Java
- When done right, it solves a lot of issues with copy constructors
- But it often isn’t done right, and causes confusion

Covariant Return Types

- The need to cast the clone return is annoying

```
public Object clone() {
    Vehicle cloned = (Vehicle) super.clone();
    cloned.vel = (Velocity)vel.clone();
    return cloned;
}
```

- Recent versions of Java allow you to override a method in a subclass and change its return type to a subclass of the original's class

```
class A {}

class B extends A {}
```

```
class C {
    A mymethod() {}
}

class D extends C {
    B mymethod() {}
}
```

Marker Interfaces

- If you look at what's in the Cloneable interface, you'll find it's empty!! What's going on?
- Well, the clone() method is already inherited from Object so it doesn't need to specify it
- This is an example of a **Marker Interface**
  - A marker interface is an empty interface that is used to label classes
  - This approach is found occasionally in the Java libraries

Collections, Comparisons

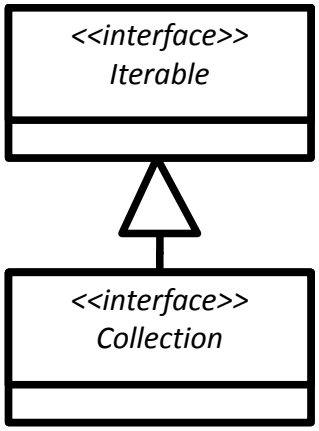
Collections

Java Class Library

- Java the platform contains around 4,000 classes/interfaces
  - Data Structures
  - Networking, Files
  - Graphical User Interfaces
  - Security and Encryption
  - Image Processing
  - Multimedia authoring/playback
  - And more...

- All neatly(ish) arranged into packages (see API docs)

Java's Collections Framework

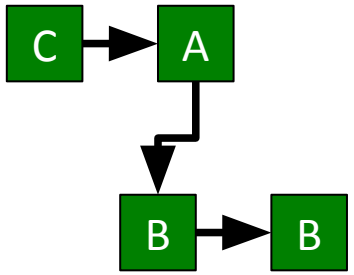


- Important chunk of the class library
- A Collection is some sort of grouping of things (objects)
- Usually when we have some grouping we want to go through it ("***iterate*** over it")

- The Collections framework has two main interfaces: Iterable and Collection. They define a set of operations that all classes in the Collections framework support
- add(Object o), clear(), isEmpty(), etc.

Lists

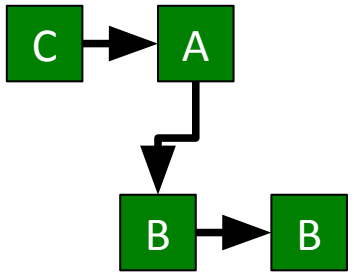
- <<interface>> List
  - An ordered collection of elements that may contain duplicates
  - LinkedLIst: linked list of elements
  - ArrayList: array of elements (efficient access)
  - Vector: legacy class, as ArrayList but threadsafe




```
LinkedList<Double> ll = new LinkedList<Double>();
ll.add(1.0);
ll.add(0.5);
ll.add(3.7);
ll.add(0.5);
ll.get(1);  // get element 2 (==3.7)
```

ArrayList vs LinkedList

- **ArrayList**
  - Good general purpose implementation
  - Use as default
  - More CPU cache sympathetic



- **LinkedList**
  - Worse performance for many read operations
  - Use when adding elements at start
  - Or when adding/remove a lot

	get	add	contains	remove
`ArrayList`	O(1)	O(1) amortised	O(N)	O(N)
`LinkedList`	O(N)	O(1)	O(N)	O(N)

Iteration

- for loop

```
LinkedList<Integer> list = new LinkedList<Integer>();
...
for (int i=0; i<list.size(); i++) {
    Integer next = list.get(i);
}
```

- foreach loop

```
LinkedList list = new LinkedList();
...
for (Integer i : list) {
   ...
}
```

Iterators

- What if our loop changes the structure?

```
for (int i=0; i<list.size(); i++) {
    If (i==3) list.remove(i);
}
```

- Java introduced the Iterator class

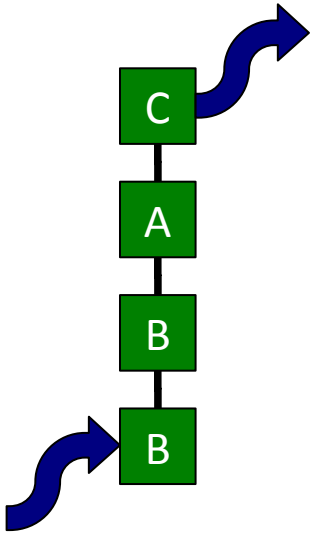
```
Iterator<Integer> it = list.iterator();

while(it.hasNext()) {Integer i = it.next();}

for (; it.hasNext(); ) {Integer i = it.next();}
```

- Safe to modify structure

```
while(it.hasNext()) {
    it.remove();
}
```

Queues

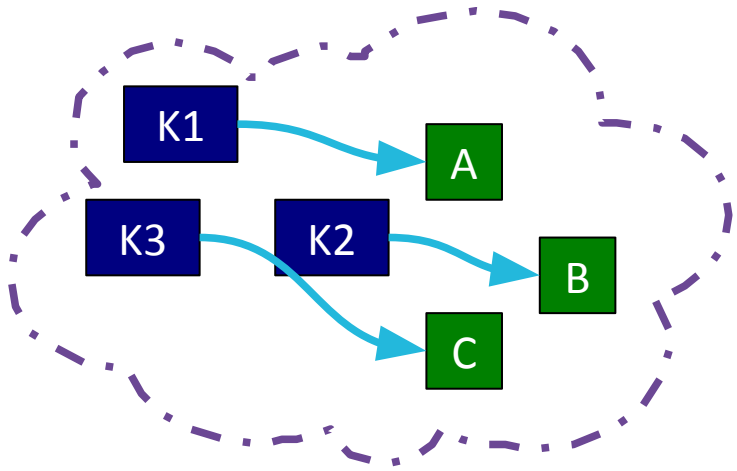
- <<interface>> Queue
  - An ordered collection of elements that may contain duplicates and supports removal of elements from the head of the queue
  - offer() to add to the back and poll() to take from the front
  - LinkedList: supports the necessary functionality
  - PriorityQueue: adds a notion of priority to the queue so more important stuff bubbles to the top




```
LinkedList<Double> ll = new LinkedList<Double>();
ll.offer(1.0);
ll.offer(0.5);
ll.poll(); // 1.0
ll.poll(); // 0.5
```

Maps

- <<interface>> Map
  - Like dictionaries in ML
  - Maps key objects to value objects
  - Keys must be unique
  - Values can be duplicated and (sometimes) null.
  - TreeMap: keys kept in order
  - HashMap: Keys not in order, efficient access (see Algorithms)



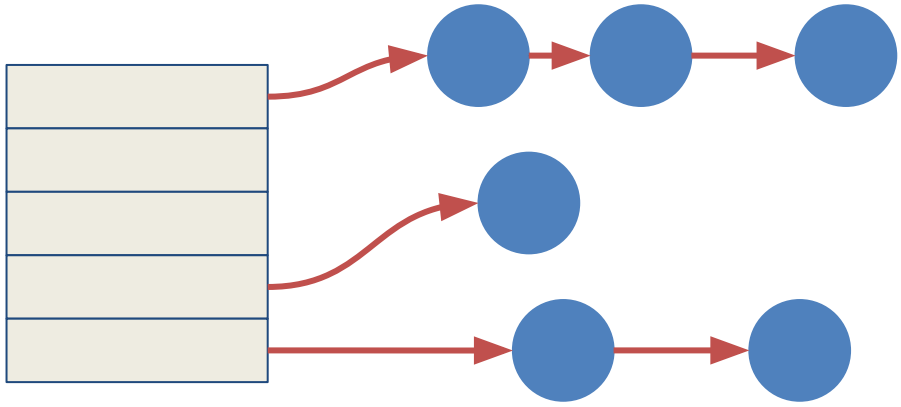

```
TreeMap<String, Integer> tm =  new TreeMap<String,Integer>();
tm.put("A",1);
tm.put("B",2);
tm.get("A");   // returns 1
tm.get("C"); // returns null
tm.contains("G");  // false
```

Hashing

Idea: somehow boil everything in an Object down to a single number in a chosen range, say 0 → 128. This number is its ***hash, h***

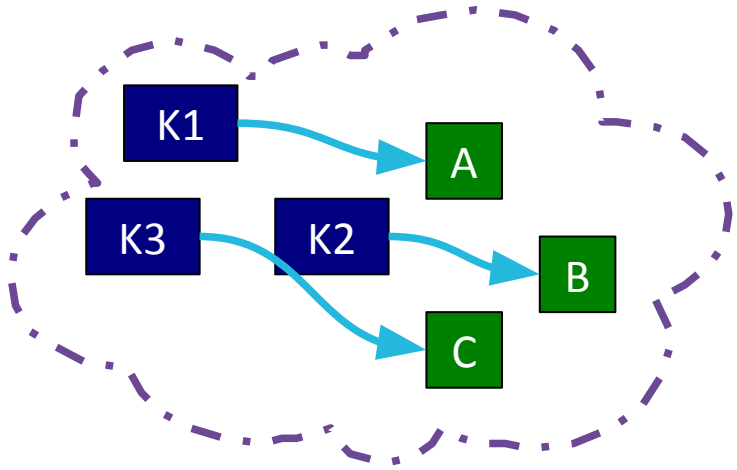
Assign the object to an array element a[h]. Then we have instant lookup for it!

