Object Oriented Programming
Dr Andrew Rice

IA CST and NST (CS)
Michaelmas 2019/20

With thanks to Dr Robert Harle who designed this course and wrote the material.

The OOP Course

- So far you have studied some procedural programming in Java and functional programming in ML
- Here we take your procedural Java and build on it to get object-oriented Java
- You have practical work too
  - This course ***complements*** the practicals
  - Some material appears only here
  - Some material appears only in the practicals
  - Some material appears in both: deliberately*!

* Some material may be repeated unintentionally. If so I will claim it was deliberate.

Practical work is on chime.cl.cam.ac.uk

- Selection of exercises roughly mapped to lectures
- I want to write more so let me know where you see holes
- Attempt to get a bit closer to what you would do in industry
  - Git version control system
  - Automated testing

Drop-in sessions

- Thursday afternoons are drop-in help sessions
  - Intel Lab: 2-4pm
  - 21st Nov, 28th Nov, 16th Jan, 23rd Jan
- I will be there with some demonstrators
- Come talk to me about Java
- Bring your laptop if you want some help with your code

Other ways to get help

- Use the discussion forum on Moodle
  - Do not post your code or give answers: you'll spoil the practical work for others
  - If you need to include your code then please include a link to chime instead
  - Please answer your own question if you resolve it!
- Your supervisor
  - They can see your work on chime (if they ask me)
- Please do not email me directly – I get a lot of email

Assessment (1 of 2): Tripos exam

- There are two OOP questions in Paper 1
  - You will need to choose one of them
- Previous year's questions for this course are a good example of what you might be asked this year
- Only material that I lecture is examinable

Assessment (2 of 2): Take-home test

- 9am on 21st April – 9am on 23rd April 2020
- Pass/fail – worth 2 ticks
- I will aim for an exercise which will take about 4 hours (but there is big variance on this)
- Take-home test will be done through chime too
  - But no automated tests
- I'll provide a mock test for you to try

Outline

1.How to do a practical exercise
2.Types, Objects and Classes
3. Designing Classes
4. Pointers, References and Memory
5. Inheritance
6. Polymorphism
7. Lifecycle of an Object
8. Error Handling
9. Java Collections
10. Object Comparison
11. Design Patterns
12. Design Pattern (cont.)

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* by Gamma et al.
- My favourites
  - Effective Java by Joshua Bloch
  - Java Puzzlers by Joshua Bloch (this one is just for fun)

Books and Resources II

- Also check the course web page
  - Updated notes (with annotations)
  - Videos of the lectures (if I can make it work)
  - Links to practical work
  - Code from the lectures
  - Sample tripos questions
  - Suggested supervision work

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

- **And the Moodle site "Computer Science Paper 1 (1A)"**
- **Watch for course announcements**

Lecture 1:
How to do a practical exercise

Objectives

- To understand the workflow and tools to complete a practical exercise

---

We’d like to use your code for research

- Research into teaching and learning is important!
- We want your consent to use your code and share it with others
  - We will ‘anonymise’ it
- Consent is optional and it has no impact on your grades or teaching if you do not

Demo: Log into chime and opt-in/opt-out

We use git over SSH for version control

- Same setup as github and gitlab.developers.cam.ac.uk
- Generate an SSH key
- Put the **public** part of the key on chime

Demo: creating an SSH key and adding it to chime

Practical exercises are linked online

- Go to the course webpages to find links to the practical exercises
- Follow the link and start the task

Demo: starting a task

Software licensing and copyright

- Complicated area…
- The default is that if you write software you own the copyright and other people can’t copy it
- We add licenses to make it clear what people can and can’t do
- The initial code for the tasks is Apache 2 Licensed
- The system assumes your changes will be licensed the same...but they don’t have to be
- Apache 2 License lets you do almost anything
  - Except remove or change the license

Demo: licenses on your code

Using an IDE is recommended!

- I’ll use IntelliJ here but you can use whatever you like
- You only need the (free) ‘community edition’
- IntelliJ has built-in support for git but you can use the command line or other tools if you prefer
  - Sourcetree on Mac is really nice

Demo: cloning your task into a new project

Maven is a build system for Java

- In the pre-arrival course you built your code manually
- This doesn't scale well
- Use a build system!
- There are many build systems for Java
  - All of them have strengths and weaknesses
  - We will use Maven in this course

Demo: Maven pom file and build

Be careful about what you check in

- Imagine you are working in a team on a shared code base
- Other engineers don't want your IDE settings
- Or your temp files
- Or your class files
- Or personal information!!!
- We use .gitignore to tell git to ignore some files

IntelliJ can run tests and a debugger

Demo: solve the task, run the tests, debug something

Git can be very simple

- Your local repository contains all information
- Local workflow: **add** files, then **commit** them
- There's another copy on chime
  - You use this as a **remote**
  - It's default name is 'origin'
- Full workflow: **pull** updates from remote, **add** and **commit** files, **push** back to remote

Git can be complicated

- You can have as many remotes as you like
- You can have branches and merge changes and and and…
- Just remember to pull before you make any changes and push when you are done and you should avoid any complexity

Demo: git with IntelliJ

Chime can run acceptance tests for you

- These are designed to give you feedback on your solution and whether its right
- These are hard to write so please help me improve them
  - If your solution was wrong but passed the tests then let me know
  - And vice-versa

Demo: run tests on chime

You should be writing your own tests

- Some tasks will measure instruction coverage
- In this course we're interested in 'unit tests'
  - Test a single, small piece of functionality

Demo: running tests with coverage in IntelliJ, writing a test

---

Lecture 2:
Types, Objects and Classes

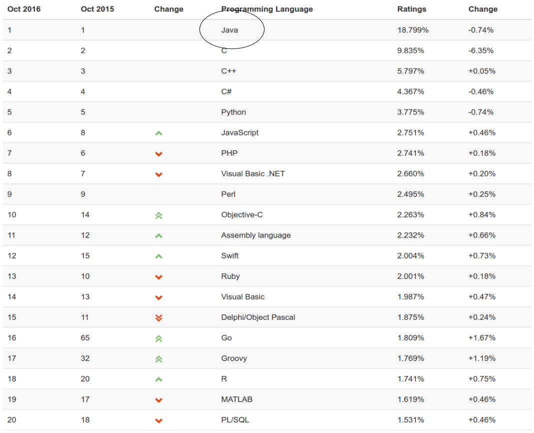
Objectives

- Remember procedural Java
- Understand function overloading
- Know the difference between a class and an object
- Know how to construct an object

Types of Languages

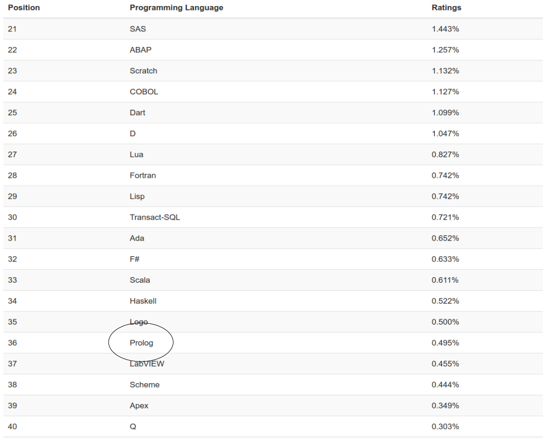
- Declarative - specify what to do, not how to do it. i.e.
  - E.g. HTML describes what should appear on a web page, and not how it should be drawn to the screen
  - E.g. SQL statements such as "select * from table" tell a program to get information from a database, but not how to do so
- Imperative – specify both what and how
  - E.g. "triple x" might be a declarative instruction that you want the variable x tripled in value. Imperatively we would have "x=x*3" or "x=x+x+x"

Top 20 Languages 2016



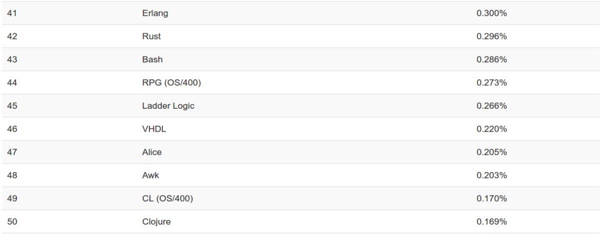
Oct 2016	Oct 2015	Change	Programming Language	Ratings	Change
1	1		Java	18.799%	-0.74%
2	2		C	9.835%	-6.35%
3	3		C++	5.797%	+0.05%
4	4		C#	4.367%	-0.46%
5	5		Python	3.775%	-0.74%
6	8	▲	JavaScript	2.751%	+0.46%
7	6	▼	PHP	2.741%	+0.18%
8	7	▼	Visual Basic .NET	2.660%	+0.20%
9	9		Perl	2.495%	+0.25%
10	14	▲▲	Objective-C	2.263%	+0.84%
11	12	▲	Assembly language	2.232%	+0.66%
12	15	▲	Swift	2.004%	+0.73%
13	10	▼	Ruby	2.001%	+0.18%
14	13	▼	Visual Basic	1.987%	+0.47%
15	11	▼▼	Delphi/Object Pascal	1.875%	+0.24%
16	65	▲▲	Go	1.809%	+1.67%
17	32	▲▲	Groovy	1.769%	+1.19%
18	20	▲	R	1.741%	+0.75%
19	17	▼	MATLAB	1.619%	+0.46%
20	18	▼	PL/SQL	1.531%	+0.46%

Top 20 Languages 2016 (Cont)



