Object Oriented Programming
Dr Andrew Rice

IA CST and NST (CS)

Michaelmas 2018/19

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

annotations by me!

these are still examinable

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 ticks in Java
  - This course ***complements*** the practicals
  - Some material appears only here
  - Some material appears only in the practicals
  - Some material appears in both: deliberately*!

Ticks released on friday, due the following friday

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

Outline

1. Types, Objects and Classes
2. Designing Classes
3. Pointers, References and Memory
4. Inheritance
5. Polymorphism
6. Lifecycle of an Object
7. Error Handling
8. Copying Objects
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

+ Effective Java
Joshua Bloch

- Design Patterns
  - *Design Patterns* by Gamma et al.
  - Lots of good resources on the web

Books and Resources II

- Also check the course web page
  - Updated notes (with annotations where possible)
  - Code from the lectures
  - Sample tripos questions

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

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

Objectives

1) Remember procedural Java!
2) Understand function overloading
3) Know the difference between a class and an object
4) Know how to construct an object
5) Understand the static keyword

Lecture 1:

Types, Objects and Classes

Types of Languages

- Declarative - specify <u>what</u> to do, not <u>how</u> 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 <u>both</u> 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, Stand[illegible]L, 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.

```
fun fact 0 = 1
  | fact n = n * fact (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 m(int x, int y) = x*y;~~

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

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

val x = ref 5;
x := 7;

has type unit

Java

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

int x=9;
```

has type int
and value 7

demo: returning
vs printing

Types and Variables

- ~~Most imperative languages don't have type inference~~

Java10

var x = 512;

Java and C++ have limited forms of type inference

```
int 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. It's usually 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

in Java its 32-bit

in C/C++ it might be

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 of
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

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

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

State and Behaviour

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

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

State and Behaviour

```
datatype '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

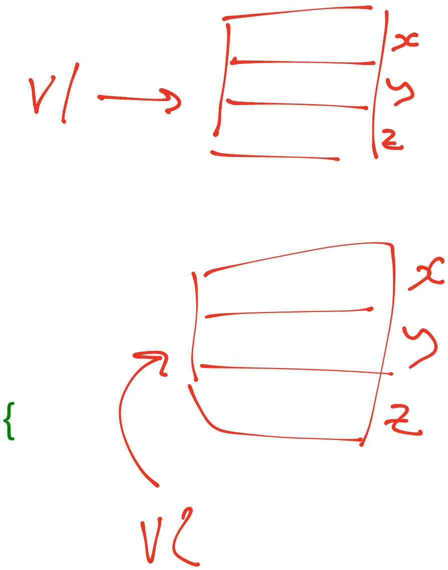
- Classes can be seen as templates for representing various ***concepts***
- We create ***instances*** of classes in a similar way. e.g.

```
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 initialize a field it gets set to the 'zero' value for that type. (don't do this)

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

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;
  ....
}
```

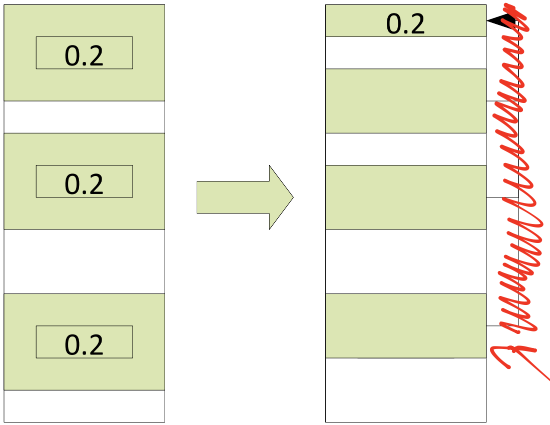
One of these created <u>every</u> time a new ShopItem is instantiated. Nothing keeps them all in sync.

<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

static field

object instance

- ~~Auto synchronised~~ shared across 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
- The compiler can produce more efficient code since no specific object is involved → do not worry about this in this course

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

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

vs

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

```
…
Math.sqrt(9.0);
…
```

Objectives: 1) understand the static keyword

2) what should be an object?

more on this later in the course

3) Why does one help with modularity?

4) What does encapsulation mean?

5) What do the different access modifiers mean?

Lecture 2:

Designing Classes

6) How to make an immutable object and why is this good?