Problem: either we have enormous arrays or we have multiple Objects going to the same slot



Solution: link-list the objects with the same hash

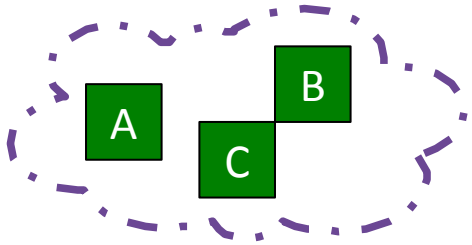
TreeMap vs HashMap



	put	get
`TreeMap`	O(lg n)	O(lg n)
`HashMap`	O(1)	O(1)

Sets

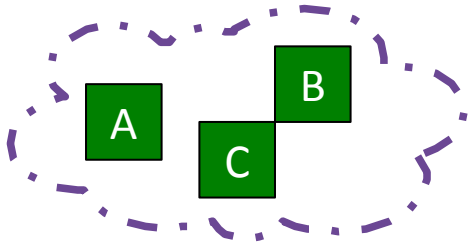
- <<interface>> Set
  - A collection of elements with no duplicates that represents the mathematical notion of a set
  - TreeSet: objects stored in order
  - HashSet: objects in unpredictable order but fast to operate on (see Algorithms course)




```
TreeSet<Integer> ts = new TreeSet<Integer>();
ts.add(15);
ts.add(12);
ts.contains(7);  // false
ts.contains(12); // true
ts.first(); // 12 (sorted)
```

TreeSet vs Hashset

- TreeSet
  - Based on a TreeMap
  - Asserts a consistent ordering
- HashSet
  - Based on a HashMap



	add	remove	contains
`TreeSet`	O(lg n)	O(lg n)	O(lg n)
`HashSet`	O(1)	O(1)	O(1)

Collections Methods

- The `Collections` class is packed with handy static methods to do things like:

- Make an unmodifiable *view*

```
ArrayList<Double> list = new ArrayList<>();
List<Double> imList = Collections.unmodifiableList(list);
list.add(6.0);  // fine
imList.add(3.0); // exception
```

- Synchronized *view* (see Part IB)

```
ArrayList<Double> list = new ArrayList<>();
ArrayList<Double> threadsafeList = Collections.synchronizedList(list);
```

Comparing Objects

Comparing Primitives

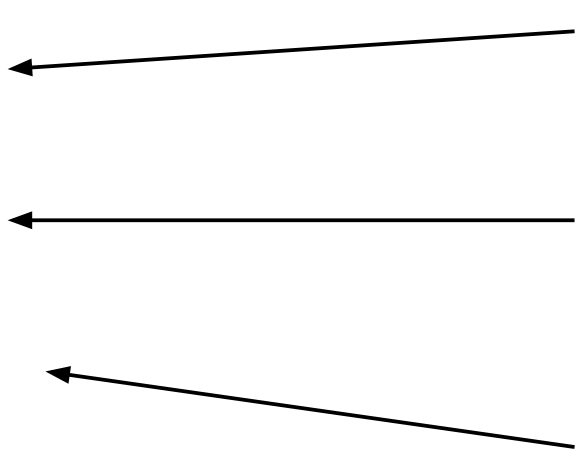
- > Greater Than
- >= Greater than or equal to
- == Equal to
- != Not equal to
- < Less than
- <= Less than or equal to

- Clearly compare the value of a primitive
- But what does (ref1==ref2) do??
  - Test whether they point to the same object?
  - Test whether the objects they point to have the same state?

Reference Equality

- r1==r2, r1!=r2
- These test *reference equality*
- i.e. do the two references point ot the same chunk of memory?

```
Person p1 = new Person("Bob");
Person p2 = new Person("Bob");

(p1==p2);
(p1!=p2);
(p1==p1);
```



Value Equality

- Use the equals() method in Object
- Default implementation just uses reference equality (==) so we have to override the method

```
public EqualsTest {
   public int x = 8;

   @Override
   public boolean equals(Object o) {
      EqualsTest e = (EqualsTest)o;
      return (this.x==e.x);
   }

   public static void main(String args[]) {
      EqualsTest t1 = new EqualsTest();
      EqualsTest t2 = new EqualsTest();
      System.out.println(t1==t2);
      System.out.println(t1.equals(t2));
   }
}
```

Back to hashCode()

- Java requires:

  if equals(o1, o2)
  then
  o1.hashCode()==o2.hashCode()

Generating hashes

- Let your IDE do the heavy lifting
- Or use java.util.Objects.hash(...);
- Always use the same fields as equals()

Comparable<T> Interface I

- int compareTo(T obj);

- Part of the Collections Framework
- Doesn't just tell us true or false, but smaller, same, or larger: useful for sorting.
- Returns an integer, r:
  - r<0 This object is less than obj
  - r==0 This object is equal to obj
  - r>0 This object is greater than obj

Comparable<T> Interface II

```
public class Point  implements Comparable<Point> {
    private final int mX;
    private final int mY;
    public Point (int, int y) { mX=x; mY=y; }

    // sort by y, then x
    public int compareTo(Point p) {
       if ( mY>p.mY) return 1;
       else if (mY<p.mY) return -1;
       else {
          if (mX>p.mX) return 1;
          else if (mX<p.mX) return -1;
          else return 0.
       }
    }
}


// This will be sorted automatically by y, then x
Set<Point> list = new TreeSet<Point>();
```

Comparator<T> Interface I

- int compare(T obj1, T obj2)

- Also part of the Collections framework and allows us to specify a specific ordering for a particular job
- E.g. a Person might have natural ordering that sorts by surname. A Comparator could be written to sort by age instead...

Comparator<T> Interface II

```
public class Person  implements Comparable<Person> {
    private String mSurname;
    private int mAge;
    public int compareTo(Person p) {
        return mSurname.compareTo(p.mSurname);
    }
}

public class AgeComparator implements Comparator<Person> {
  public int compare(Person p1, Person p2) {
     return (p1.mAge-p2.mAge);
  }
}


…
ArrayList<Person> plist = …;
…
Collections.sort(plist);   // sorts by surname
Collections.sort(plist, new AgeComparator());   // sorts by age
```

Operator Overloading

- Some languages have a neat feature that allows you to overload the comparison operators. e.g. in C++

```
class Person {
  public:
    Int mAge
    bool operator==(Person &p) {
        return (p.mAge==mAge);
    };
  }


Person a, b;
b == a;   // Test value equality
```

Generics

Reminder: Generics

Non-generics type: List

Generics: List<String>, List<Integer>

Why do we want types?

- Type systems assign types to key terms in our source code
- There are logical rules that can be applied to the types to ensure type safety and reduce the chance of bugs
- Static type checking checks types at compile (which is where we want to capture bugs!)
- Dynamic type checking checks type safety at runtime

Why do we want Generics?

- Generics are part of a type system and they aim to allow *"a type or method to operate on objects of various types while providing compile-time type safety"* [Wikipedia]
- Generics are aka Parametric Polymorphism
- In real terms it stops a specific type of error at compile time…

Collections are a good motivator

```
// Make a TreeSet object
TreeSet ts = new TreeSet();

// Add integers to it
ts.add(new Integer(3));

// Loop through
iterator it = ts.iterator();
while(it.hasNext()) {
    Object o = it.next();
    Integer i = (Integer)o;
}
```

- The original Collections framework just dealt with collections of Objects
  - Everything in Java "is-a" Object so that way our collections framework will apply to any class
  - But this leads to:
    - Constant casting of the result (ugly)
    - The need to know what the return type is
    - Accidental mixing of types in the collection

Collections are a good motivator

```
// Make a TreeSet object
TreeSet ts = new TreeSet();

// Add integers to it
ts.add(new Integer(3));
ts.add(new Person("Bob"));

// Loop through
iterator it = ts.iterator();
while(it.hasNext()) {
    Object o = it.next();
    Integer i = (Integer)o;
}
```

Going to fail for the second element!
(But it will compile: the error will be at runtime)

Generics lets us catch errors at compile

```
List<String> list = new ArrayList<>();
list.add("a");
list.add("b");
list.add(8); // compile error
```

(Note the shorthand of using <> instead of <String> on the RHS: Java fills in the type from the LHS)

Declaring a Generic class

```
public class Box<T> {

    private T t;

    public Box(T t) {
        this.t = t;
    }

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}
```

There's nothing special about 'T' - you can use what you like

Bounded Parameters

```
public class Box<T extends Number> {

    private T t;

    public Box(T t) {
        this.t = t;
    }

    public void set(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}
```

```
Box<Integer> box1; // ok
Box<String> box2; // error
Box<BigInteger> box3; // ok
Box<Object> box4; // error
Box<Double> box5; // ok
```

Methods too!

```
public class GenericMethod {

    public static <T> void fillList(
                        List<T> list,
                        T val){
            for (int i=0; i<list.size(); i++)
                list.set(i, val);
    }
}
```

Java's Generics Implementation

Java's goal

Add generics to its type safety system, retaining backwards compatibility

Option 1: templates

The generic class is treated as a template by the compiler, which generates new classes from it whenever you ask for something

E.g. If your code contains ArrayList<Integer> it would generate a Java class for ArrayListInteger or some such

Essentially you search/replace the template param, T, with "Integer" to get a new class. Repeat for any other types used in the code

C++ does this

Option 1: templates

```
class MyClass<T> {
  T membervar;
};
```

```
class MyClass_float {
  float membervar;
};
class MyClass_int {
  int membervar;
};
class MyClass_double {
  double membervar;
};
...
```

C++ does this

Option 2: type erasure

At compile time, do all the type checks you can.

Then **delete** the type information in the compiler output.

I.e. ArrayList<Integer> is checked, and then written to bytecode as plain ArrayList. The JVM will never know and so dynamic checks aren't possible.

Java does this

Option 2: type erasure

```
LinkedList<Integer> ll =
    new LinkedList<Integer>();

…

for (Integer i : ll) {
    do_sthing(i);
}
```

```
LinkedList ll =
    new LinkedList();

…

for (Object i : ll) {
    do_sthing( (Integer)i );
}
```

Pros/Cons of Type Erasure

Pros

- Bytecode unchanged: backwards compatible
- Compile time type checking reduces bugs
- Avoids bloat of templates (all those extra classes)

Cons

- No runtime (dynamic) checking
- Has some unexpected consequences…

You can't use primitive parameters

- The compiler replaces the template parameter with Object

  ```
  T memberVar → Object memberVar
  ```

- Obviously can't work for primitives (which don't descend from Object)

Creation is tricky

```
T memberVar = new T();
```

Objects are created via `new` at runtime in the JVM

But the JVM doesn't know what T was (it's been erased after compile). So all it can do is:

```
Object memberVar = new Object();
```

Which isn't particularly useful…

Method overloading is limited

```
void addAll(List<String> items) {...}

void addAll(List<Integer> items) {...}
```

While the raw code has distinguishable types in its argument list, they will both erase to LinkedList so you just can't do this:

```
void addAll(List items) {...}

void addAll(List items) {...}
```

Generics, Inheritance and Covariance

Covariance

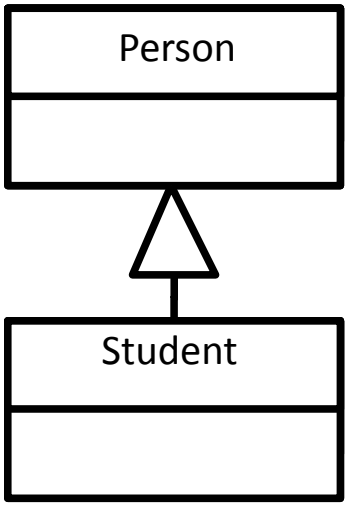
Covariance: If B is a subtype of A then I should be able to use B everywhere I expect an A.