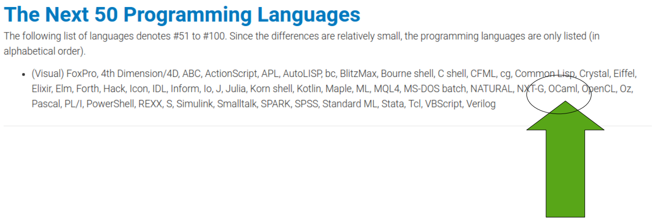
Position	Programming Language	Ratings
21	SAS	1.443%
22	ABAP	1.257%
23	Scratch	1.132%
24	COBOL	1.127%
25	Dart	1.099%
26	D	1.047%
27	Lua	0.827%
28	Fortran	0.742%
29	Lisp	0.742%
30	Transact-SQL	0.721%
31	Ada	0.652%
32	F#	0.633%
33	Scala	0.611%
34	Haskell	0.522%
35	Logo	0.500%
36	Prolog	0.495%
37	LabVIEW	0.455%
38	Scheme	0.444%
39	Apex	0.349%
40	Q	0.303%

Top 20 Languages 2016 (Cont Cont)



41	Erlang	0.300%
42	Rust	0.296%
43	Bash	0.286%
44	RPG (OS/400)	0.273%
45	Ladder Logic	0.266%
46	VHDL	0.220%
47	Alice	0.205%
48	Awk	0.203%
49	CL (OS/400)	0.170%
50	Clojure	0.169%

Top 20 Languages 2016 (Cont Cont Cont)



The Next 50 Programming Languages

The following list of languages denotes #51 to #100. Since the differences are relatively small, the programming languages are only listed (in alphabetical order).

- (Visual) FoxPro, 4th Dimension/4D, ABC, ActionScript, APL, AutoLISP, bc, BlitzMax, Bourne shell, C shell, CFML, cg, Common Lisp, Crystal, Eiffel, Elixir, Elm, Forth, Hack, Icon, IDL, Inform, Io, J, Julia, Korn shell, Kotlin, Maple, ML, MQL4, MS-DOS batch, NATURAL, NXT-G, OCaml, OpenCL, Oz, Pascal, PL/I, PowerShell, REXX, S, Simulink, Smalltalk, SPARK, SPSS, Standard ML, Stata, Tcl, VBScript, Verilog

ML as a Functional Language

- **Functional** languages are a subset of declarative languages
  - ML is a functional language
  - 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.

```
let rec factorial n =
  match n with
   | 0 -> 1
   | 1 -> 1
   | n -> n * (factorial (n – 1));
```

Function Side Effects

- Functions in imperative languages can use or alter larger system state → *procedures*

Maths: m(x,y) = xy

ML: fun m(x,y) = x*y;

Java:

```
int y = 7;
int m(int x) {
        y=y+1;
        return x*y;
}
```

Side effect

void Procedures

- A void procedure returns nothing:

```
int count=0;

void addToCount() {
  count=count+1;
}
```

Void is not quite the same as unit in ML

count+=1 count++ ++count

Control Flow: Looping

for(*initialisation; termination; increment* **)**

```
for (int i=0; i<8; i++) …

int j=0;  for(; j<8; j++) …

for(int k=7;k>=0; j--) …
```

Demo: printing the numbers from 1 to 10

while(*boolean_expression* **)**

```
int i=0;  while (i<8) { i++; …}

int j=7; while (j>=0) { j--; …}
```

Control Flow: Looping Examples

```
int arr[] = {1,2,3,4,5};

for (int i=0; i<arr.length;i++) {
        System.out.println(arr[i]);
}

int i=0;
while (i<arr.length) {
        System.out.println(arr[i]);
        i=i+1;
}
```

Control Flow: Branching I

- Branching statements interrupt the current control flow
- **return**
  - Used to return from a function at any point

```
boolean linearSearch(int[] xs, int v) {
   for (int i=0;i<xs.length; i++) {
     if (xs[i]==v) return true;
   }
   return false;
}
```

Control Flow: Branching II

- Branching statements interrupt the current control flow
- **break**
  - Used to jump out of a loop

```
boolean linearSearch(int[] xs, int v) {
   boolean found=false;
   for (int i=0;i<xs.length; i++) {
     if (xs[i]==v) {
          found=true;
          break;    // stop looping
     }
   }
   return found;
}
```

Control Flow: Branching III

- Branching statements interrupt the current control flow
- **continue**
  - Used to skip the current iteration in a loop

```
void printPositives(int[] xs) {

   for (int i=0;i<xs.length; i++) {
     if (xs[i]<0) continue;
     System.out.println(xs[i]);
   }
}
```

Immutable to Mutable Data

ML

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

ML is a language of expressions

Java is a language of statements and expressions

Java

```
int x=5;
x=7;

int x=9;
for(int i=0;i<10;i++) {
  System.out.println(i);
}
```

Evaluates to the value 7 with type int

Does not evaluate to a value and has no type

Demo: returning vs printing

Types and Variables

- Java and C++ have limited forms of type inference

```
var x = 512;
int y = 200;
int z = x+y;
```

- The high-level language has a series of *primitive* (built-in) types that we use to signify what's in the memory
  - The compiler then knows what to do with them
  - E.g. An "int" is a primitive type in C, C++, Java and many languages. In Java it is a 32-bit signed integer.
- A variable is a name used in the code to refer to a specific instance of a type
  - x,y,z are variables above
  - They are all of type int

E.g. 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
- 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

Widening
Vs
Narrowing

Demo: int → byte overflow

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

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) {...}
```

Custom Types

```
type 'a seq =
  | Nil
  |  Cons of 'a * (unit -> 'a seq);
```

```
public class Vector3D {
   float x;
   float y;
   float z;
}
```

State and Behaviour

```
type 'a seq =
  | Nil
  |  Cons of 'a * (unit -> 'a seq);

fun hd (Cons(x,_)) = x;
```

State and Behaviour

```
type 'a seq =
  | Nil
  |  Cons of 'a * (unit -> 'a seq);

fun hd (Cons(x,_)) = x;
```

```
public class Vector3D {
  float x;
  float y;
  float z;

  void add(float vx, float vy, float vz) {
    x=x+vx;
    y=y+vy;
    z=z+vz;
  }
}
```

STATE

BEHAVIOUR

Loose Terminology (again!)

State

Fields
Instance Variables
Properties
Variables
Members

Behaviour

Functions
Methods
Procedures

Classes, Instances and Objects

- Classes can be seen as templates for representing various ***concepts***
- We create ***instances*** of classes in a similar way. e.g.

```
MyCoolClass m = new  MyCoolClass();
MyCoolClass n = new  MyCoolClass();
```

makes two instances of class MyCoolClass.

- An instance of a class is called an **object**

Defining a Class

```
public class Vector3D {
  float x;
  float y;
  float z;

  void add(float vx, float vy, float vz) {
    x=x+vx;
    y=y+vy;
    z=z+vz;
  }
}
```

Constructors

```
MyObject m = new MyObject();
```

- You will have noticed that the RHS looks rather like a function call, and that's exactly what it is.
- It's a method that gets called when the object is constructed, and it goes by the name of a **constructor** (it's not rocket science). It maps to the datatype constructors you saw in ML.
- We use constructors to initialise the state of the class in a convenient way
  - A constructor has the same name as the class
  - A constructor has no return type

Constructors with Arguments

```
public class Vector3D {
  float x;
  float y;
  float z;

  Vector3D(float xi, float yi, float zi) {
    x=xi;
    y=yi;
    z=zi;
  }


  // ...
}
```

You can use 'this' to disambiguate names if needed: e.g. this.x = xi;

```
Vector3D v = new Vector3D(1.f,0.f,2.f);
```

Overloaded Constructors

```
public class Vector3D {
  float x;
  float y;
  float z;

  Vector3D(float xi, float yi, float zi) {
    x=xi;
    y=yi;
    z=zi;
  }

   Vector3D() {
    x=0.f;
    y=0.f;
    z=0.f;
  }

  // ...
}
```

```
Vector3D v = new Vector3D(1.f,0.f,2.f);
Vector3D v2 = new Vector3D();
```

Default Constructor

```
public class Vector3D {
  float x;
  float y;
  float z;
}

Vector3D v = new Vector3D();
```

If you don’t initialise a field it gets set to the ‘zero’ value for that type (don’t do this)

If you provide any constructor then the default will not be generated

- No constructor provided
- So blank one generated with no arguments

Lecture 3:
Designing Classes

Objectives

- Understand the static keyword
- Be able to identify what should be an object
- Start thinking about why OOP helps with modularity
- Know what encapsulation means
- Know what the access modifiers mean
- Be able to make an immutable object
- Understanding of simple generics

Class-Level Data and Functionality I

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

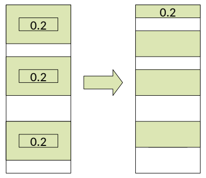
```
public class ShopItem {
  float mVATRate;
  static float sVATRate;
  ....
}
```

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

<u>Only</u> one of these created <u>ever</u>. Every ShopItem object references it.

static => associated with the class
instance => associated with the object

Class-Level Data and Functionality II



instance field
(one per object)

static field
(one per class)

- Shared between instances
- Space efficient

static fields are good for constants. otherwise use with care.

- Also static methods:

```
public class Whatever {
  public static void main(String[] args) {
    ...
  }
}
```

Why use Static Methods?

- Easier to debug (only depends on static state)
- Self documenting
- Groups related methods in a Class without requiring an object

```
public class Math {
  public float sqrt(float x) {…}
  public double sin(float x) {…}
  public double cos(float x) {…}
}