7) A brief mention of generics

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

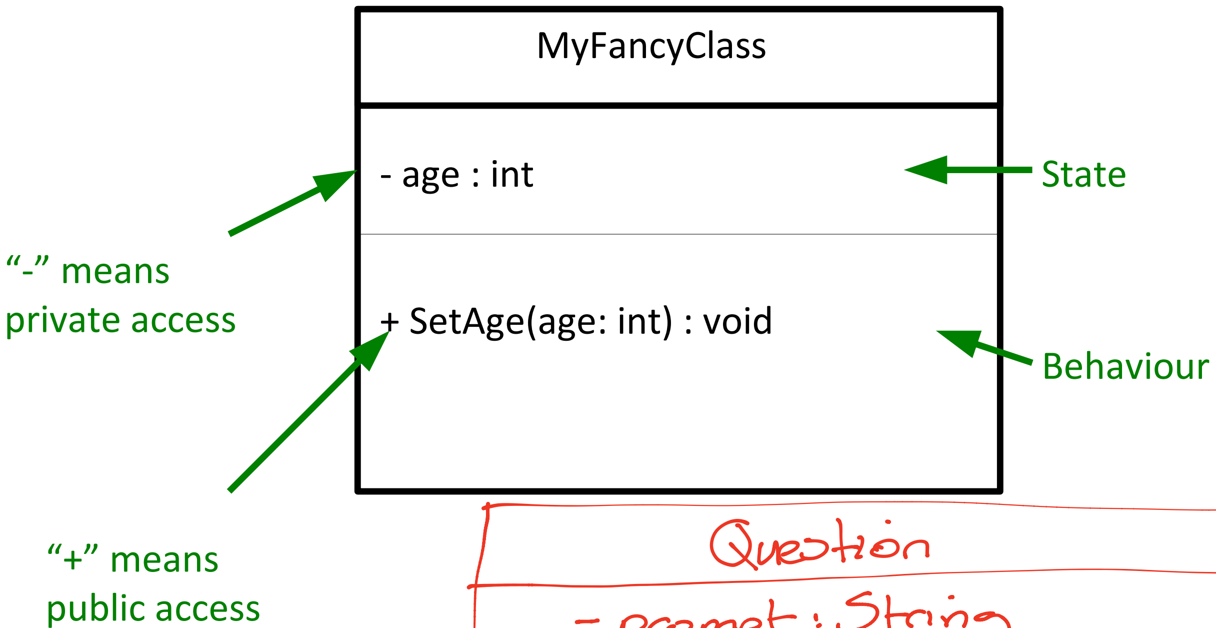
Identifying Classes

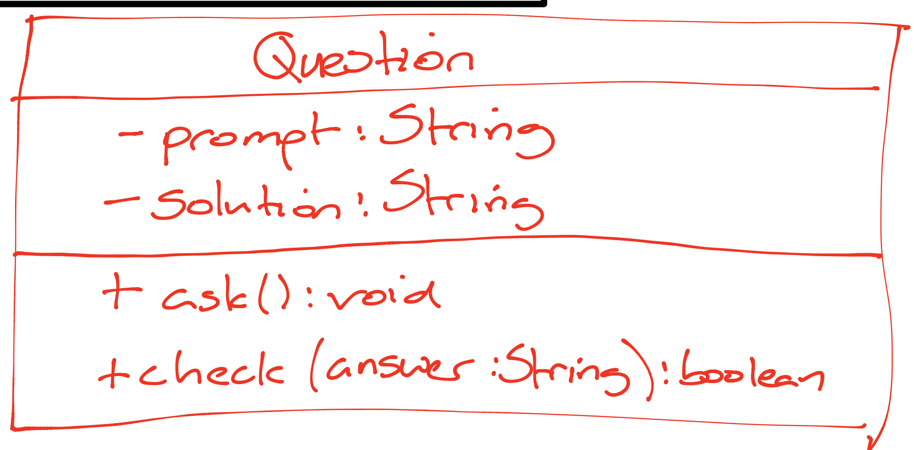
- We want our class to be a grouping of conceptually-related state and behaviour
- One popular way to group is using grammar
  - Noun → Object
  - Verb → Method

"A <u>simulation</u> of the <u>Earth</u>'s orbit around the <u>Sun</u>"

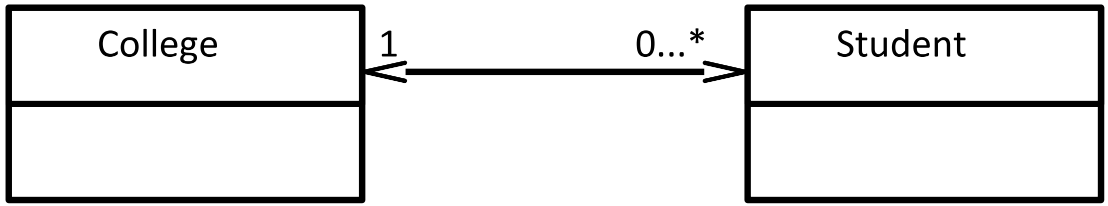
a quiz program that asks questions and checks the answers are correct

UML: Representing a Class Graphically





The has-a Association



- Arrow going left to right says "a College has zero or more students"

Quiz 0..* Question

- 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 to primitive types.

Demo: implement Quiz

Anatomy of an OOP Program (Java)

Class name

```
public class MyFancyClass {

        public int someNumber;
        public String someText;

        public void someMethod() {

        }

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

}
```

Access modifier

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

Anatomy of an OOP Program (C++)

Class name

Access modifier

I'll come back to this later if there is time

Class state

Class behaviour

don't do this!

```
class MyFancyClass {

public:
        int someNumber;
        public String someText;

        void someMethod() {

        }

};

void main(int argc, char **argv) {
        MyFancyClass c;

        MyFancyClass *cp = new MyFancyClass()

}
```

'Magic' start point for the program

Create an object of type MyFancyClass and call it cc

Create a pointer to a MyFancyClass object and call it cp

Create an object of type MyFancyClass and return a reference to it

OOP Concepts

- OOP provides the programmer with a number of important concepts:
  - Modularity
  - Code Re-Use
  - Encapsulation
  - Inheritance — in Lecture 4
  - Polymorphism - in Lecture 5
- 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)			X	X
protected		X	X	X
public	X	X	X	X