(Think Liskov substitution principle)

Java classes are covariant

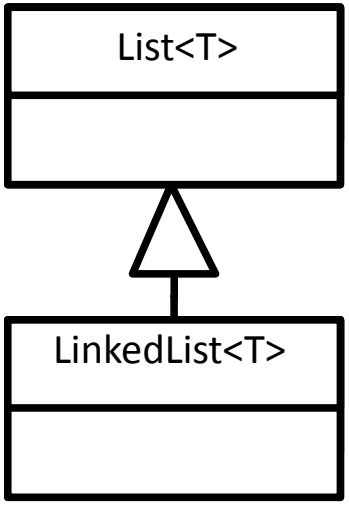
```
Student s = new Student();
Person p = (Person) s;  // fine
```

Covariance




```
// Object casting
Student s = new Student();
Person p = (Person) s;
```

A Student is-a Person




```
// List casting
LinkedList<Person> pllist = new LinkedList<Person>();
List<Person> plist = (List<Person>) playlist;
```

A LinkedList of Person is a List of Person

Java Arrays are covariant

Java arrays are also covariant

If B is a subtype of A then I should be able to use B[] everywhere I expect an A[].

```
Student[] sarray= new Student[20];
Person[] parray = (Person[]) sarray;  // fine
```

But is this right??

```
Student[] sarray= new Student[20];
Person[] parray = (Person[]) sarray;  // fine
```

Is an array of students an array or persons?

But is this right??

```
Student[] sarray= new Student[20];
Person[] parray = (Person[]) sarray;  // fine
```

Is an array of students an array or persons?

NO

But is this right??

```
Student[] sarray= new Student[20];
Person[] parray = (Person[]) sarray;

Parray[2] = new Lecturer(); // Eeek!!!
```

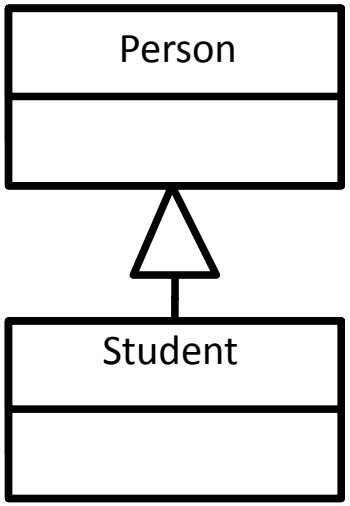
If you try this, it will compile, but you get a nasty runtime error

Java Arrays are reified

So covariance in arrays opens up the possibility of runtime errors

This is possible because arrays know their type at runtime - they are **reified**

What about Generics?




```
// This compiles
Student[] s = new Student[20];
Person[] p = (Person[])s;

// This doesn't compile
List<Student> s = new LinkedList<Student>();
List<Person> s = (List<Person>)p;
```

Generics are **invariant**: why the different approach?

Conceptual: making arrays coovariant was arguably wrong and it led to runtime errors

Practical: Generic types are erased so even if they were covariant, the runtime wouldn't know if we did something bad → no runtime error, just nasty effects later in your program!!

This presents a problem…

Imagine you wanted a function that can be handed a list and will just print out everything in it, regardless of type

```
public void printList( ?? )
```

Wildcards

We can use wildcards to do this

```
void printAll (List<?> list) {
   for (Object o : list)
      System.out.println(o);
}
```

You can call anything on list that returns the underlying type e.g. **`list.get(3);`**

If you call anything that takes the type as input, it won't compile e.g.

`list.add("hi");`

Bounded Wildcard: Lower

<? extends A> matches anything that is type A or a subtype of it (Covariance)

```
public void printNumberList (
        List<? extends Number> list) {

   for (Number n: list) {
      System.out.println(n);
    }
}
```

Bounded Wildcard: Lower

```
List<? extends Number>
```

It's safe to **read** Number types from this

It's dangerous to **write anything** to this (you can't tell if it's Double or Integer, etc)

Bounded Wildcard: Upper

`<? super A>` matches anything that is type A or a supertype of it (contravariance)

```
List<? super Number>
```

It's only safe to read Objects from this

It's safe to write Number or its **sub**classes

Bounded Wildcard: Both

```
public static void copy(
   List<? extends Number> src,
   List<? super Number> dest) {

   for(Number number : src)
      dest.add(number);

}
```

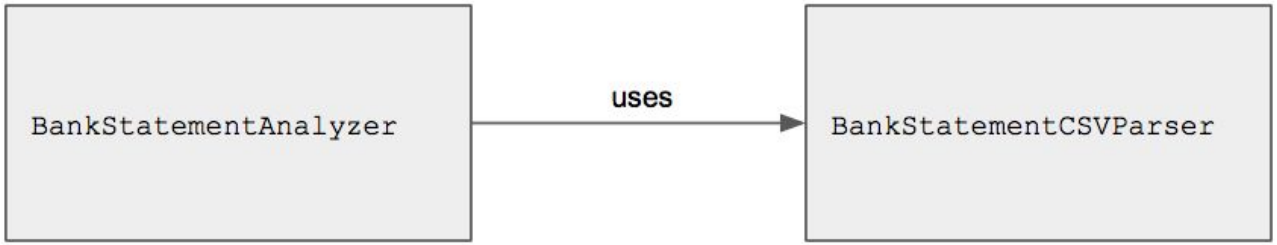
Coupling, Errors and Exceptions

Coupling

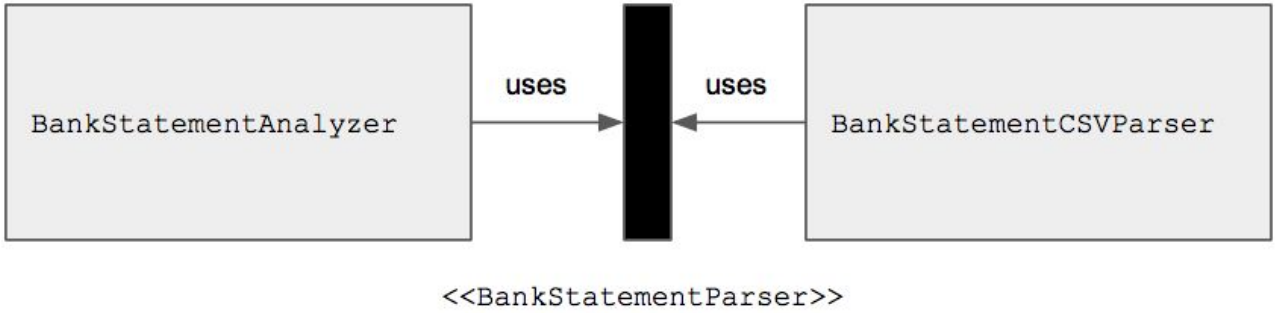
- Degree to which different parts of a program depend on each others
- **High coupling**: relying on internals/implementation details
- **Loose coupling**: relying on interface and defined behaviour
  - No need to know how a smartwatch works to read the time
  - Changes to the watch's internals (software) do not affect reading the time

Coupling

High coupling



Low coupling



Bad Coupling

- Relying on internal implementation details which may change
- Accessing / updating poorly encapsulated fields
- Reckless use of inheritance introduces high coupling between two classes because if the parent class changes (fields, methods…) it could affect the children classes

Boxing

Boxing and Unboxing

- Java automatically converts between the primitive types and their corresponding object wrapper classes to make life simpler. This is called *autoboxing*.
- **Boxing**: turn an `int` into an `Integer`
- **Unboxing**: turn an `Integer` into an `int`

Note that boxed objects have more memory overhead!

An `int` takes up 4 bytes

An `Integer` ~16 bytes (special headers and flags to be an Object)

Auto-Boxing

```
public void something(Integer I) {

}

int i = 4;

something(i); // works: auto-boxing
```

Auto-Unboxing

```
public void other(int i) {

}

Integer i = 3;

other(i); // auto-unboxing
```

Auto-Unboxing Warning

```
public void other(int i) {
}
Integer i = null;
other(i); // auto unbox gives NPE!
```

(if you'd tried other(null) it would not have compiled…)

Errors

Return Codes

The traditional imperative way to handle errors is to return a value that indicates success/failure/error

```
public int divide(double a, double b) {
  if (b==0.0) return -1; // error
  double result = a/b;
  return 0; // success
}

…

if ( divide(x,y)<0) System.out.println(“Failure!!”);
```

Problems with Return Codes

Could (and often do) ignore the return value

Have to keep checking what the return values are meant to signify, etc.

The actual result often can't be returned in the same way

Error handling code is mixed in with normal execution *(makes code harder to read and hence maintain)*

Example

```
#include <stdio.h>
#include <stdlib.h>


int main() {
    const char* filename = "example.txt";
    FILE* file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return EXIT_FAILURE;
    }
    char buffer[100];
    while (fgets(buffer,
          sizeof(buffer), file) != NULL) {
        // Check for read errors
        if (ferror(file)) {
            perror("Error reading from
file");
            fclose(file);
            return EXIT_FAILURE;
        }
        printf("%s", buffer);
    }
    if (ferror(file)) {
        perror("Error reading from
file");
        fclose(file);
        return EXIT_FAILURE;
    }

    // Close the file
    if (fclose(file) != 0) {
        perror("Error closing file");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS; }
```

Example

```
#include <stdio.h>
#include <stdlib.h>


int main() {
    const char* filename = "example.txt";
    FILE* file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return EXIT_FAILURE;
    }
    char buffer[100];
    while (fgets(buffer,
          sizeof(buffer), file) != NULL) {
        if (ferror(file)) {
            perror("Error reading from
file");
            fclose(file);
            return EXIT_FAILURE;
        }
        printf("%s", buffer);
    }
    if (ferror(file)) {
        perror("Error reading from
file");
        fclose(file);
        return EXIT_FAILURE;
    }

    if (fclose(file) != 0) {
        perror("Error closing file");
        return EXIT_FAILURE;
    }

    return EXIT_SUCCESS; }
```

Deferred Error Handling

A similar idea (with the same issues) is to set some state in the system that needs to be checked for errors.