…
Math mathobject = new Math();
mathobject.sqrt(9.0);
...
```

vs

```
public class Math {
  public static float sqrt(float x) {…}
  public static float sin(float x) {…}
  public static float cos(float x) {…}
}

…
Math.sqrt(9.0);
...
```

What Not to Do

- Your ML has doubtless been one big file where you threw together all the functions and value declarations
- Lots of C programs look like this :-(
- We *could* emulate this in OOP by having one class and throwing everything into it
- We can do (much) better

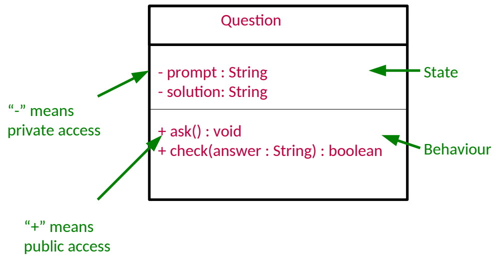
Identifying Classes

- 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 quiz program that asks questions

and checks the answers are correct”

UML: Representing a Class Graphically


Question

- prompt : String
- solution: String

+ ask() : void
+ check(answer : String) : boolean

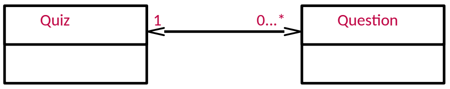
State

Behaviour

“-” means private access

“+” means public access

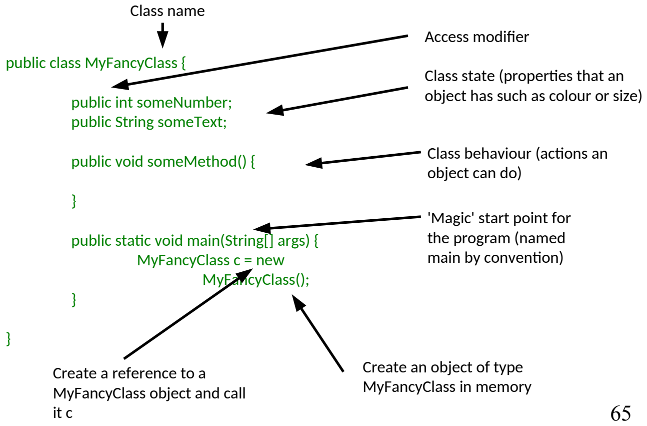
The has-a Association



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

Demo: implement quiz

Anatomy of an OOP Program (Java)


Class name

Access modifier

```
public class MyFancyClass {

        public int someNumber;
        public String someText;

        public void someMethod() {

        }

        public static void main(String[] args) {
                MyFancyClass c = new
                        MyFancyClass();
        }

}
```

Class state (properties that an object has such as colour or size)

Class behaviour (actions an object can do)

'Magic' start point for the program (named main by convention)

Create a reference to a MyFancyClass object and call it c

Create an object of type MyFancyClass in memory

OOP Concepts

- OOP provides the programmer with a number of important concepts:
  - Modularity
  - Code Re-Use
  - Encapsulation
  - Inheritance (lecture 5)
  - Polymorphism (lecture 6)
- Let's look at these more closely...

Modularity and Code Re-Use

- 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.

Encapsulation I

```
class Student {
  int age;
};

void main() {
  Student s = new Student();
  s.age = 21;

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

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

Encapsulation II

```
class Student {
  private int age;

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

  int getAge() {return age;}
}

void main() {
  Student s = new Student();
  s.setAge(21);
}
```

Encapsulation III

```
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);}
}
```

Encapsulation =
1) hiding internal state
2) bundling methods with state

Access Modifiers

	Everyone	Subclass	Same package (Java)	Same Class
private				X
package (Java)		Surprising!	X	X
protected		X	X	X
public	X	X	X	X

Immutability

- Everything in ML was immutable (ignoring the reference stuff). Immutability has a number of advantages:
  - Easier to construct, test and use
  - Can be used in concurrent contexts
  - Allows lazy instantiation
- We can use our access modifiers to create immutable classes
- If you mark a variable or field as ‘final’ then it can’t be changed after initalisation

Demo: NotImmutable

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
```

Fun fact: identity is the only function in ML with type ‘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>()
```

Creating Parameterised Types

- These just require a placeholder type

```
class Vector3D<T> {
  private T x;
  private T y;

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

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

Generics use type-erasure

```
class Vector3D<T> {
  private T x;
  private T y;

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

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

Vector3D<Integer> v =
      new Vector3D<>();
Integer x = v.getX();
v.setX(4);
```

after type checking this compiles to ------>

```
class Vector3D {
  private Object x;
  private Object y;

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

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

Vector3D v = new Vector3D();
Integer x = (Integer)v.getX();
v.setX((Object)4);
```

Lecture 4:
Pointers, References and Memory

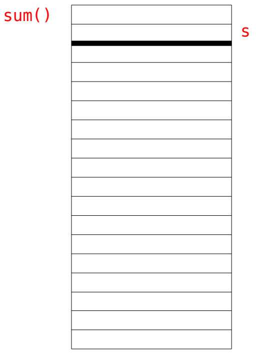
Objectives

- Know what a call-stack and a heap are
- Understand the difference between pointers and Java references

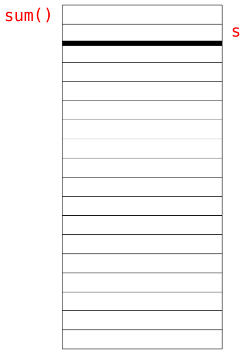
---

```
>> static int sum() {
     int s = sum(3);
     return s;
   }

   static int sum(int n) {
     if (n == 0) {
       return 0;
     }
     int m = sum(n - 1);
     int r = m + n;
     return r;
   }
```

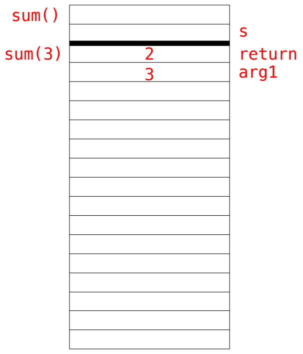
```
>> static int sum() {
     int s = sum(3);
     return s;
   }

   static int sum(int n) {
     if (n == 0) {
       return 0;
     }
     int m = sum(n - 1);
     int r = m + n;
     return r;
   }
```



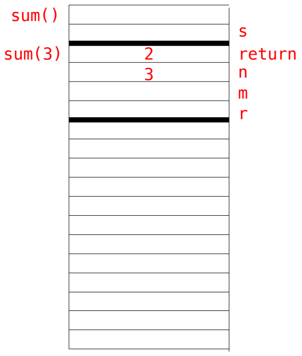
```
   static int sum() {
>>   int s = sum(3);
     return s;
   }

   static int sum(int n) {
     if (n == 0) {
       return 0;
     }
     int m = sum(n - 1);
     int r = m + n;
     return r;
   }
```



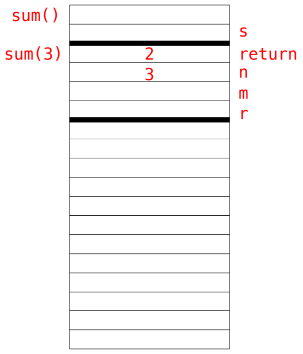
```
   static int sum() {
>>   int s = sum(3);
     return s;
   }

   static int sum(int n) {
     if (n == 0) {
       return 0;
     }
     int m = sum(n - 1);
     int r = m + n;
     return r;
   }
```



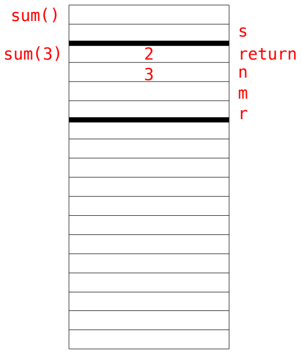
```
   static int sum() {
     int s = sum(3);
     return s;
   }

>> static int sum(int n) {
     if (n == 0) {
       return 0;
     }
     int m = sum(n - 1);
     int r = m + n;
     return r;
   }
```



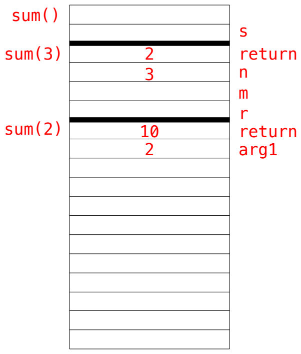
```
   static int sum() {
     int s = sum(3);
     return s;
   }

   static int sum(int n) {
>>   if (n == 0) {
       return 0;
     }
     int m = sum(n - 1);
     int r = m + n;
     return r;
   }
```



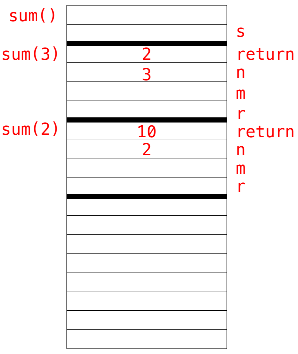
```
   static int sum() {
     int s = sum(3);
     return s;
   }

   static int sum(int n) {
     if (n == 0) {
       return 0;
     }
>>   int m = sum(n - 1);
     int r = m + n;
     return r;
   }
```



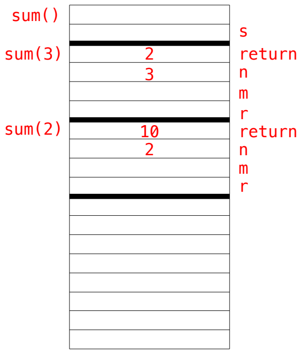
```
   static int sum() {
     int s = sum(3);
     return s;
   }

   static int sum(int n) {
     if (n == 0) {
       return 0;
     }
>>   int m = sum(n − 1);
     int r = m + n;
     return r;
   }
```



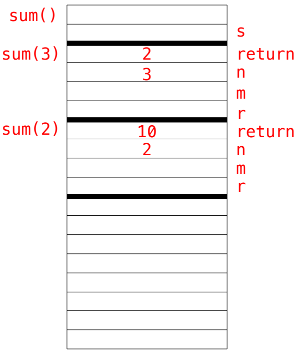
```
   static int sum() {
     int s = sum(3);
     return s;
   }