I find this one 'surprising'

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

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 $\alpha \to \alpha$

- 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;}
}
```

Java implements generics using something called type erasure

→ just remember this for now and I will explain later.

Objectives! what is a call stack & a heap?
how is it used.
difference between pointers and references
argument passing styles

Lecture 3:
Pointers, References and Memory

Memory and Pointers

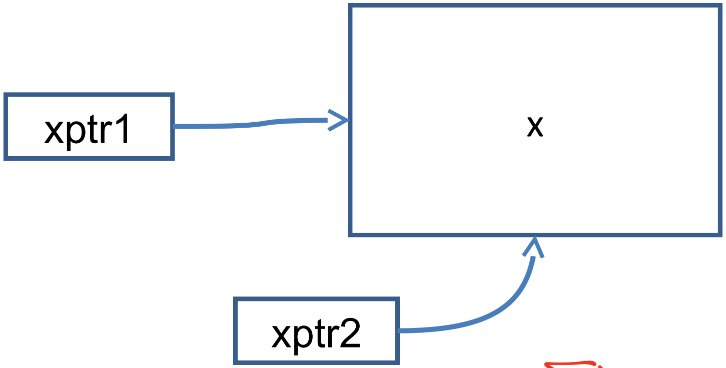
- In reality the compiler stores a mapping from variable name to a specific memory address, along with the type so it knows how to interpret the memory (e.g. *“x is an int so it spans 4 bytes starting at memory address 43526”*).
- Lower level languages often let us work with memory addresses directly. Variables that store memory addresses are called **pointers** or sometimes **references**
- Manipulating memory directly allows us to write fast, efficient code, but also exposes us to bigger risks
  - Get it wrong and the program 'crashes' .

(Switch to other handout)

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

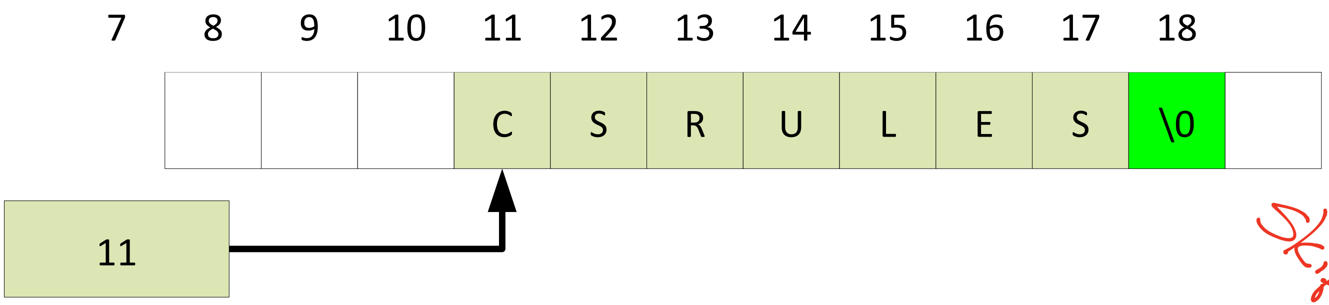
```
int x = 72;
int *xptr1 = &x;
int *xptr2 = xptr1;
```



Skipped

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

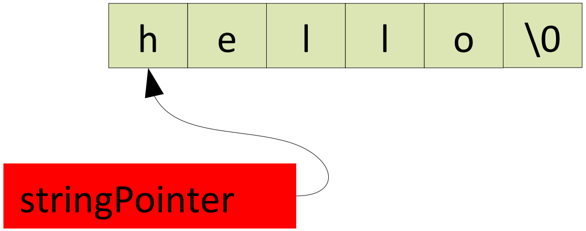


Skipped

- 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);
```



Skipped

References

- A reference is an **alias** for another thing (object/array/etc)
- When you use it, you are 'redirected' somehow to the underlying thing
- Properties:
  - Either assigned or unassigned
  - If assigned, it is valid
  - You can easily check if assigned

Skipped

Implementing References

- A sane reference implementation in an imperative language is going to use pointers
- So each reference is the same as a pointer except that the compiler restricts operations that would violate the properties of references
- For this course, thinking of a reference as a restricted pointer is fine

Skipped.

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 == null	**No**	Yes
Can perform arithmetic	Yes	**No**

Languages and References

- Pointers are useful but dangerous
- C, C++: pointers *and* references
- Java: references *only*
- ML: references *only*