C++ does this for streams:

```
ifstream file( "test.txt" );
if ( file.good() )
{
        cout << "An error occurred opening the file" << endl;
}
```

Exceptions

- An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- An exception is an object that can be *thrown* or *raised* by a method when an error occurs and *caught* or*handled* by the calling code
- Example usage:

```
try {
   double z = divide(x,y);
}
catch(DivideByZeroException d) {
   // Handle error here
}
```

Flow Control During Exceptions

- When an exception is thrown, any code left to run in the try block is skipped

```
double z=0.0;
boolean failed=false;
try {
   z = divide(5,0);
   z = 1.0;
}
catch(DivideByZeroException d) {
   failed=true;
}
z=3.0;
System.out.println(z+” “+failed);
```

Throwing Exceptions in Java

- An exception is an object that has `Exception` as an ancestor
- So you need to create it (with new) before throwing

```
double divide(double x, double y) throws DivideByZeroException {
  if (y==0.0) throw new DivideByZeroException();
  else return x/y;
}
```

Multiple Handlers

- A try block can result in a range of different exceptions. We test them in sequence

```
try {
   FileReader fr = new FileReader(“somefile”);
   int r = fr.read();
}
catch(FileNotFound fnf) {
   // handle file not found by the FileReader
}
catch(IOException d) {
   // handle read() failed
}
```

Union catch blocks

- You can catch multiple in the same block

```
try {
   FileReader fr = new FileReader(“somefile”);
   int r = fr.read();
}
catch(FileNotFound fnf | SomeOtherException e) {
   // handle  the same way
}
catch(Exception d) {
   // handle anything else
}
```

finally

With resources in particular we often want to ensure that they are closed whatever happens

An use a `finally` block that will always run (after any handler)

```
static String readFirstLineFromFile(String path) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

Try-with-resources

Still easy to forget the finally block

Try-with-resources in java does it for us

```
try (BufferedReader br = new BufferedReader(new FileReader(path)))
{

    return br.readLine();

}
```

The objects we create in the try brackets must implement `AutoCloseable` so that the compiler can insert a finally block that does the close for us

Creating Exceptions

Just extend `Exception` (or `RuntimeException` if you need it to be unchecked - see later). Good form to add a detail message in the constructor but not required.

```
public class DivideByZero extends Exception {}

public class ComputationFailed extends Exception {
    public ComputationFailed(String msg) {
        super(msg);
    }
}
```

You can also add more data to the exception class to provide more info on what happened (e.g. store the numerator and denominator of a failed division)

Exception Inheritance Hierarchies

You can use inheritance hierarchies

```
public class MathException extends Exception {...}
public class InfiniteResult extends MathException {…}
public class DivByZero extends MathException {…}
```

And catch parent classes

```
try {
   …
}
catch(InfiniteResult ir) {
   // handle an infinite result in a special way
}
catch(MathException me) {
  // handle any MathException or DivByZero
}
```

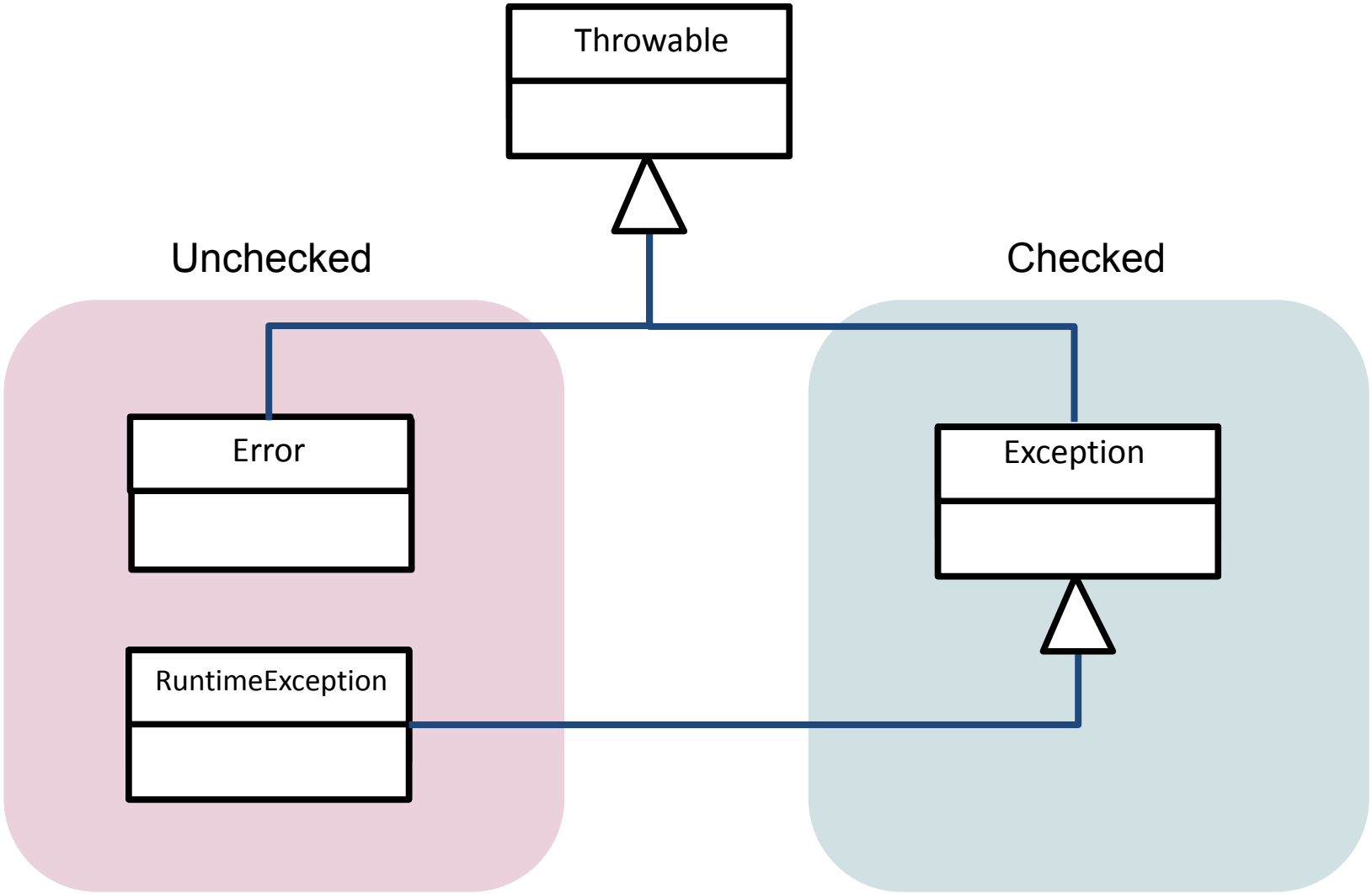
Exception Benefits in Java

1. **Documentation**: The language supports exceptions as part of method signatures

2. **Type safety**: The type system figures out whether you are handling the exceptional flow

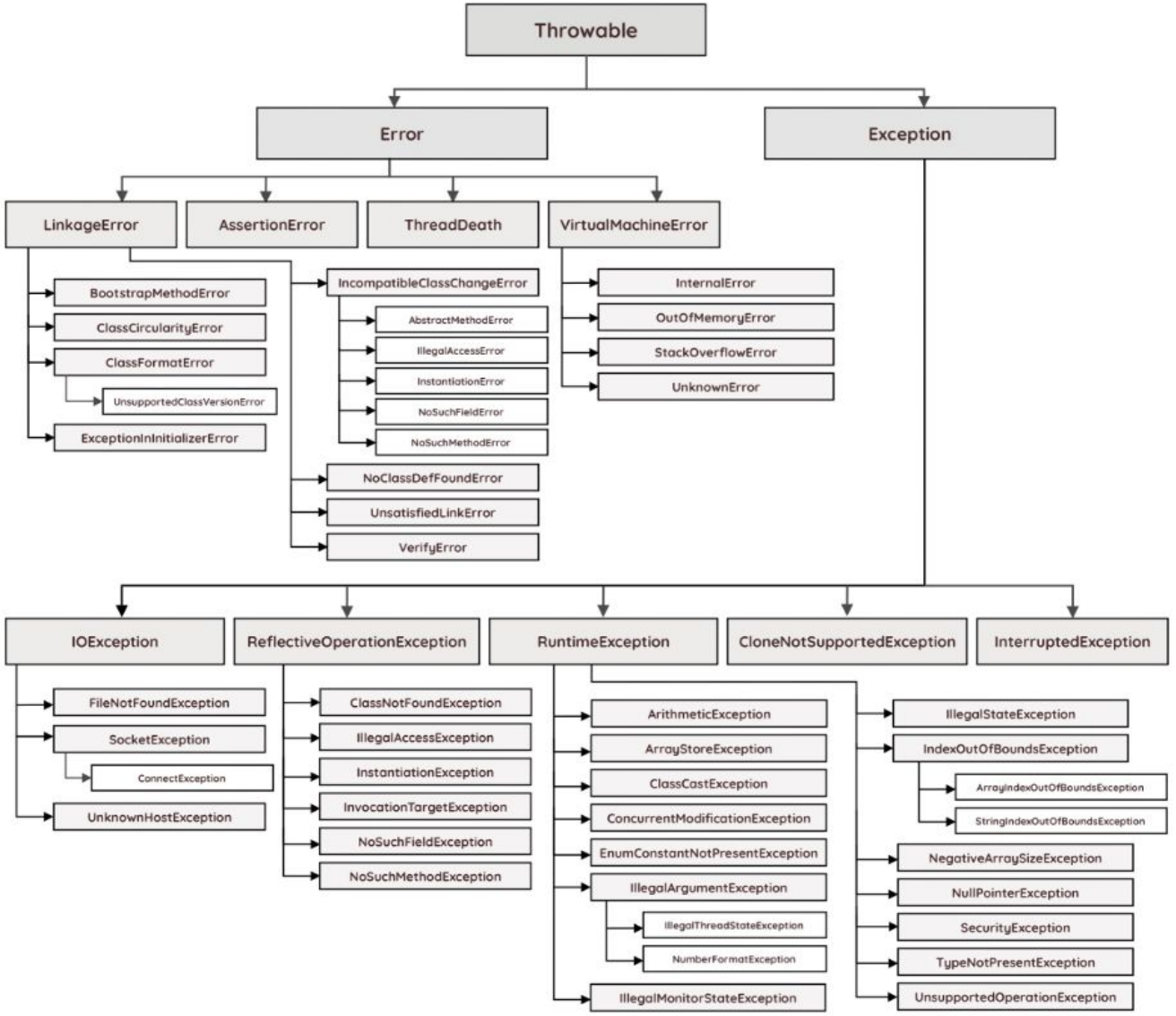
3. **Separation of concern**: business logic and exception recovery are separated out with a `try` and `catch` block

Java's Error Hierarchy



Error is intended for things completely outside of the programmer's control; Exceptions are for problems ultimately caused by the programmer in some way

Java's Error Hierarchy



Checked vs Unchecked Exceptions

- Checked: must be handled or passed up.
  - Client must take a recovery action (e.g. display a message or retry)
  - Java requires you to declare checked exceptions that your method throws
  - Java requires you to catch the exception when you call the function

- Unchecked: not expected to be handled.
  - Programming error (e.g. null or wrong pattern / format)
  - There's nothing the client could do (e.g. system error)
  - Extends `RuntimeException`
  - Good example is `NullPointerException`

Guidelines for Exception use

1. **Never** ignore an exception

If no handling mechanism, re-throw unchecked (“Exception translation”)