>> static int sum(int n) {
     if (n == 0) {
       return 0;
     }
     int m = sum(n − 1);
     int r = m + n;
     return r;
   }
```



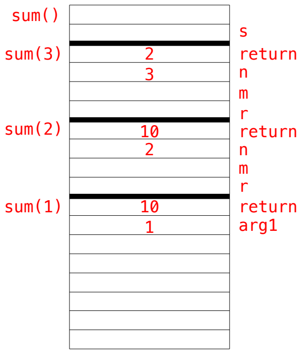
```
   static int sum() {
     int s = sum(3);
     return s;
   }

   static int sum(int n) {
>>   if (n == 0) {
       return 0;
     }
     int m = sum(n − 1);
     int r = m + n;
     return r;
   }
```



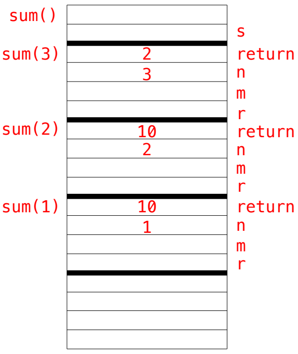
```
   static int sum() {
     int s = sum(3);
     return s;
   }

   static int sum(int n) {
     if (n == 0) {
       return 0;
     }
>>   int m = sum(n − 1);
     int r = m + n;
     return r;
   }
```



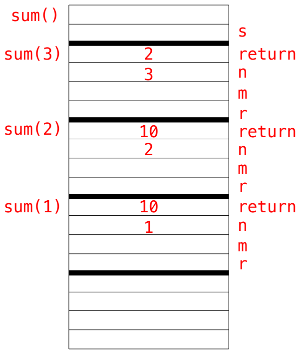
```
   static int sum() {
     int s = sum(3);
     return s;
   }

   static int sum(int n) {
     if (n == 0) {
       return 0;
     }
>>   int m = sum(n − 1);
     int r = m + n;
     return r;
   }
```



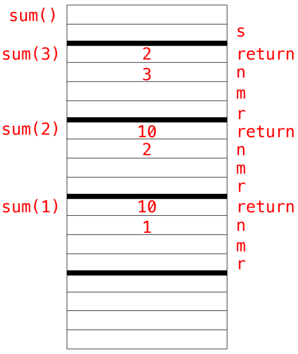
```
   static int sum() {
     int s = sum(3);
     return s;
   }

>> static int sum(int n) {
     if (n == 0) {
       return 0;
     }
     int m = sum(n − 1);
     int r = m + n;
     return r;
   }
```



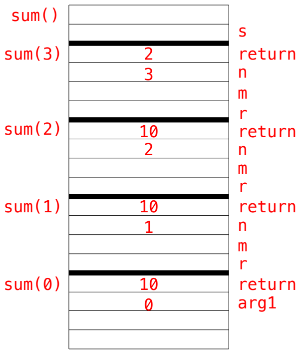
```
    static int sum() {
      int s = sum(3);
      return s;
    }

    static int sum(int n) {
>>    if (n == 0) {
        return 0;
      }
      int m = sum(n - 1);
      int r = m + n;
      return r;
    }
```



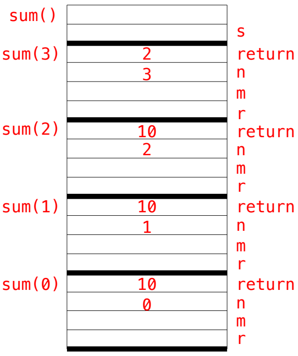
```
    static int sum() {
      int s = sum(3);
      return s;
    }

    static int sum(int n) {
      if (n == 0) {
        return 0;
      }
>>    int m = sum(n - 1);
      int r = m + n;
      return r;
    }
```



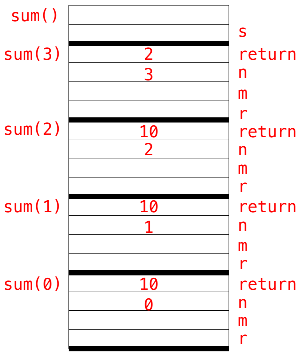
```
    static int sum() {
      int s = sum(3);
      return s;
    }

    static int sum(int n) {
      if (n == 0) {
        return 0;
      }
>>    int m = sum(n - 1);
      int r = m + n;
      return r;
    }
```



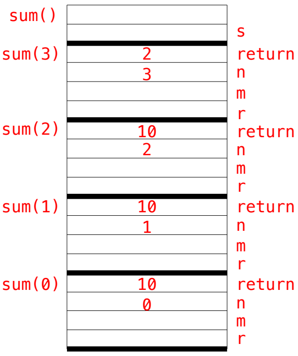
```
    static int sum() {
      int s = sum(3);
      return s;
    }

>>  static int sum(int n) {
      if (n == 0) {
        return 0;
      }
      int m = sum(n - 1);
      int r = m + n;
      return r;
    }
```



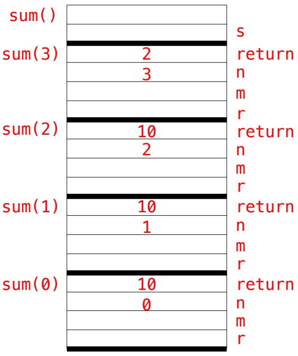
```
    static int sum() {
      int s = sum(3);
      return s;
    }

    static int sum(int n) {
>>    if (n == 0) {
        return 0;
      }
      int m = sum(n - 1);
      int r = m + n;
      return r;
    }
```



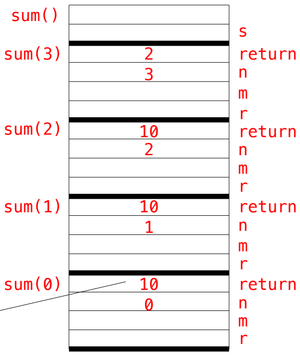
```
    static int sum() {
      int s = sum(3);
      return s;
    }

    static int sum(int n) {
      if (n == 0) {
>>      return 0;
      }
      int m = sum(n - 1);
      int r = m + n;
      return r;
    }
```



```
    1  static int sum() {
    2    int s = sum(3);
    3    return s;
    4  }
    5
    6  static int sum(int n) {
    7    if (n == 0) {
>>  8      return 0;
    9    }
   10    int m = sum(n - 1);
   11    int r = m + n;
   12    return r;
   13  }
```

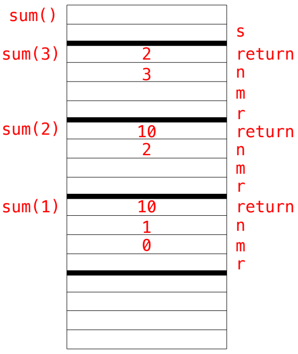


```
    1  static int sum() {
    2    int s = sum(3);
    3    return s;
    4  }
    5
    6  static int sum(int n) {
    7    if (n == 0) {
>>  8      return 0;
    9    }
   10    int m = sum(n - 1);
   11    int r = m + n;
   12    return r;
   13  }
```

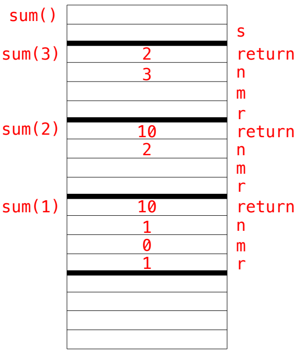


Return the value 0 and then execute instruction 10

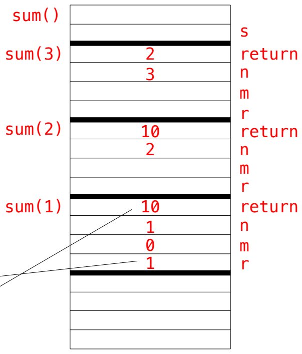
```
    1  static int sum() {
    2    int s = sum(3);
    3    return s;
    4  }
    5
    6  static int sum(int n) {
    7    if (n == 0) {
    8      return 0;
    9    }
>> 10    int m = sum(n - 1);
   11    int r = m + n;
   12    return r;
   13  }
```



```
    1  static int sum() {
    2    int s = sum(3);
    3    return s;
    4  }
    5
    6  static int sum(int n) {
    7    if (n == 0) {
    8      return 0;
    9    }
   10    int m = sum(n - 1);
>> 11    int r = m + n;
   12    return r;
   13  }
```

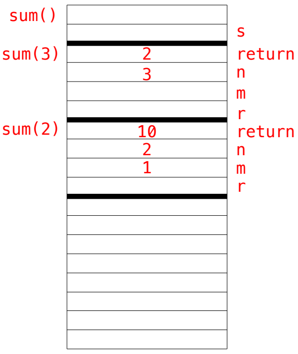