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;
```

Arrays

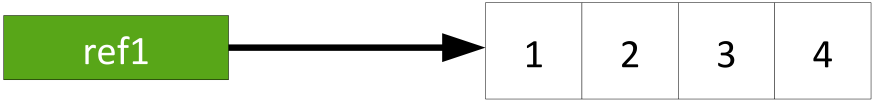
byte[] arraydemo1 = new byte[6];
byte   arraydemo2[] = new byte[6];

Skipped

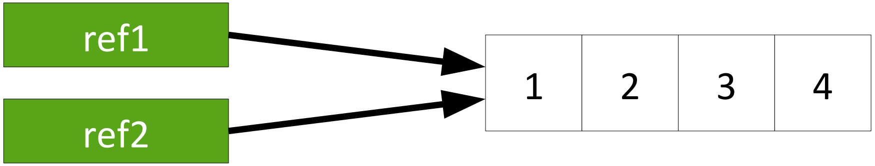
References Example (Java)

int[] ref1 = null;

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

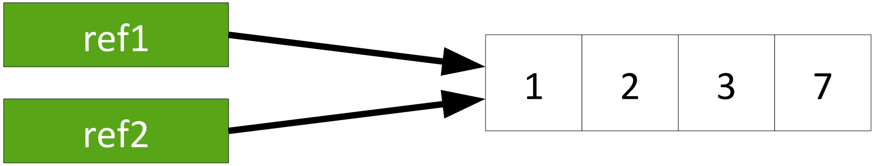


int[] ref2 = ref1;

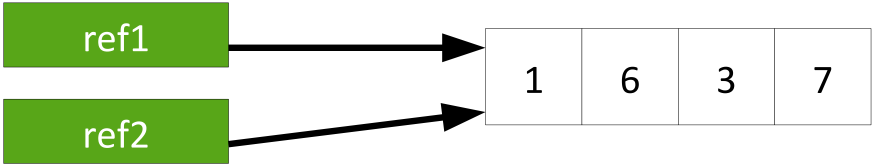


ref1[3]=7;

demo



ref2[1]=6;



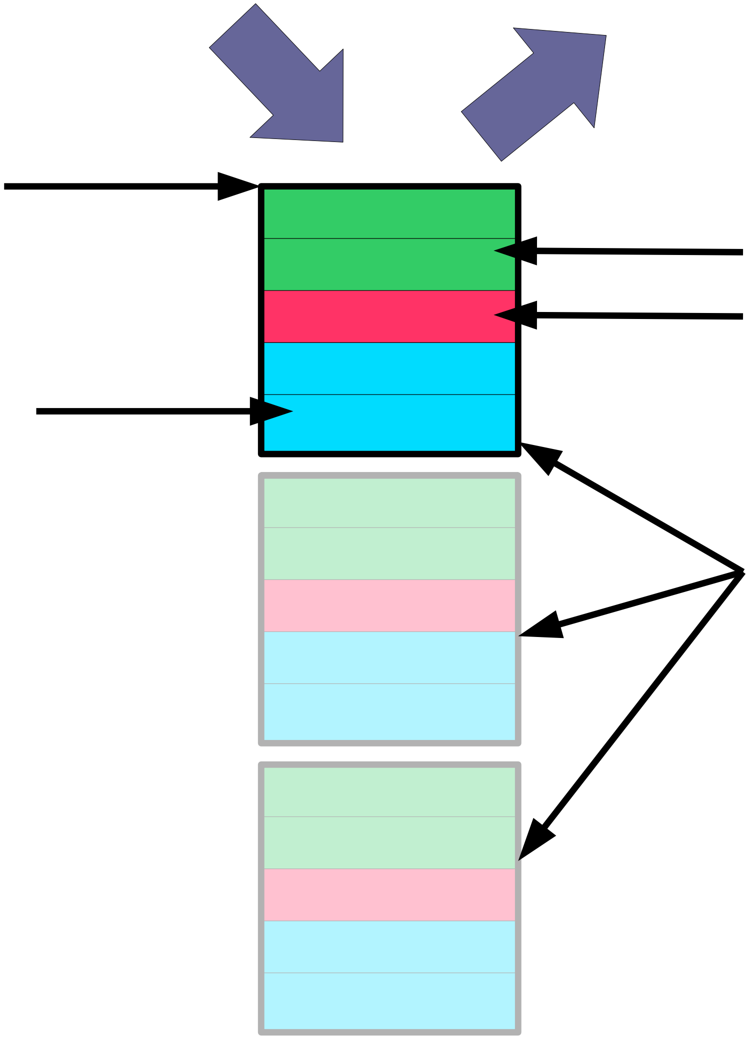
Keeping Track of Function Calls

- We need a way of keeping track of which functions are currently running