```
catch (WeirdException e) {
    // TODO. I'll deal with this later. Maybe..
}
```

vs.

```
try {
  callToAPI();
} catch (WeirdException exception) {
  throw new RuntimeException(exception);
}
```

2. Do not catch Exception…

…that would swallow up Runtime exception too

```
catch (Exception e) { /* TODO (yeah right) */}
```

vs.

```
catch(WeirdException e) {

}
```

Catch specific exceptions to improve readability and provide more specific exception handling

3. Document exceptions at the API level

Java supports specially formatted comments on classes and methods that are used to produce pretty API webpages (“Javadoc”).

Use this facility fully to describe when exceptions would occur in the context of the method

```
/**
 * Parses a CSV settings file into an AppSettings object
 *
 * @param filename Full path to file
 * @return the parsed settings as an Appsettings object
 * @throws NoSuchfileexception if the filename supplied is invalid
 * @throws Badinput if the settings file is corrupt.
 **/
public AppSettings loadSettings(String filename) {

  …

}
```

4. Avoid implementation-specific exceptions

If your exception describes what's going on under the hood, you are breaking encapsulation E.g.

```
public String read(Source source) throws
SQLException
```

Clearly relates to an implementation detail (it's using SQL), which is not helpful to a user or your class/API.

```
public String read(Source source) throws
ResourceNotFoundException
```

Is much more useful

5. **Never** use exceptions for control flow

```
try {
    while (true) {
        System.out.println(iterator.next());
    }
}
catch(NoSuchElementException e) {
   // All done
}
```

It's just evil

Assertions

Assertions

- Assertions are a form of error checking designed for debugging (only). We met them in the Bootcamp.

- They are a simple statement that evaluates a boolean: if it's true nothing happens, if it's false, the program ends.

- In Java:

```
assert (x>0);

// or

assert (a==0) : “Some error message here”;
```

Assertions are NOT for Production Code!

- Assertions are there to help you check the logic of your code is correct i.e. when you're trying to get an algorithm working

- **They should be switched OFF** for code that gets released ("production code")

- In Java, the JVM takes a parameter that enables (-ea) or disables (-da) assertions. The default is for them to be disabled.

-
```
> java -ea SomeClass

> java -da SomeClass
```

As Oracle Puts It

"Assertions are meant to require that the program be consistent with itself, not that the user be consistent with the program"

Great for Postconditions

- Postconditions are things that must be true at the end of an algorithm/function if it is functioning correctly
- E.g.

```
public float sqrt(float x) {
   float result = ….
   // blah
   assert(result>=0.f);
}
```

Sometimes for Preconditions

- Preconditions are things that are assumed true at the start of an algorithm/function
- E.g.

```
private void method(SomeObject so) {
  assert (so!=null);
  //...
}
```

- **BUT you shouldn't** use assertions to check for **public** preconditions

```
public float method(float x) {
  assert (x>=0);
  //...
}
```

- (you should use exceptions for this)

Sqrt Example

```
public float method(float x) throws InvalidInputException {
  .// Input sanitisation (precondition)
  if (x<0.f) throw new InvalidInputException();

  float result=0.f;
  // compute sqrt and store in result

  // Postcondition
  assert (result>=0);

  return result;
}
```

Assertions can be slow if you Like

```
public int[] sort(int[] arr) {
   Int[] result = ...
   // blah
   assert(isSorted(result));
}
```

- Here, isSorted() is presumably quite costly (at least O(n)).
- That's OK for debugging (it's checking the sort algorithm is working, so you can accept the slowdown)
- And will be turned off for production so that's OK

- *(but your assertion shouldn't have side effects)*

NOT for Checking your Compiler/Computer