```
    1  static int sum() {
    2    int s = sum(3);
    3    return s;
    4  }
    5
    6  static int sum(int n) {
    7    if (n == 0) {
    8      return 0;
    9    }
   10    int m = sum(n - 1);
   11    int r = m + n;
>> 12    return r;
   13  }
```

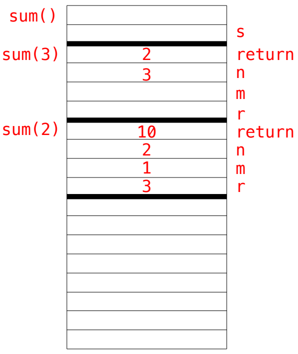


Return the value 1 and then execute instruction 10

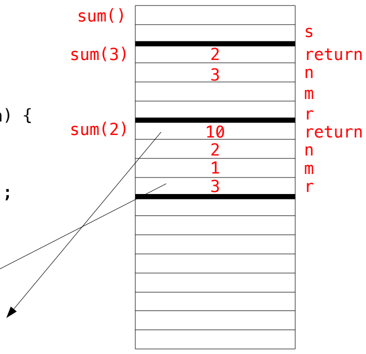
```
    1  static int sum() {
    2    int s = sum(3);
    3    return s;
    4  }
    5
    6  static int sum(int n) {
    7    if (n == 0) {
    8      return 0;
    9    }
>> 10    int m = sum(n - 1);
   11    int r = m + n;
   12    return r;
   13  }
```



```
    1  static int sum() {
    2    int s = sum(3);
    3    return s;
    4  }
    5
    6  static int sum(int n) {
    7    if (n == 0) {
    8      return 0;
    9    }
   10    int m = sum(n - 1);
>> 11    int r = m + n;
   12    return r;
   13  }
```

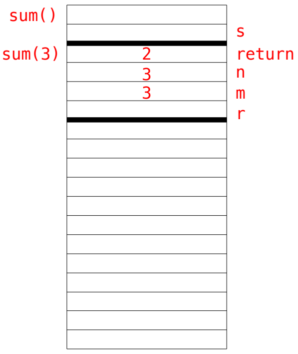


```
    1  static int sum() {
    2    int s = sum(3);
    3    return s;
    4  }
    5
    6  static int sum(int n) {
    7    if (n == 0) {
    8      return 0;
    9    }
   10    int m = sum(n - 1);
   11    int r = m + n;
>> 12    return r;
   13  }
```

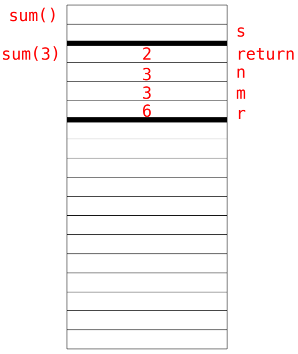


Return the value 3 and then execute instruction 10

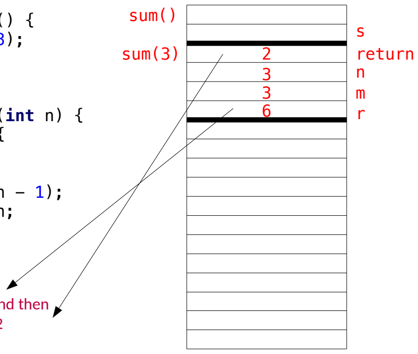
```
    1  static int sum() {
    2    int s = sum(3);
    3    return s;
    4  }
    5
    6  static int sum(int n) {
    7    if (n == 0) {
    8      return 0;
    9    }
>> 10    int m = sum(n - 1);
   11    int r = m + n;
   12    return r;
   13  }
```



```
    1  static int sum() {
    2    int s = sum(3);
    3    return s;
    4  }
    5
    6  static int sum(int n) {
    7    if (n == 0) {
    8      return 0;
    9    }
   10    int m = sum(n - 1);
>> 11    int r = m + n;
   12    return r;
   13  }
```

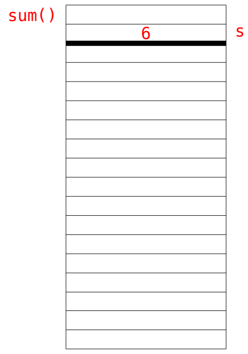


```
    1  static int sum() {
    2    int s = sum(3);
    3    return s;
    4  }
    5
    6  static int sum(int n) {
    7    if (n == 0) {
    8      return 0;
    9    }
   10    int m = sum(n - 1);
   11    int r = m + n;
>> 12    return r;
   13  }
```

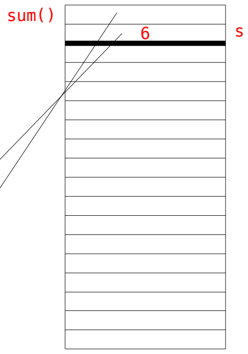


Return the value 6 and then execute instruction 2

```
    1  static int sum() {
>>  2    int s = sum(3);
    3    return s;
    4  }
    5
    6  static int sum(int n) {
    7    if (n == 0) {
    8      return 0;
    9    }
   10    int m = sum(n - 1);
   11    int r = m + n;
   12    return r;
   13  }
```



```
   static int sum() {
     int s = sum(3);
>>   return s;
   }

   static int sum(int n) {
     if (n == 0) {
       return 0;
     }
     int m = sum(n - 1);
     int r = m + n;
     return r;
   }
```



Return the value 6 and then execute whatever called us

Distinguishing References and Pointers

	Pointers	References in Java
Can be unassigned (null)	Yes	Yes
Can be assigned to established object	Yes	Yes
Can be assigned to an arbitrary chunk of memory	Yes	**No**
Can be tested for validity	**No**	Yes
Can perform arithmetic	Yes	**No**

References in Java

- Declaring unassigned

```
SomeClass ref = null;  // explicit

SomeClass ref2;  // implicit
```

- Defining/assigning

```
// Assign
SomeClass ref = new ClassRef();

// Reassign to alias something else
ref = new ClassRef();

// Reference the same thing as another reference
SomeClass ref2 = ref;
```

Lecture 5: Inheritance

Objectives

- Understand what pass-by-value means in Java
- Know the difference between code and type inheritance
- Can apply narrowing and widening to subtyping relations
- Appreciate how fields are inherited and shadowed
- Know how to override a method

Argument Passing

- **Pass-by-value**. Copy the value into a new one in the stack

```
void test(int x) {...}
int y=3;
test(y);

void test(Object o) {...}
Object p = new Object();
test(p);
```

The value passed here is the reference to the object.

When run the test method's argument o is copy of the reference p that points to the same object

Passing Procedure Arguments In Java

```
class Reference {

  public static void update(int i, int[] array) {
    i++;
    array[0]++;
  }

  public static void main(String[] args) {
    int test_i = 1;
    int[] test_array = {1};
    update(test_i, test_array);
    System.out.println(test_i);
    System.out.println(test_array[0]);
  }

}
```

the value here is an int

the value here is a reference to an int array

prints 1

prints 2

Demo: reference aliasing

Inheritance I

```
class Student {
  private int age;
  private String name;
  private int grade;
  ...
}

class Lecturer {
  private int age;
  private String name;
  private 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

Demo: expression evaluator

Inheritance II

```
class Person {
  protected int age;
  protected String name;
  ...
}

class Student extends Person {
  private int grade;
  ...
}

class Lecturer extends Person {
  private int salary;
  ...
}
```

- We create a *base class* (Person) and add a new notion: classes can *inherit* properties from it
  - Both state, functionality and type
- We say:
  - Person is the *superclass* of Lecturer and Student
  - Lecturer and Student *subclass* Person

'extends' in Java gives you both code- and type-inheritance

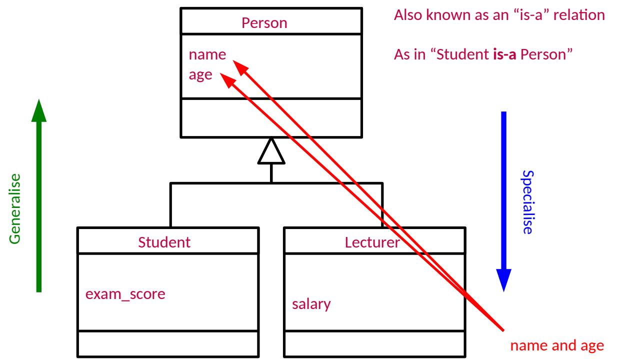
Note: Java is a **nominative** type language (rather than a **structurally** typed one)

If you mark a class 'final' then it can't be extended and 'final' methods can't be overridden

Liskov Substitution Principle

- If S is a subtype of T then objects of type T may be replaced with objects of type S
- Student is a subtype of Person so anywhere I can have a Person I can have a Student instead

Representing Inheritance Graphically



Also known as an "is-a" relation

As in "Student **is-a** Person"

name and age inherited if not private

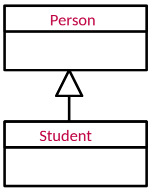
Casting

- Many languages support *type casting* 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 to any of the types above it in the inheritance tree...