```
public void a() {
  //...
}

public void b() {
  a();
}
```

Skipped

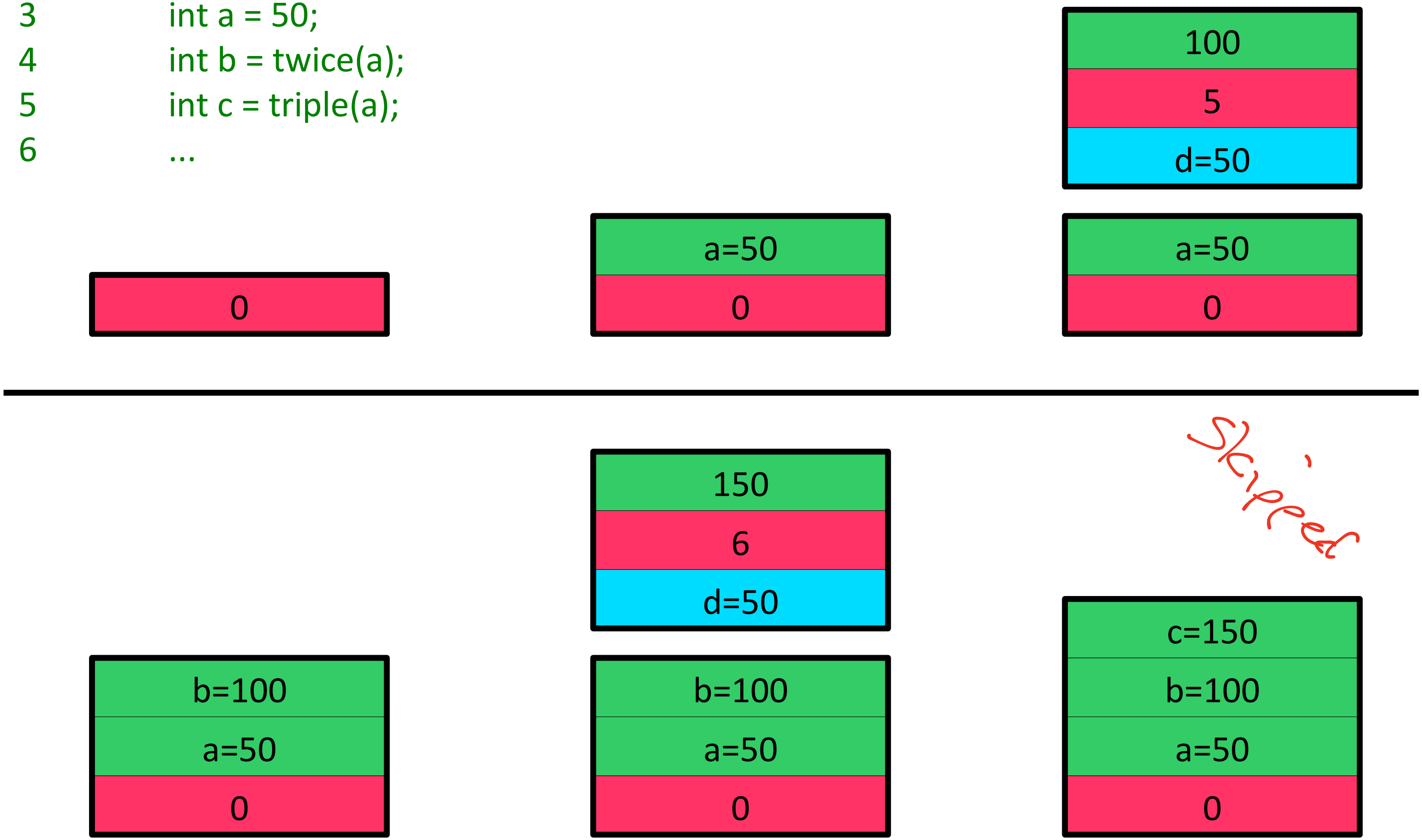
The Call Stack



Skipped

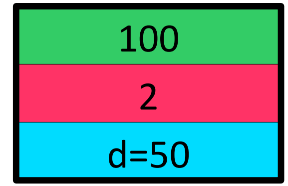
The Call Stack: Example

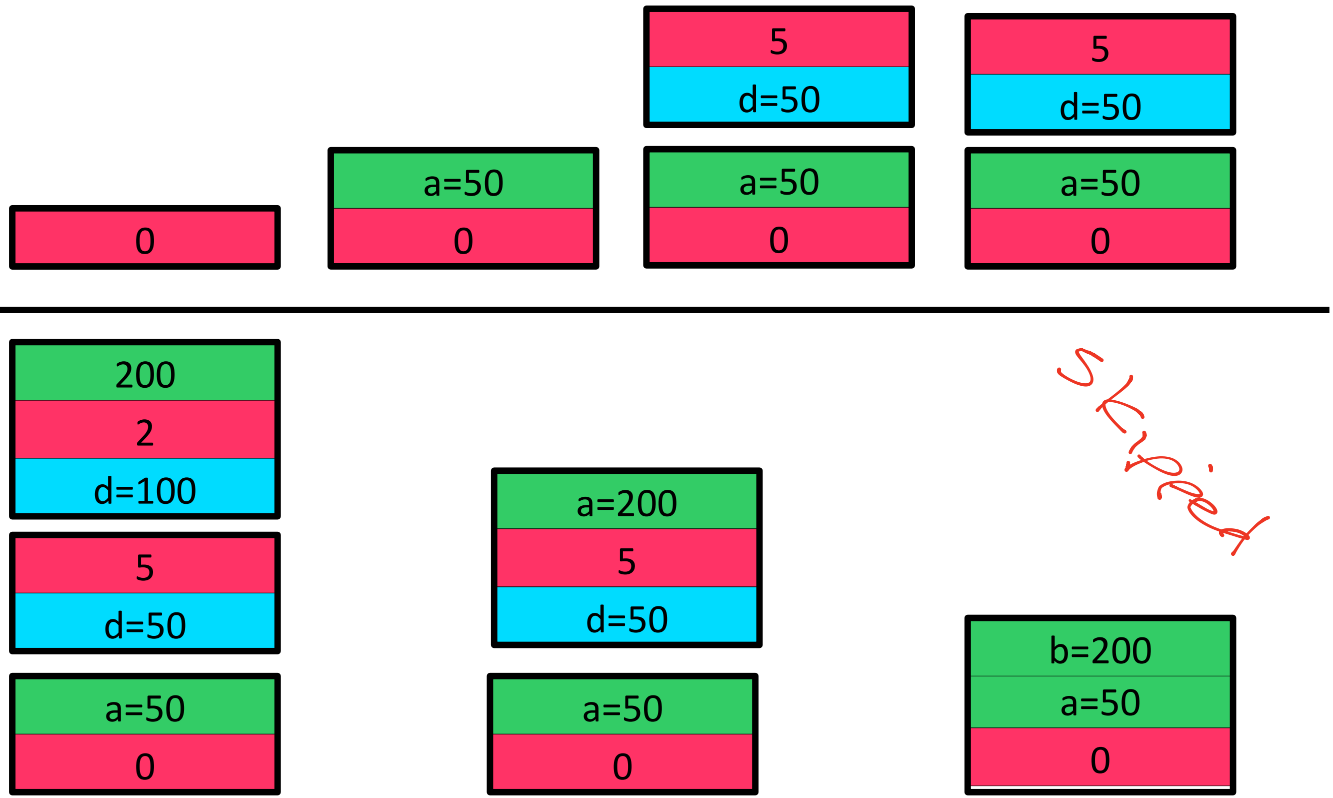
```
int twice(int d) return 2*d;
int triple(int d) return 3*d;
int a = 50;
int b = twice(a);
int c = triple(a);
...
```



Nested Functions