```
public void method() {
   Int a=10;
   assert (a==10);
   //...
}
```

- If this isn't working, there is something much bigger wrong with your system!
- It's pointless putting in things like this

Design Patterns

Design Patterns

- A Design Pattern is a general reusable solution to a commonly occurring problem in software design
- Coined by Erich Gamma in his 1991 Ph.D. thesis
- Originally 23 patterns, now many more. Useful to look at because they illustrate some of the power of OOP (and also some of the pitfalls)
- We will only consider a subset

The Open-Closed Principle

- ***Classes should be open for extension but closed for modification***

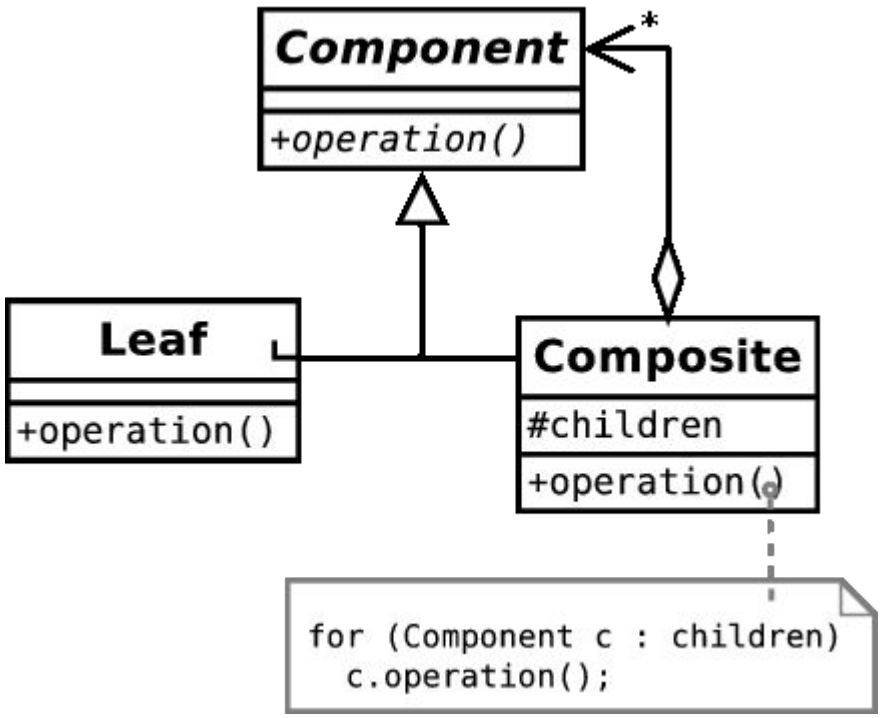
- i.e. we would like to be able to modify the behaviour without touching its source code
- This rule-of-thumb leads to more reliable large software and will help us to evaluate the various design patterns

Composite

- Abstract problem: How can we treat a group of objects as a single object?

- Example problem: Representing a DVD box-set as well as the individual films without duplicating info and with a 10% discount

Composite in General



- The composite pattern lets us treat objects and groups of objects uniformly

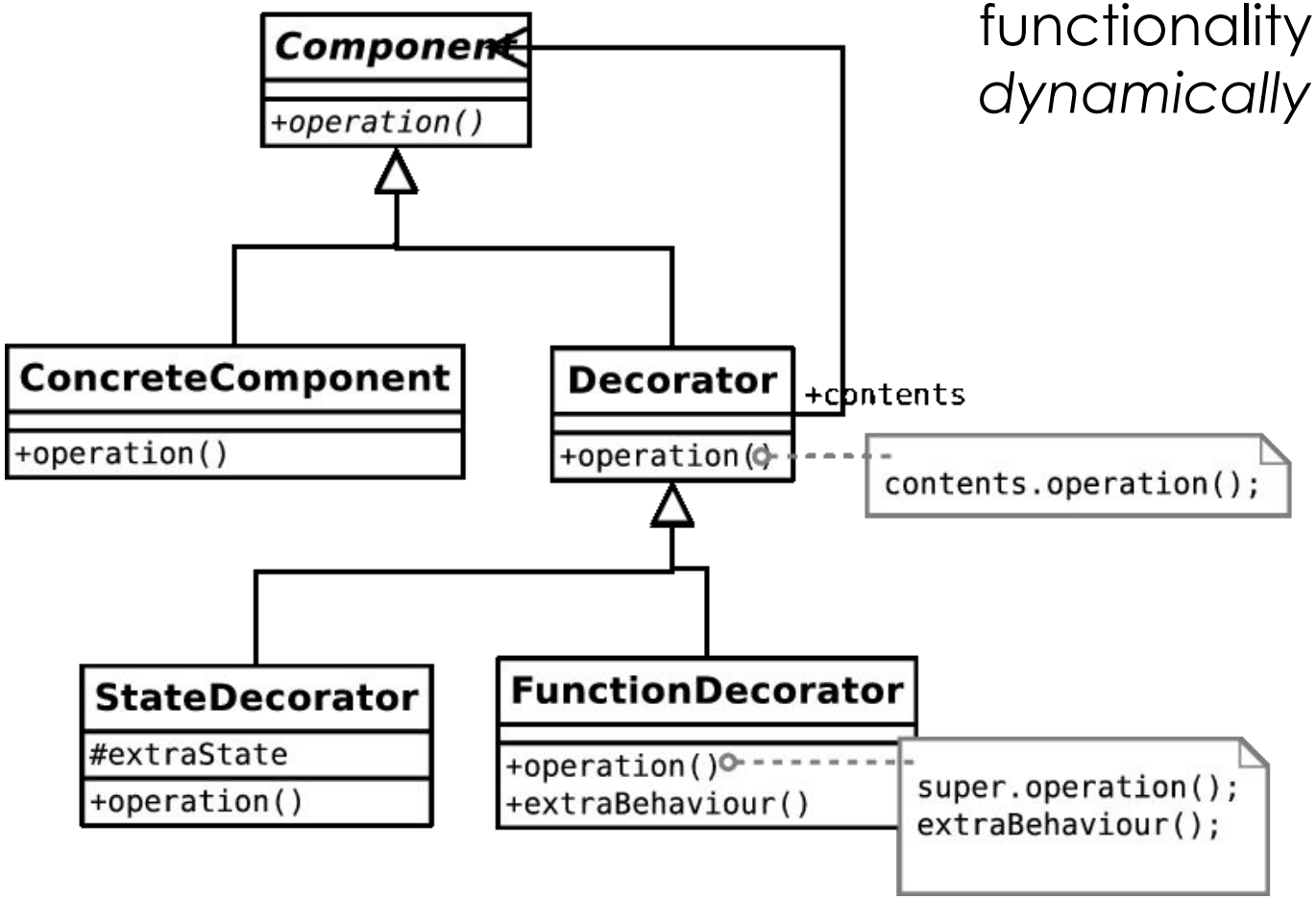
Decorator

- Abstract problem: How can we add state or methods at runtime?

- Example problem: How can we efficiently support gift-wrapped books in an online bookstore?

Decorator in General

- The decorator pattern adds state and/or functionality to an object *dynamically*

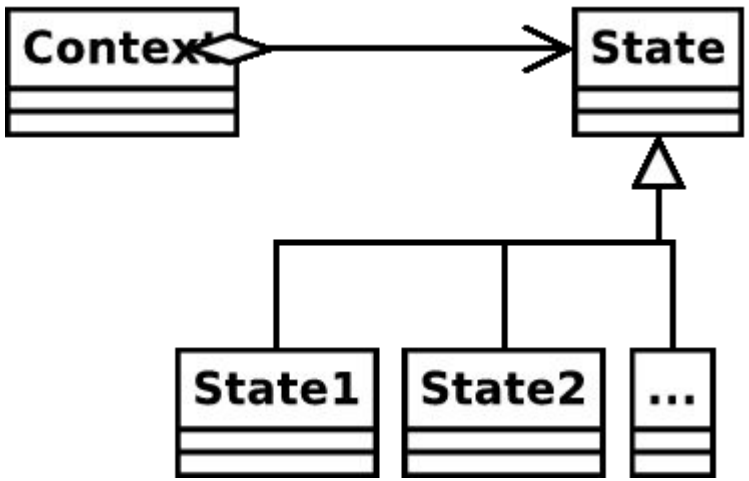


State

- Abstract problem: How can we let an object alter its behaviour when its internal state changes?

- Example problem: Representing academics as they progress through the rank

State in General



- The state pattern allows an object to cleanly alter its behaviour when internal state changes

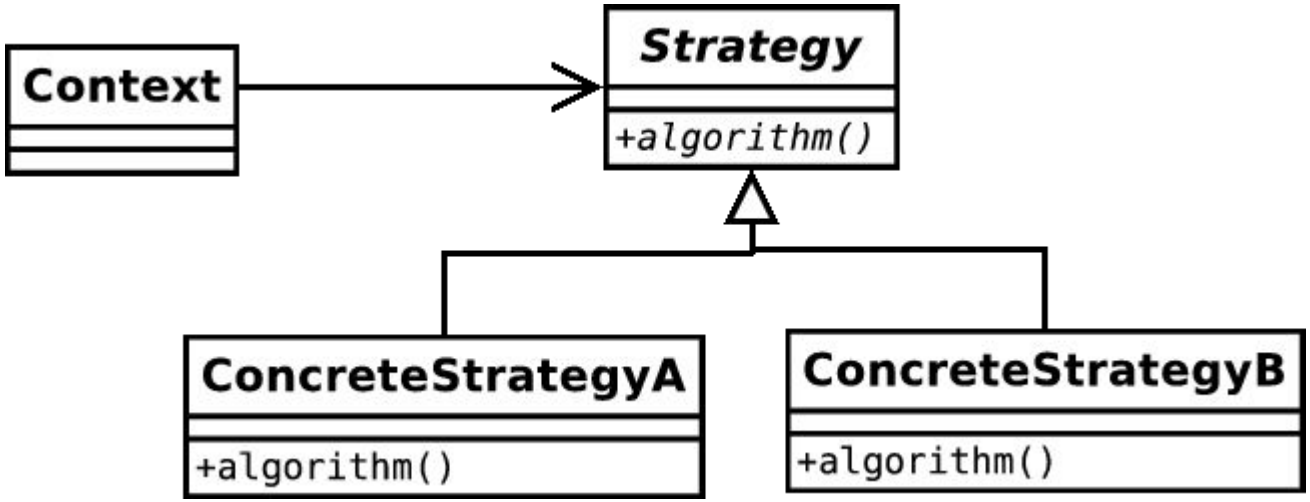
Strategy

- Abstract problem: How can we select an algorithm implementation at runtime?

- Example problem: We have many possible change-making implementations. How do we cleanly change between them?

Strategy in General

- The strategy pattern allows us to cleanly interchange between algorithm implementations

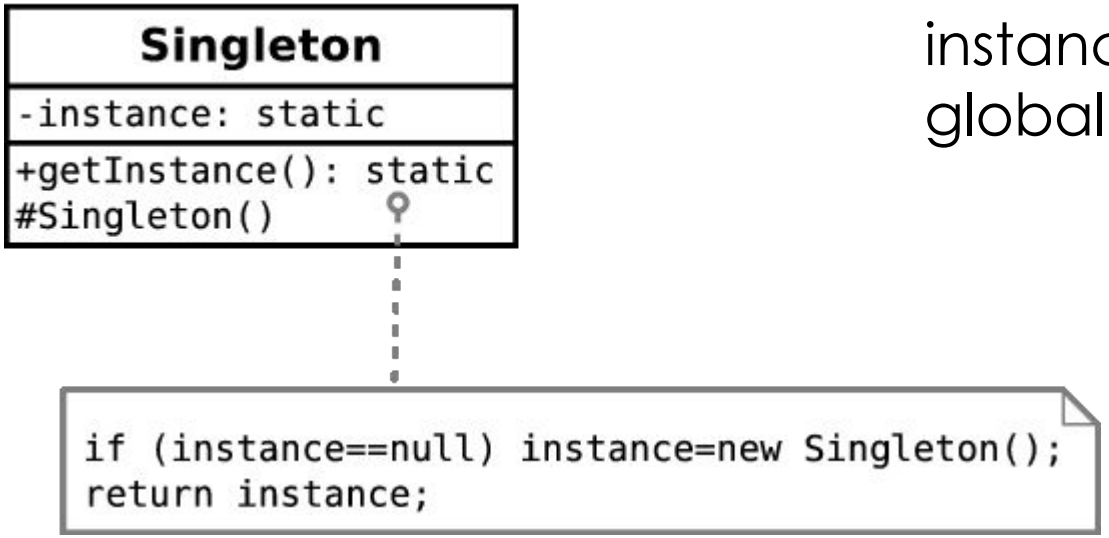


Singleton

- Abstract problem: How can we ensure only one instance of an object is created by developers using our code?

- Example problem: You have a class that encapsulates accessing a database over a network. When instantiated, the object will create a connection and send the query. Unfortunately you are only allowed

Singleton in General



- The singleton pattern ensures a class has only one instance and provides global access to it

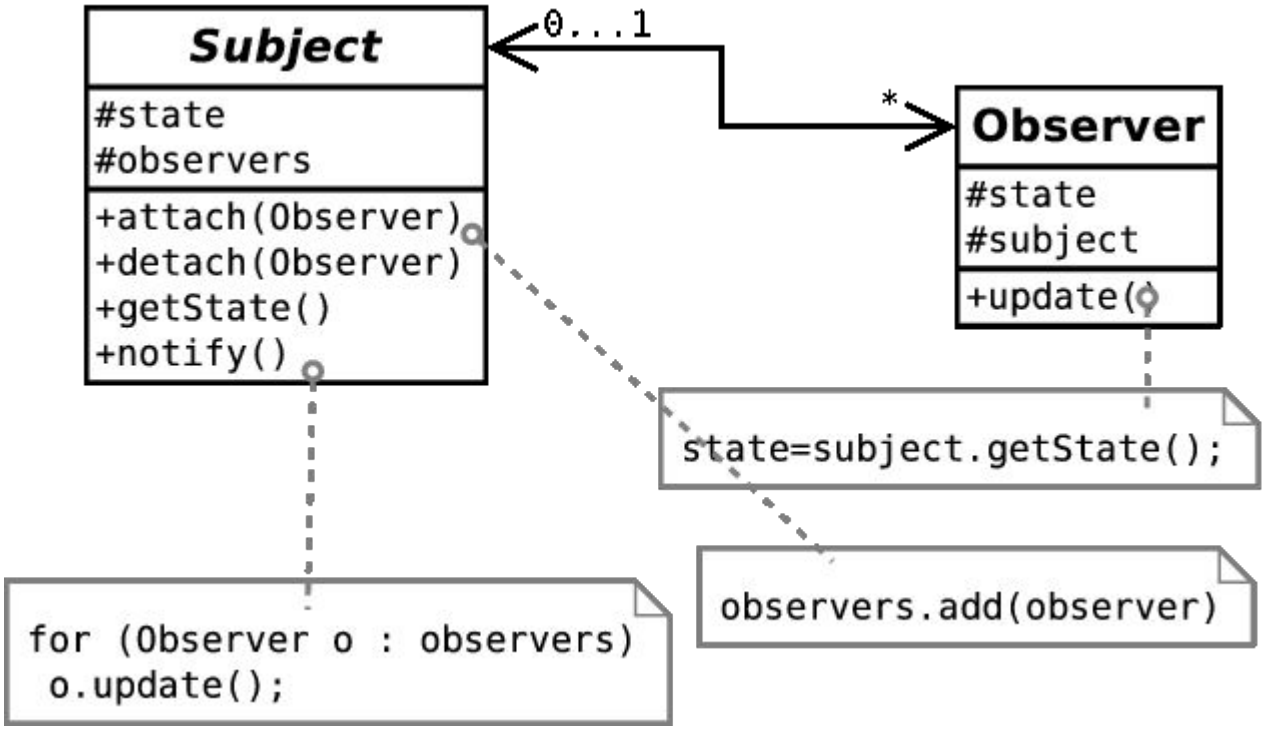
Observer

- Abstract problem: When an object changes state, how can any interested parties know?

- Example problem: How can we write phone apps that react to accelerator events?

Observer in General

- The observer pattern allows an object to have multiple dependents and propagates updates to the dependents automatically.



Takeaways

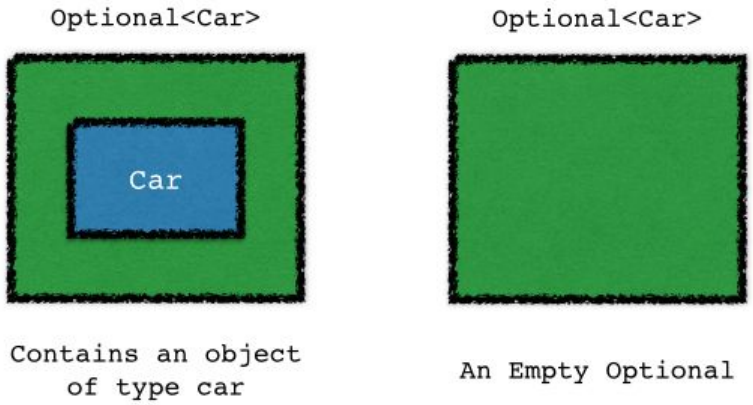
- We covered the following patterns:
  - Singleton
  - Decorator
  - Composite
  - State
  - Strategy
  - Observer

Bonus: Optional<T>

- Not an official design Pattern, but worth us talking about
- Using null as a way to indicate the lack of an object is rather risky: you've all seen NullPointerExceptions!
- In general null has three problems:
  - Error-pone checking
  - Verbose checking
  - No useful semantic meaning

Bonus: Optional<T>

- java.util.Optional<T> encapsulates an optional value
- You can view Optional as a single-value container that either contains a value or doesn't



Bonus: Optional<T>

- More comprehensible model where it's immediately understandable whether to expect an optional value → better maintainability
- You need to actively unwrap an Optional to deal with the absence of a value → fewer errors