Widening



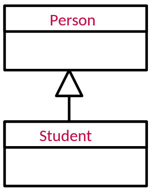
- Student is-a Person
- Hence we can use a Student object anywhere we want a Person object
- Can perform *widening* conversions (up the tree)

```
Student s = new Student()

Person p = s;
```

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

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

Implicit widening

Narrowing



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

```
Person p = new Person();

Student s = (Student) p;
```

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

OK because underlying object really is a Student

```
public void print(Person p) {
  Student s = (Student) p;
}

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

Fields and Inheritance

```
class Person {
   public String name;
   protected int age;
   private double height;
}

class Student extends Person {

  public void do_something() {
    name="Bob";
    age=70;
    height=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

Student inherits this but as a **private** variable and so cannot access it directly
This line doesn'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;
  }
}
```

'this' is a reference to the current object

'super' is a reference to the parent object

all classes extend Object (capital O)

if you write 'class A {}' you actually get 'class extends Object {}'

Object a = new A();   // substitution principle

Don't write code like this. No-one will understand you!

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...

Know the difference: overriding vs overloading

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

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

class Lecturer extends Person {
   public void dance(int duration) {...}
}
```

Person defines an original implementation of dance()

Student overrides the original

Lecturer overloads the inherited dance() method

Lecturer inherits the original implementation and jiggles

Abstract Methods

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

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

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

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

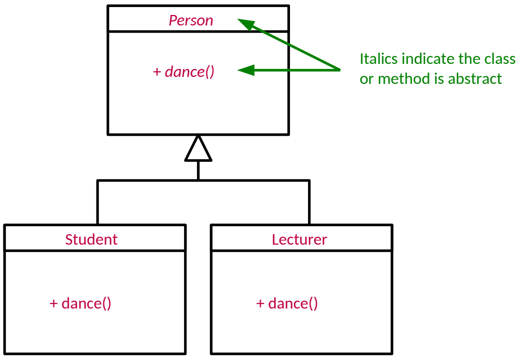
Abstract Classes

- 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!

Representing Abstract Classes



Italics indicate the class or method is abstract

Lecture 6:
Polymorphism and Multiple Inheritance

Objectives

- Dynamic and static polymorphism
- Problems that arise from multiple code inheritance
- Java interfaces provide multiple type inheritance

Polymorphic Methods

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

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

Demo: revisit expressions from last time

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

Polymorphism: values and variables can have more than one type

```
int eval(Expression e) {

}
```

can be Literal, Mult or Plus

Polymorphic Concepts I

- **Static** polymorphism
  - Decide at compile-time
  - Since we don't know what the true type of the object will be, we just run the method based on its static type

```
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

C++ can do this. Java cannot

Polymorphic Concepts II

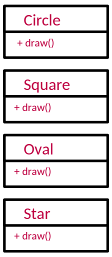
- **Dynamic** polymorphism
  - Run the method in the child
  - Must be done at run-time since that's when we know the child's type
  - Also known as ‘dynamic dispatch’

```
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

C++ can do this when you choose, Java does it always

The Canonical Example I



Circle
+ draw()

Square
+ draw()

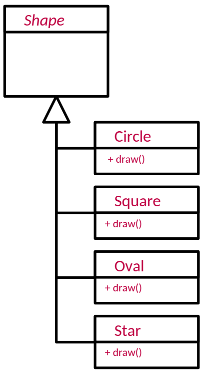
Oval
+ draw()

Star
+ draw()

- 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?

Demo

The Canonical Example II



Shape

Circle
+ draw()

Square
+ draw()

Oval
+ draw()

Star
+ draw()

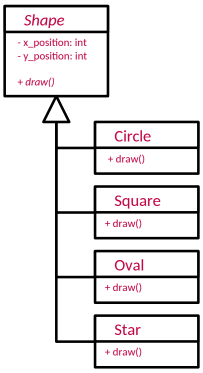
- **Option 2**
  - Keep a single list of Shape references
  - Figure out what each object really is, narrow the reference and then draw()

```
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?

Demo

The Canonical Example III



Shape
- x_position: int
- y_position: int
+ *draw()*

Circle
+ draw()

Square
+ draw()

Oval
+ draw()

Star
+ draw()

- **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?

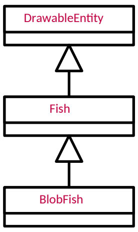
Demo

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...

Harder Problems

- Given a class Fish and a class DrawableEntity, how do we make a BlobFish class that is a drawable fish?



DrawableEntity

Fish

BlobFish

X Dependency between two independent concepts

DrawableEntity

0..1

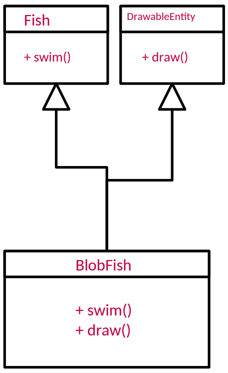
BlobFish

0..1

Fish

X Conceptually wrong

Multiple Inheritance

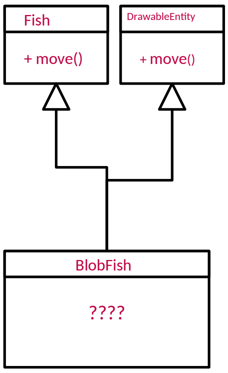


- If we multiple inherit, we capture the concept we want
- BlobFish inherits from both and is-a Fish and is-a DrawableEntity
- C++:

```
class Fish {...}
class DrawableEntity {...}

class BlobFish : public Fish,
                 public DrawableEntity {...}
```

- But...

Multiple Inheritance Problems



- What happens here? Which of the move() methods is inherited?
- Have to add some grammar to make it explicit
- C++:

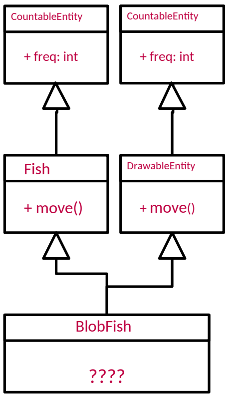
```
BlobFish *bf = new BlobFish();
bf->Fish::move();
bf->DrawableEntity::move();
```

- Yuk.

This is like field shadowing e.g.

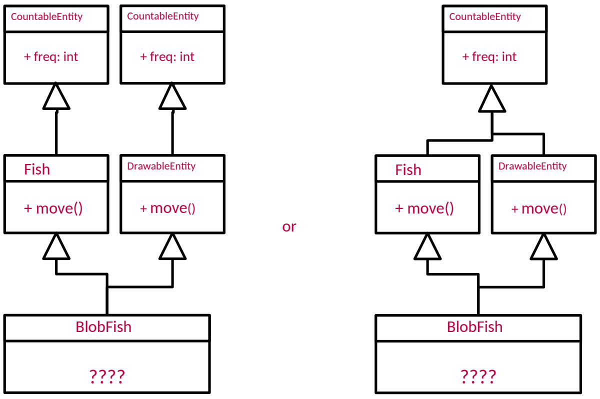
```
class A {              class B extends A {
  int x;                 int x;
}                      }
```

Multiple Inheritance Problems

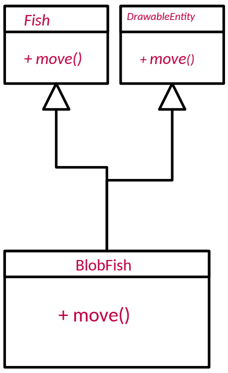


- What happens if Fish and DrawableEntity extend the same class?
- Do I get two copies?

The diamond problem



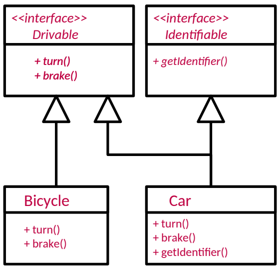
Fixing with Abstraction



- Actually, this problem goes away if one or more of the conflicting methods is abstract

Java's Take on it: Interfaces

- Classes can have at most **one** parent. Period.
- But special 'classes' that are totally abstract can do multiple inheritance – call these **interfaces**



```
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() {...}
}
```

adjective

This is type inheritance (not code inheritance)

Interfaces have a load of implicit modifiers

```
interface Foo {
   int x = 1;
   int y();
}
```

means

```
interface Foo {
    public static final int x = 1;
    public int y();
}
```

Interfaces can have default methods

```
interface Foo {
   int x = 1;
   int y();
   default int yPlusOne() {
      return y() + 1;
   }
}
```

- Allows you to add new functionality without breaking old code
- If you implement conflicting default methods you have to provide your own

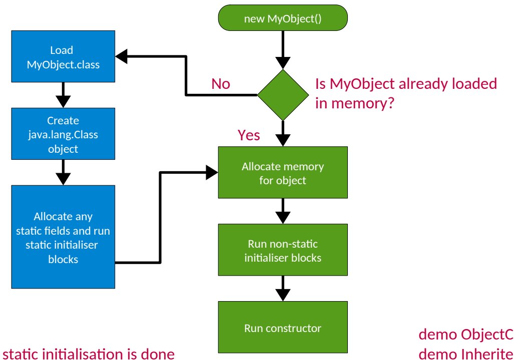
---

Lecture 7:
Lifecycle of an Object

Objectives

- Know the procedure for object initialisation
- Difference between destructors and finalisers
- RAII and TWR
- High level idea of a garbage collector

Creating Objects in Java



new MyObject()

Is MyObject already loaded in memory?

No

Load MyObject.class

Create java.lang.Class object

Allocate any static fields and run static initialiser blocks

Yes

Allocate memory for object

Run non-static initialiser blocks

Run constructor

static initialisation is done in textual order

demo ObjectConstruction
demo InheritedConstruction

Initialisation Example

```
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();
```

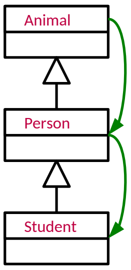
1. Blah loaded
2. sX created
3. sX set to 9
4. sX set to 3
5. Blah object allocated
6. mX set to 7
7. mX set to 5
8. Constructor runs (mX=1, sX=9)
9. b set to point to object

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

Constructor Chaining

- When you construct an object of a type with parent classes, we call the constructors of all of the parents in sequence

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



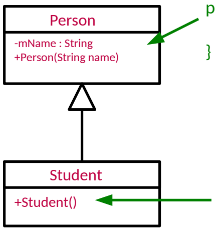
1. Call Animal()

2. Call Person()

3. Call Student()

Chaining without Default Constructors

- What if your classes have explicit constructors that take arguments? You need to explicitly chain
- Use **super** in Java:



```
public Person (String name) {
    mName=name;
}
```

```
public Student () {
    super("Bob");
}
```

Demo: NoDefaultConstructor

Deterministic Destruction

- Objects are created, used and (eventually) destroyed. Destruction is very language-specific
- Deterministic destuction is what you would expect
  - Objects are deleted at predictable times
  - Perhaps manually deleted (C++):

```
void UseRawPointer()
{
   MyClass *mc = new MyClass();
   // ...use mc...
   delete mc;
}
```

  - Or auto-deleted when out of scope (C++):

```
void UseSmartPointer()
{
   MyClass mc;
  // ...use mc...
}  // mc deleted here
```

In C++ this means create a new instance of MyClass on the stack using the default constructor

---

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) {

  FileReader f;

  // Use object here
  ...

}  // object destructor called here
```

This is called RAII = Resource Acquisition Is Initialisation

Non-Deterministic Destruction

- Deterministic destruction is easy to understand and seems simple enough. But it turns out we humans are rubbish of keeping track of what needs deleting when
- We either forget to delete (→ memory leak) or we delete multiple times (→ crash)
- **We can instead leave it to the system to figure out when to delete**
  - **"Garbage Collection"**
  - The system somehow figures out when to delete and does it for us
  - In reality it needs to be cautious and sure it can delete. This leads to us not being able to predict exactly when something will be deleted!!
- **This is the Java approach!!**

Demo: Finalizer

What about Destructors?

- Conventional destructors don't make sense in non-deterministic systems
  - When will they run?
  - Will they run at all??
- Instead we have finalisers: same concept but they only run when the system deletes the object (which may be never!)
- Java provides try-with-resources as an alternative to RAII

Demo: TryWithResources

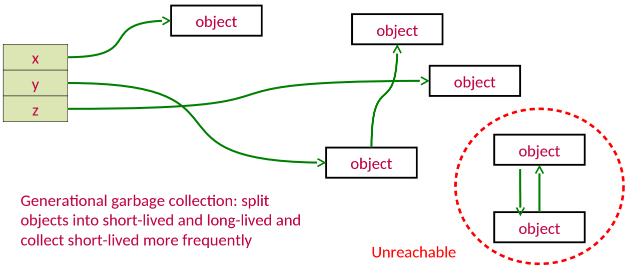
Garbage Collection

- So how exactly does garbage collection work? How can a system know that something can be deleted?
- The garbage collector is a separate process that is constantly monitoring your program, looking for things to delete
- Running the garbage collector is obviously not free. If your program creates a lot objects, you will soon notice the collector running
    - Can give noticeable pauses to your program!
    - But minimises memory leaks (it does not prevent them…)
- Keywords:
    - ‘Stop the world’ - pause the program when collecting garbage
    - ‘incremental’ - collect in multiple phases and let the program run in the gaps
    - ‘concurrent’ - no pauses in the program

Demo: Leak

Mark and sweep

- Start with a list of all references you can get to
- Follow all references recursively, marking each object
- Delete all objects that were not marked



Generational garbage collection: split objects into short-lived and long-lived and collect short-lived more frequently

Unreachable so deleted

Lecture 8:
Java Collections and Object Comparison

Objectives

- Understand boxing and unboxing
- A general idea about Java collections: Set, List, Queue and Map
- Fail-fast iterators

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)

lots of this in 1B Further Java

Boxing and unboxing

- Boxing: turn an int into an Integer
- Unboxing: turn an Integer into an int
- Java will do auto-boxing and unboxing

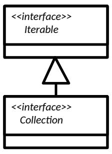
```
public void something(Integer I) {
    ...
}

int i = 4;
something(i);
```

auto-boxing

```
public void other(int i) {
    …
}

Integer i = null;
other(i);
```

auto-unboxing
(and a NPE)

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.

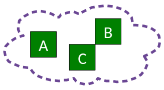
Sometimes an operation doesn't make sense – throw UnsupportedOperationError

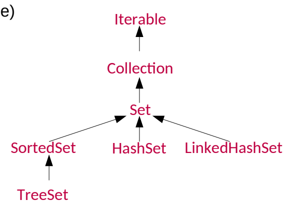
---

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)





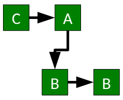
```
Set<Integer> ts = new TreeSet<>();
ts.add(15);
ts.add(12);
ts.contains(7);  // false
ts.contains(12); // true
ts.first(); // 12 (sorted)
```

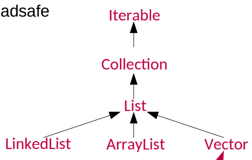
A form of type inference

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, as ArrayList but threadsafe





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

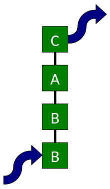
good default choice

legacy

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

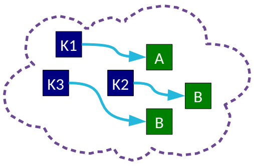


```
Queue<Double> ll = new LinkedList<>();
ll.offer(1.0);
ll.offer(0.5);
ll.poll(); // 1.0
ll.poll(); // 0.5
```

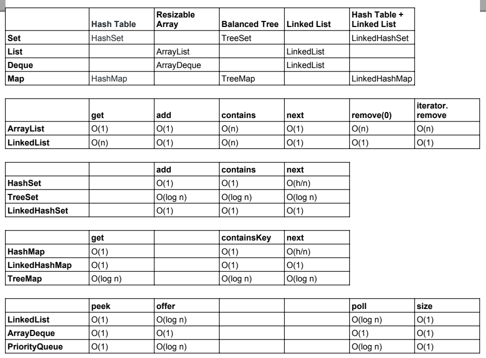
Maps

<<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 (see Algorithms)



```
Map<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
```



	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

	get	add	contains	next	remove(0)	iterator. remove
ArrayList	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
LinkedList	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)

		add	contains	next
HashSet		O(1)	O(1)	O(h/n)
TreeSet		O(log n)	O(log n)	O(log n)
LinkedHashSet		O(1)	O(1)	O(1)

	get		containsKey	next
HashMap	O(1)		O(1)	O(h/n)
LinkedHashMap	O(1)		O(1)	O(1)
TreeMap	O(log n)		O(log n)	O(log n)

	peek	offer			poll	size
LinkedList	O(1)	O(log n)			O(log n)	O(1)
ArrayDeque	O(1)	O(1)			O(1)	O(1)
PriorityQueue	O(1)	O(log n)			O(log n)	O(1)

Source: https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html

Source: Java Generics and Collections (pages: 188, 211, 222, 240)

Don't just memorise these – think about how the datastructure works

Specific return type and general argument

- Should your method take a Set, a SortedSet or a TreeSet?
- General rule of thumb:
  - use the most general type possible for parameters
  - use the most specific type possible for return values (without over committing your implementation)

Iteration

- for loop

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

- foreach loop (Java 5.0+)

```
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();
}
```

Demo: Fast fail behaviour

Comparing Objects

- You often want to impose orderings on your data collections
- For TreeSet and TreeMap this is automatic

```
TreeMap<String, Person> tm = ...
```

- For other collections you may need to explicitly sort

```
LinkedList<Person> list = new LinkedList<Person>();
//...
Collections.sort(list);
```

- For numeric types, no problem, but how do you tell Java how to sort Person objects, or any other custom class?

Lecture 9:
Error Handling Revisited

Objectives

- Comparing and Comparable
- Error handling approaches
- How to define your own exceptions
- Pros and cons of exceptions

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);
```

(p1==p2); ← False (references differ)

(p1!=p2); ← True (references differ)

(p1==p1); ← True

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));
   }
}
```

Learn the ‘equals’ contract

Demo: What’s wrong with equals

Java Quirk: hashCode()

- Object also gives classes hashCode()
- Code assumes that if equals(a,b) returns true, then a.hashCode() is the same as b.hashCode()
- So you should override hashCode() at the same time as equals()

Learn the ‘hashcode’ contract

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.
       }
    }
}
```

implementing Comparable defines a natural ordering for your class

ideally this should be consistent with equals i.e. x.compareTo(y) == 0 <=> x.equals(y)

must define a total order

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

Demo

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
```

delegate to the field's compareTo method

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
```

people argue about whether this is good or bad.

(Java won't let you do it)

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!!");
```

Go – returns a pair res, err
Haskell – Maybe type

- Problems:
  - Could 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

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 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

- 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(FileNoteFound fnf) {
  // handle file not found with FileReader
}
catch(IOException d) {
  // handle read() failed
}
```

finally

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

```
try {
  fr,read();
  fr.close();
}
catch(IOException ioe) {
  // read() failed but we must still close the FileReader
  fr.close();
}
```

finally II

- The finally block is added and will *always* run (after any handler)

```
try {
  fr.read();
}
catch(IOException ioe) {
  // read() failed
}
finally {
  fr.close();
}
```

Remember try-with-resources

Creating Exceptions

- Just extend Exception (or RuntimeException if you need it to be unchecked). 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);
   }
}
```

If your exception is caused by another then chain them - demo

- 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)

Keyword: exception chaining

Exception 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
}
catch(MathException me) {
 // handle any MathException or DivByZero
}
```

Checked vs Unchecked Exceptions

- Checked: must be handled or passed up.
  - Used for recoverable errors
  - Java requires you to declare checked exceptions that your method throws
  - Java requires you to catch the exception when you call the function