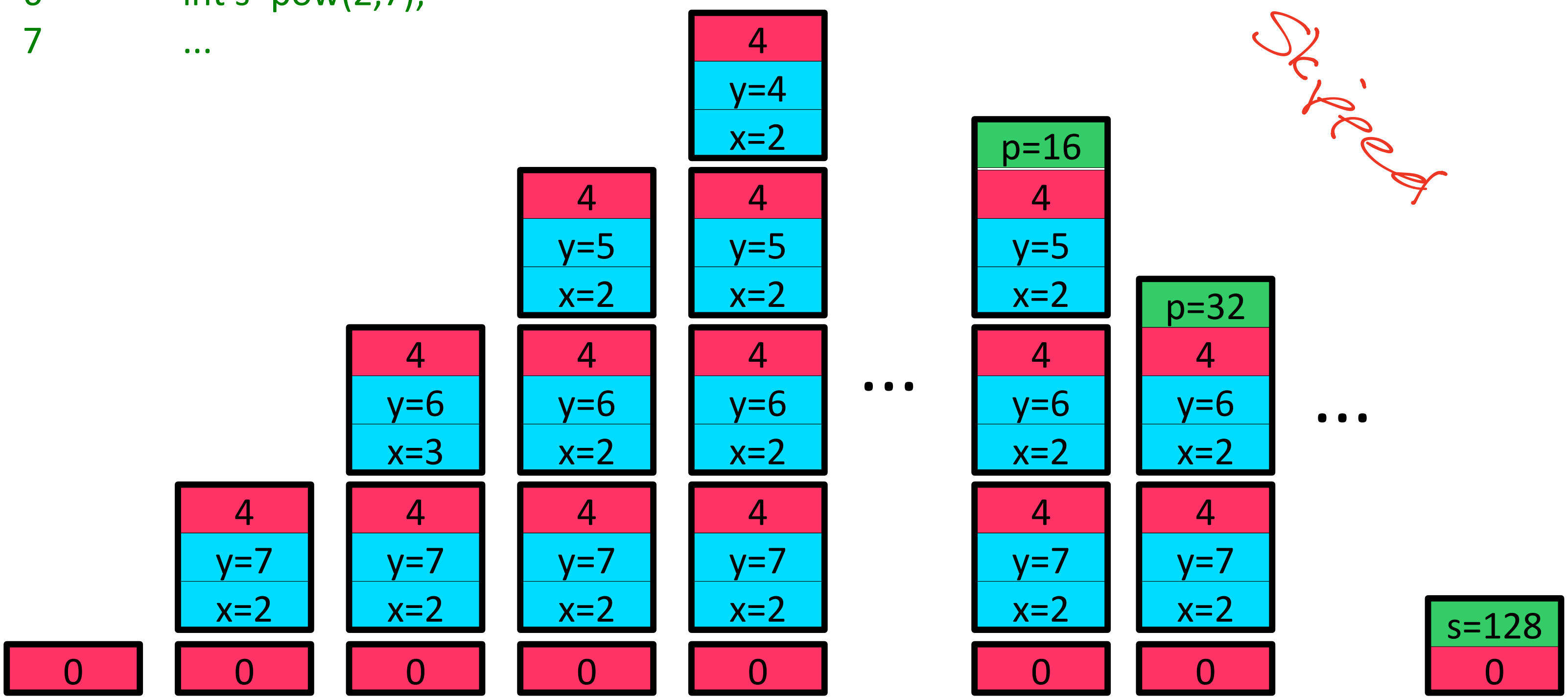
```
int twice(int d) return 2*d;
int quadruple(int d) return twice(twice(d));
int a=50;
int b = quadruple(a);
...
```





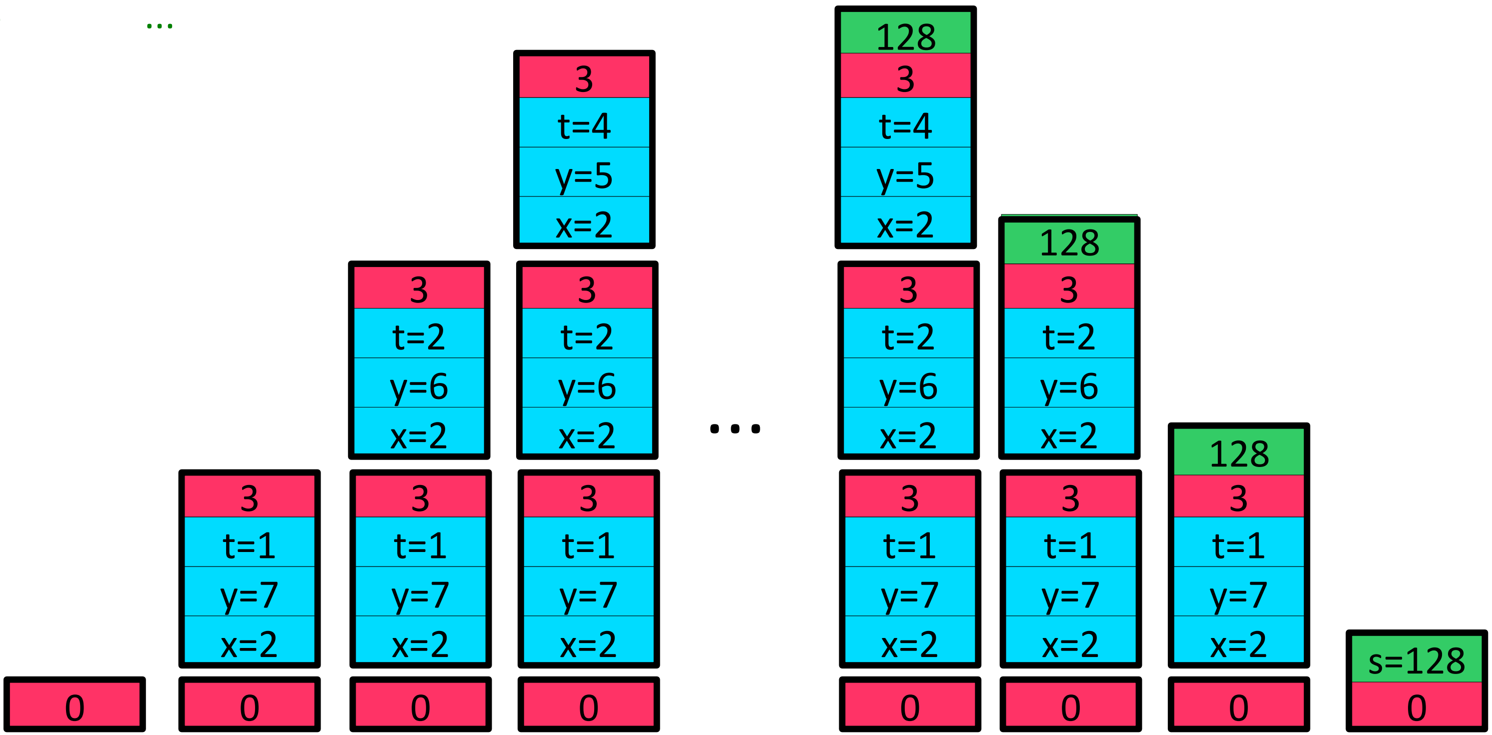
Recursive Functions

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



Tail-Recursive Functions I

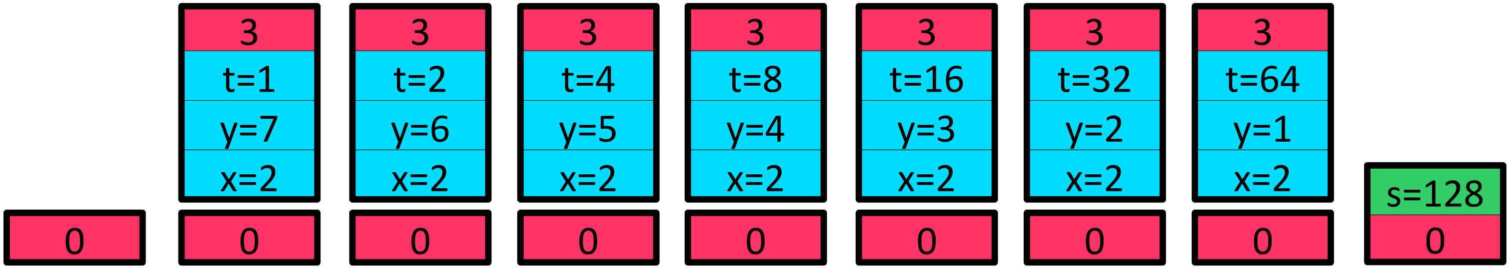
```
int pow (int x, int y, int t) {
        if (y==0) return t;