```
String name = "hello";
Optional<String> opt = Optional.of(name);
if (opt.isPresent()) {
   // code here
}
```

Lambdas, Method References, Streams

Lambdas

Task

```
public class Apple {
    private String colour;
    private double weight;

    public String getColour() { return colour; }
    public double getWeight() { return weight; }

    // Constructors etc.
}
```

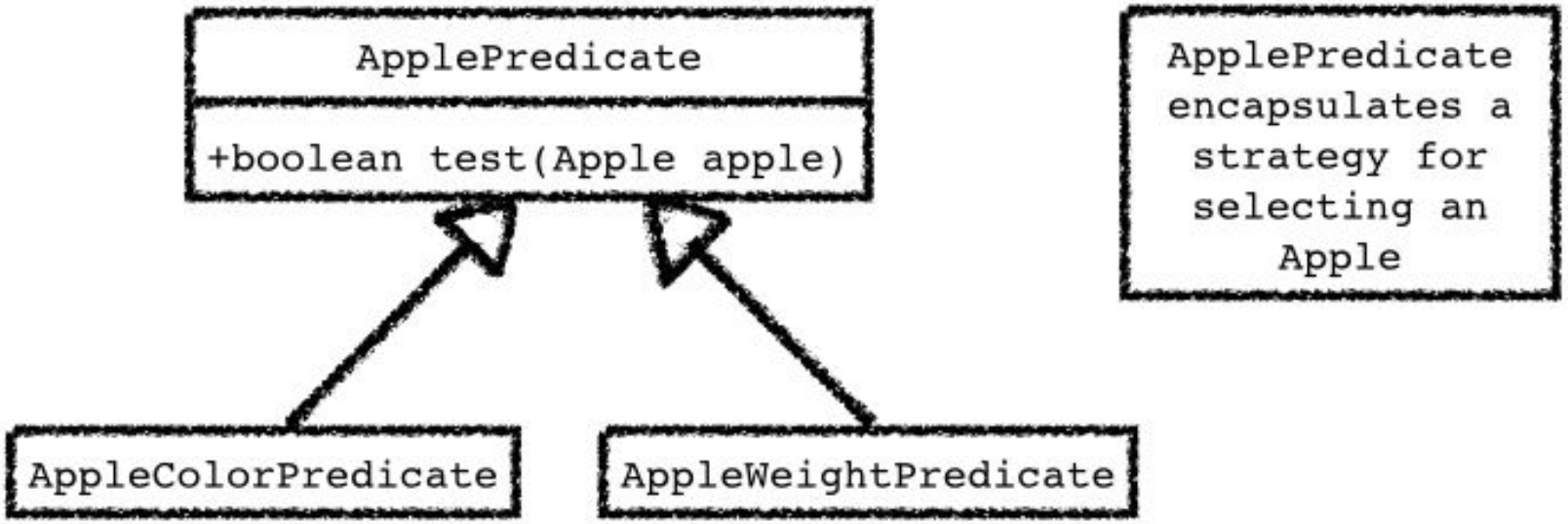
You have a `List<Apple>` and you need to filter it by colour and weight. How do you do it?

Option 1

```
public static List<Apple> filterApplesByColour( List<Apple> apples,
String colour) {

    List<Apple> result = new ArrayList<>();
    for (Apple apple: apples) {
        if (apple.getColour().equals(colour))) {
            result.add(apple); }
    }
    return result;
}

List<Apple> greenApples = filterApplesByColor(inventory, "green");
```

Quite a lot of code for such as simple thing. Even uglier if we extend to filter by weight too

Option 2: Strategy Pattern?




```
public interface ApplePredicate {
    boolean test (Apple apple);
}
```

Option 2: Strategy Pattern?

Create specific predicates for what we want:

```
public class AppleWeightPredicate implements ApplePredicate {
    public boolean test(Apple apple){
        return apple.getWeight() > 150;
    }
}
public class AppleGreenPredicate implements ApplePredicate {
    public boolean test(Apple apple){
        return "green".equals(apple.getColour());
    }
}
```

Option 2: Strategy Pattern?

Now filter:

```
public static List<Apple>  filter(List<Apple> inventory, ApplePredicate p) {
    List<Apple> result = new ArrayList<>();
    for(Apple apple: inventory){
        if(p.test(apple))   result.add(apple);
    }
    return result;
}

List<Apple> greenApples = filter(inventory, new AppleGreenPredicate());
List<Apple> greenLightApples = filter(greenApples, new
AppleWeightPredicate());
```

Is this good?

Option 2: Strategy Pattern?

Pros:

- Increased code flexibility. Easy-ish to write new predicates
- `filter` code is universal

Cons:

- A lot of code, with an annoying overhead of a class for every new predicate. Especially annoying if it's a one-off filter we won't reuse
- We've got a good ***abstraction***, but we have poor ***concision***

Option 3: Anonymous Classes

To partly address these issues, Java allows you to define an 'anonymous class' inline in your code

```
List<Apple> result = filter(inventory,
    new ApplePredicate() {
        public boolean test(Apple apple){
            return "red".equals(apple.getColor());
        }
    }
);
```

Helps with the one-off issue, and is generally a bit more concise

Still feels verbose, however.

Option 4: Lamdas!

```
List<Apple> result = filter(inventory,
    (Apple apple) -> "red".equals(apple.getColour()));
```

Flexible **and** concise!! The lambda defines a function without all the boilerplate using the syntax

```
(parameters) -> expression
```

or

```
(parameters) -> { statement1; statement2; …}
```

But wait: how does it know how to create an ApplePredicate object from just that??

Option 4: Lamdas!

The trick is ApplePredicate is an interface with **exactly one** method:

```
public interface ApplePredicate {
    boolean test (Apple apple);
}
```

When the compiler sees

```
(Apple apple) -> "red".equals(apple.getColour())
```

it knows it must be defining precisely `test` (because there's nothing else *to* define). It checks the argument list matches `test` (it does) and generates the rest for us.

This would not be possible if ApplePredicate had more than one function…

Functional Interfaces

This trick requires strict one-method-only interfaces to work and is very powerful as we've seen

We name such interfaces **Functional Interfaces**

What is a Lambda?

- a kind of anonymous function
- that can be passed around (OCaML, anyone?)
- it doesn't have a name, but it has a list of parameters, a body, a return type, and also possibly a list of exceptions that can be thrown.