```
double somefunc() throws SomeException {}
```

- Unchecked: not expected to be handled. Used for programming errors
  - Extends RuntimeException
  - Good example is NullPointerException

No acceptance tests for take-home test

- Get in the habit of writing good tests
- There will be no acceptance tests for the take-home test – you have to get it right on your own!

Evil I: Exceptions for Flow Control

- At some level, throwing an exception is like a GOTO
- Tempting to exploit this

```
try {
  for (int i=0; ; i++) {
    System.out.println(myarray[i]);
  }
}
catch (ArrayOutOfBoundsException ae) {
  // This is expected
}
```

- This is not good. Exceptions are for exceptional circumstances only
  - Harder to read
  - May prevent optimisations

Evil II: Blank Handlers

- Checked exceptions must be handled
- Constantly having to use try...catch blocks to do this can be annoying and the temptation is to just gaffer-tape it for now

```
try {
  FileReader fr = new FileReader(filename);
}
catch (FileNotFound fnf) {
}
```

If it can't happen then throw
a chained RuntimeException

- ...but we never remember to fix it and we could easily be missing serious errors that manifest as bugs later on that are extremely hard to track down

Advantages of Exceptions

- Advantages:
  - Class name can be descriptive (no need to look up error codes)
  - Doesn't interrupt the natural flow of the code by requiring constant tests
  - The exception object itself can contain state that gives lots of detail on the error that caused the exception
  - Can't be ignored, only handled
- Disadvantages:
  - Surprising control flow – exceptions can be thrown from anywhere
  - Lends itself to single threads of execution
  - Unrolls control flow, doesn't unroll state changes

Lecture 10:
Copying Objects

Objectives

- Substitutability: covariance and contravariance
- Inner classes
- Lambda!
- Functional interfaces

Remember the substitution principle?

- If A extends B then I should be able to use B everywhere I expect an A

```
class A {                      class B extends A {

   Polygon getShape() {           Polygon getShape() {
     return new Polygon(…);         return …
   }                              }

}                              }
```

```
void process(A o) {
   drawShape(o.getShape());
}
process(new B());
```

Covariant return types are substitutable

- Overriding methods are covariant in their return types

```
class A {                      class B extends A {

   Polygon getShape() {           Triangle getShape() {
     return new Polygon(…);         return …
   }                              }

}                              }
```

```
void process(A o) {
   drawShape(o.getShape());
}
process(new B());
```

o.getShape() returns a Triangle but Triangle is a subtype of Polygon and so by substitutability we can pass it to drawShape

Contravariant parameters also substitute

- Overriding methods can be contravariant in their parameters

```
class A {                      class B extends A {

   void setShape(Triangle o) {    void setShape(Polygon o) {
     …                              …
   }                              }

}                              }
```

o.setShape() wants a Polygon and by substitutability its ok to pass it a Triangle

```
void process(A o) {
   o.setShape(new Triangle());
}
process(new B());
```

You can't actually do this in Java! The two setShapes are overloads not overrides

Java arrays are covariant

- If B is a subtype of A then B[] is a subtype of A[]

```
String[] s = new String[] { "v1", "v2" };

Object[] t = s;

Object v = t[0];

t[1] = new Integer(4);
```

Compiles – arrays are covariant

Works – t[0] is actually a String but we can assign that to Object

Fails (at runtime) – t[] is actually an array of Strings, you can't put an Integer in it

Imagine if Arrays were a generic class

```
class Array<Object> {

  // Object x = array[i]
  Object get(int index) {
    …
  }

  // array[i] = value
  void set(int index,
           Object value) {
    …
  }
}
```

Covariant return type – all is good!

Covariant parameter type – bad news

```
class Array<String> {

  // String x = array[i]
  String get(int index) {
  }

  // array[i] = value
  void set(int index,
           String value) {
  }
}
```

Generics in Java are not covariant

- if B is a subtype of A then T<B> is not a subtype of T<A>

```
List<String> s = List.of(“v1”, “v2”);

List<Object> t = s;

Object v = t.get(0);

t.set(1,new Integer(4));
```

Does not compile

Would be safe – we can assign String to Object

Is not safe

Wildcards let us capture this

- if B is a subtype of A then T<B> is a subtype of T<? extends A>

```
List<String> s = List.of(“v1”, “v2”);

List<? extends Object> t = s;

Object v = t.get(0);

t.set(1,new Integer(4));
```

Compiles

Works: ‘? extends Object’ is assignable to Object

Does not compile – the compiler knows it needs something that extends object but it doesn’t know what it is!

Inner classes

```
class Outer {

  private static void f();
  private int x = 4;

  static class StaticInner {

     void g() {
        f();
        new Outer().x = 3;
     }
  }

  class InstanceInner {
     int g() {
        return x + 1;
     }
  }

}
```

Inner classes may not have static members

Static inner classes are a member of the outer class and so can access private members

Instance inner classes are a member of the outer object and so can access outer instance variables:

Outer o = new Outer();
InstanceInner i = o.new InstanceInner()

Method-local classes

```
class Outer {

  int y = 6;

  void f() {
     int x = 5;
     class Foo {
        int g() {
          return x + y + 1;
        }
     }
     Foo foo = new Foo();

  }

}
```

Method-local classes in instance methods can access instance variables of the class

Method-local classes can access local variables (and so are never static classes).

Anonymous inner classes

```
class Outer {

  int y = 6;

  Object f() {
     int x = 5;
     Object o = new Object() {
       public String toString() {
         return String.valueOf(x+y+1);
       }
     };
     return o;
  }
}
```

x here is ‘effectively final’ - compile error if you try to change it

o is a new class. It extends Object but it has no name. It can access all local and instance variables.

Note: here we return o to the caller and it can be used anywhere in the program even though it refers to y and x.

Lambda

```
Consumer<String> c1 = s -> System.out.println(s);
c1.accept(“hello”);


BiFunction<Integer,Integer,Boolean> c2 = (i,j) -> i+j > 5;
boolean a = c2.apply(3,1);

Predicate<Integer> b4 = v -> {
   if (v > 0) {
     return isPrime(v);
   }
   else {
     return isPrime(v*v);
   }
}
boolean a = b4.test(43431);
```

expression lambda

statement lambda

Need a Functional Interface to use them

- A functional interface has only one method in it
- (this is so the compiler knows which one to map the lambda on to)
- That’s it

Lecture 11/12:
Design patterns

Objectives

- Simple use of streams
- What is a design pattern
- Open-closed principle
- Some example patterns

Streams

- Collections can be made into streams (sequences)
- These can be **filter**ed or **map**ped!

```
List<Integer> list = ...

list.stream().map(x->x+10).collect(Collectors.toList());

list.stream().filter(x->x>5).collect(Collectors.toList());
```

create stream

element-wise operations

aggregation

Demo:streams

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
- It’s not a competition to see how many you can use in a project!

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

---

Decorator

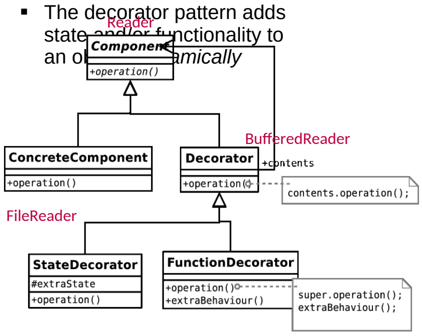
Abstract problem: How can we add state or methods at runtime?

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

Demo: Readers

Decorator in General

- The decorator pattern adds state and/or functionality to an ob *amically*



Singleton

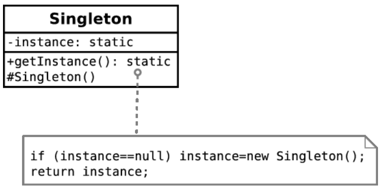
Abstract problem: How can we ensure only one instance of an object is created by developers using our code?

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 one connection at a time.

demo: SingletonConnection

Singleton in General

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



Demo: FanSpeed

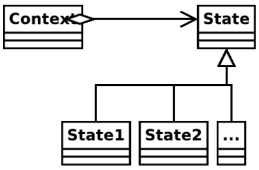
State

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

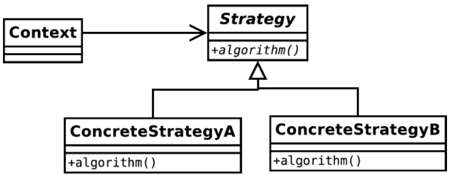
Abstract problem: How can we select an algorithm implementation at runtime?

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

Demo: ComparatorStrategy

Strategy in General

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



Composite

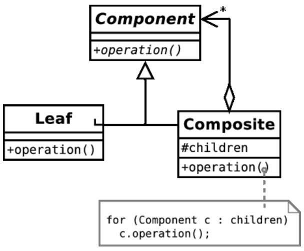
Abstract problem: How can we treat a group of objects as a single object?

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

Demo: DVDs

Composite in General

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



Observer

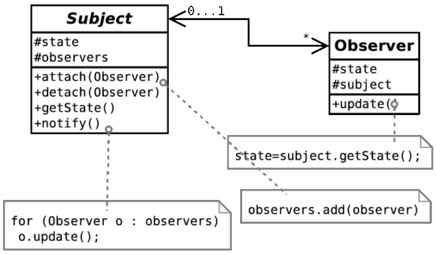
Abstract problem: When an object changes state, how can any interested parties know?

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

Demo: ActionListener

Observer in General

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



```
state=subject.getState();
```

```
observers.add(observer)
```

```
for (Observer o : observers)
 o.update();
```

End of course

- Don’t forget to keep practising with the practical exercises
- You will receive email about the take-home test organisation closer to the time
- Thanks for listening!