        return pow(x,y-1, t*x);
}
int s = pow(2,7,1);
...
```



Tail-Recursive Functions II

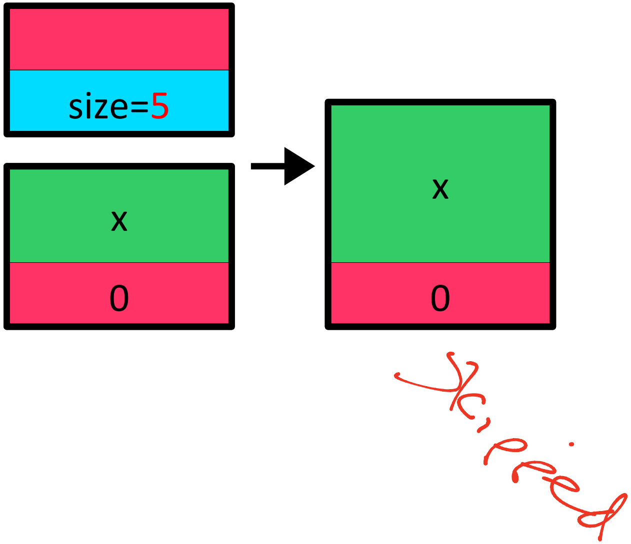
```
int pow (int x, int y, int t) {
        if (y==0) return t;
        return pow(x,y-1, t*x);
}
int s = pow(2,7,1);
...
```

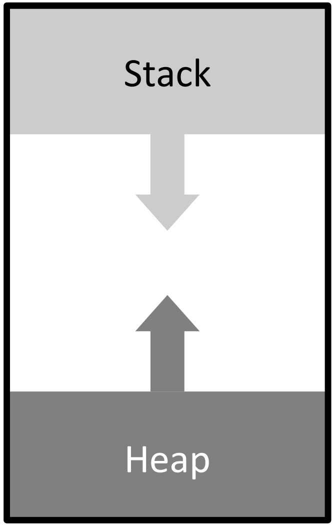
Java does not apply this optimisation

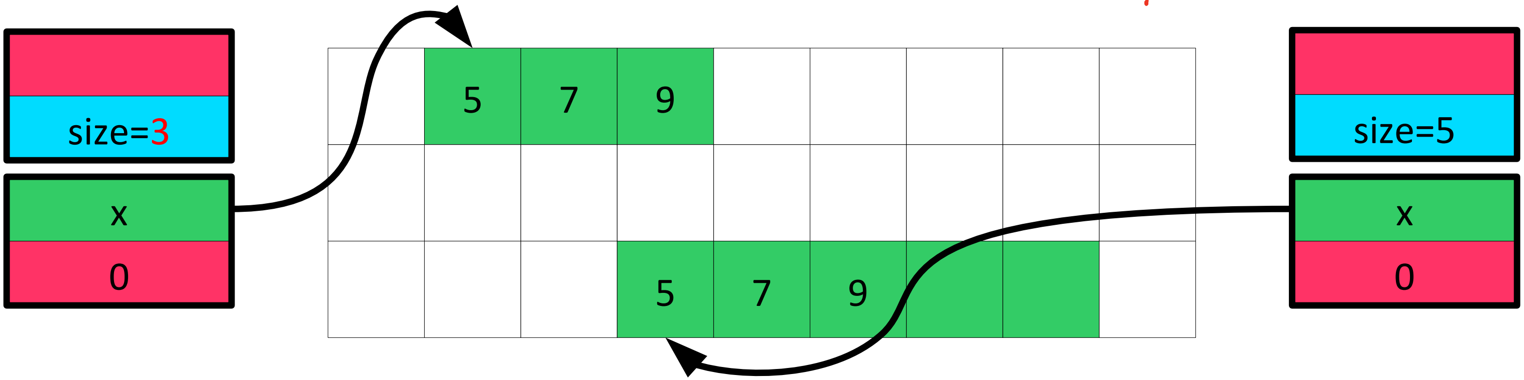


The Heap