```
(parameters) -> expression
```

or

```
(parameters) -> { statements; }
```

Abstracting further

Our filter function could be even more generic, and not be limited to Apples by using Generics:

```
public static <T> List<T> filter (List<T> list,
                                  Predicate<T> p) {
    List<T> result = new ArrayList<>();
    for(T e: list) {
        if(p.test(e)) {
            result.add(e);
        }
    }
    return result;
}
```

Abstracting further

Now it's flexible and really concise:

```
List<String> result =
    filter(strings, (String s) ->  s.endsWith(".json"));


List<Integer> result =
    filter(numbers, (Integer i) -> i % 2 == 0);


List<Apple> result =
    filter(inventory, (Apple a) -> apple.getWeight() >
                                                   150);
```

A ‘real’ example

Before

```
inventory.sort(new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
});
```

After

```
inventory.sort((Apple a1, Apple a2) ->
                a1.getWeight().compareTo(a2.getWeight())
  );
```

A ‘real’ example

Before

```
button.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent event) {
        label.setText("Sent!!");
    }
});
```

After

```
button.setOnAction((ActionEvent event) ->
                                label.setText("Sent!!"));
```

Built-in Functional Interfaces

Functional interface	Lambda signature
Predicate<T>	T -> boolean
Consumer<T>	T -> void
Function<T, R>	T -> R
Supplier<T>	() -> T
UnaryOperator<T>	T -> T
BinaryOperator<T>	(T, T) -> T
BiFunction<T, U, R>	(T, U) -> R

- Have a look in `java.util.function.*`
- Primitive specialisations exist including `ToIntFunction`, `DoubleUnaryOperator` etc

Method References

Method References

- Method references let you reuse existing method definitions and pass them just like lambdas.
- “First-class” functions (yay!)

Before: `(Apple a) -> a.getWeight()`
After: **`Apple::getWeight`**

Before: `(String str, int i) -> str.substring(i)`
After: **`String::substring`**

Example

```
List<String> str =
Arrays.asList("a","b","A","B");
```

Before

```
str.sort((String s1, String s2) ->
                s1.compareToIgnoreCase(s2));
```

After

```
str.sort(String::compareToIgnoreCase);
```

A full example

Example: Classic

```
public class AppleComparator implements
Comparator<Apple> {
    public int compare(Apple a1, Apple a2) {
      return
       a1.getWeight().compareTo(a2.getWeight());
    }
}

inventory.sort(new AppleComparator());
```

Example: Anon classes

```
inventory.sort(new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2) {
        return a1.getWeight().compareTo(a2.getWeight());
    }
});
```

Example: Lambdas

```
inventory.sort(
    (Apple a1, Apple a2) ->
       a1.getWeight().compareTo(a2.getWeight()) );
```

Example: Library help

```
Comparator<Apple> byWeight =
    Comparator.comparing((Apple apple) -> apple.getWeight());

inventory.sort(byWeight);
```

Example: Library help

```
Comparator<Apple> byWeight =
    Comparator.comparing(Apple::getWeight);


inventory.sort(byWeight);
```

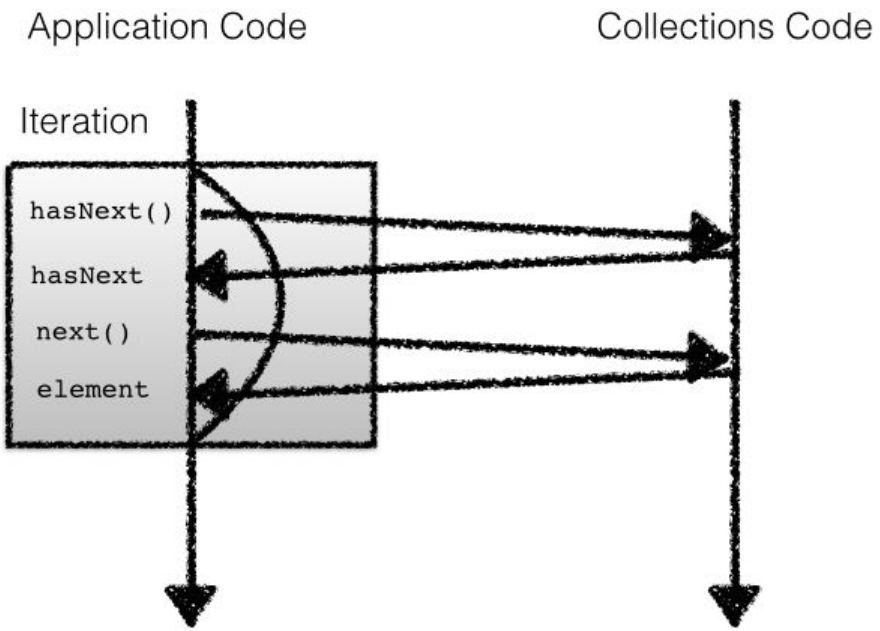
Example: Tidy up

```
inventory.sort(Comparator.comparing(Apple::getWeight));
```

Streams

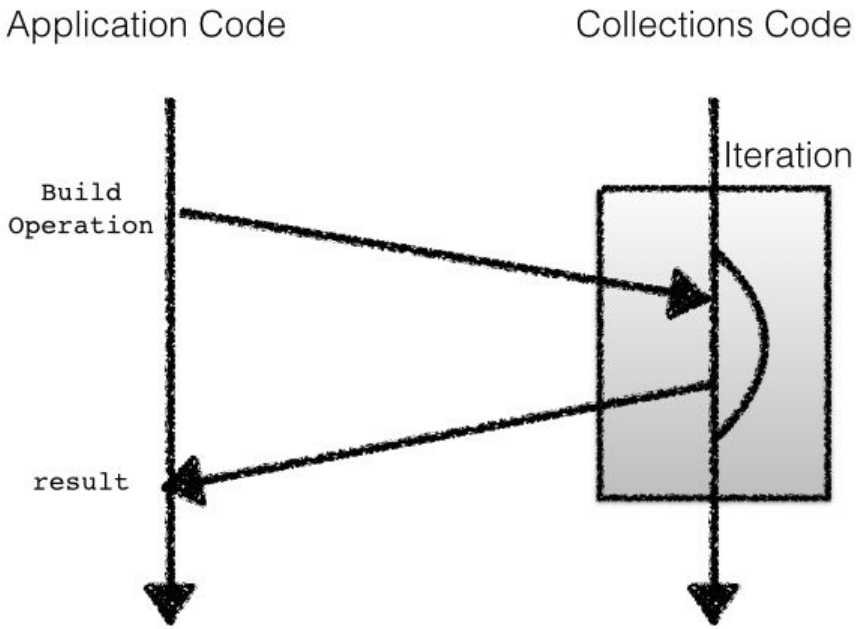
External Iteration

```
int count = 0;
for (Student student: students) {
    if (student.isFrom("Cambridge")) count++;
}
```



Internal Iteration

```
students.stream()
        .filter(student -> student.isFrom("Cambridge"))
        .count();
```

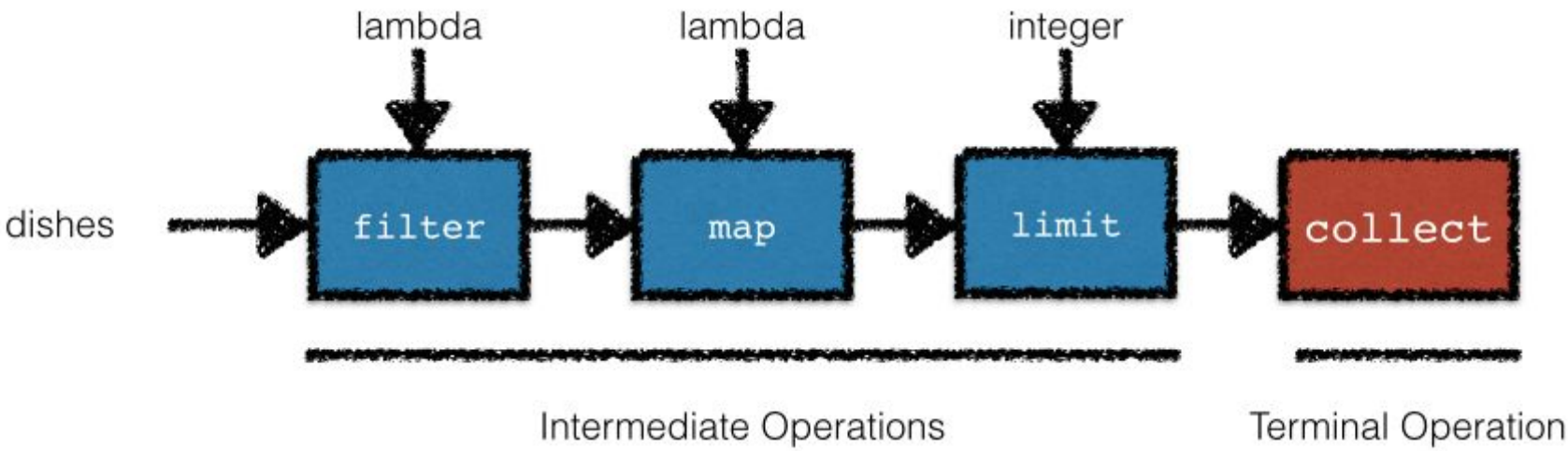


Streams

Informally: A fancy iterator with database-like operations

More formally: A sequence of elements from a source that supports aggregate operations

Allows Pipelining



Intermediate: returns a Stream and can be "connected"

Terminal: returns a non-Stream value (e.g. int, String,…)

Stream Operations

Operation	Type	Argument Type	Argument function descriptor	Result
filter	intermediate	Predicate<T>	T -> boolean	Stream<T>
distinct	intermediate			Stream<T>
skip	intermediate	long		Stream<T>
limit	intermediate	long		Stream<T>
map	intermediate	Function<T, R>	T -> R	Stream<R>
flatMap	intermediate	Function<T, Stream<R>>	T -> Stream<R>	Stream<R>
sorted	intermediate	Comparator<T>	(T, T) -> int	Stream<T>
anyMatch	terminal	Predicate<T>	T -> boolean	boolean
noneMatch	terminal	Predicate<T>	T -> boolean	boolean
allMatch	terminal	Predicate<T>	T -> boolean	boolean
findAny	terminal			Optional<T>
findFirst	terminal			Optional<T>
max/min	terminal	Comparator<T>	(T, T) -> int	Optional<T>
forEach	terminal	Consumer<T>	T -> void	void
collect	terminal	Collector<T, A, R>		R
reduce	terminal	BinaryOperator<T>	(T, T) -> T	Optional<T>
reduce	terminal	(T, BinaryOperator<T>)	(T , T) -> T	T
count	terminal			long

Before

```
List<Dish> lowCaloricDishes = new ArrayList<>();
for(Dish d: dishes) {
    if(d.getCalories() < 400) {
        lowCaloricDishes.add(d);
    }
}
Collections.sort(lowCaloricDishes,
   new Comparator<Dish>() {
    public int compare(Dish d1, Dish d2){
        return Integer.compare(d1.getCalories(), d2.getCalories());
    }
});
List<String> lowCaloricDishesName = new ArrayList<>();
for(Dish d: lowCaloricDishes) {
    lowCaloricDishesName.add(d.getName());
}
```

filter low calories

sorting by calories

extract names

After

```
List<String> lowCaloricDishesName =
                dishes.stream()
                      .filter(dish -> dish.getCalories() < 400)
                      .sorted(comparing(Dish::getCalories))
                      .map(Dish::getName)
                      .collect(toList());
```

Key advantages

Concise: makes readable code

Short-circuiting: Can stop as soon as the result is known, and not process the entire collection

Lazy evaluation: Only evaluate an expression when we need the result. I.e. only when you connect a terminal block do the intermediates get evaluated.

Aaaannnd…

…we're done!