```
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);
```





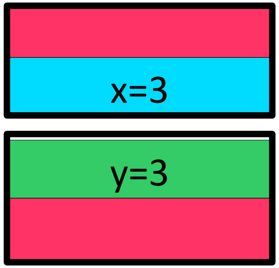


Argument Passing

(thing)

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

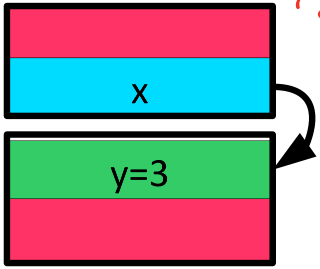
```
void test(int x) {...}
int y=3;
test(y);
```



Java only does pass by value

- **Pass-by-reference**. Create a reference to the object and pass that.

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



C++ can do this (and by value)

Passing Procedure Arguments In Java

the value here is an int

```
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 a reference

prints 1

prints 2

Passing Procedure Arguments In C++

pass by reference

```
void update(int i, int &iref){
  i++;
  iref++;
}
```

pass by value

```
int main(int argc, char** argv) {
  int a=1;
  int b=1;
  update(a,b);
  printf("%d %d\n",a,b);
}
```

Danger! how do you know this is p-b-r ?

b is changed

Check... Quiz

```
public static void myfunction2(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};

        myfunction2(num, numarray);
        System.out.println(num+" "+numarray[0]);
}
```

What does this print?

A. “1 1”
B. “1 2”
C. “2 1”
D. “2 2”

Objectives: demo for reference aliasing
argument passing
code and type inheritance
narrowing and widening again
fields and shadowing
methods and overriding

last lecture

Lecture 4:

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

demo : expression evaluator

(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;
}
```

anywhere I can use a Person I can use a Lecturer instead

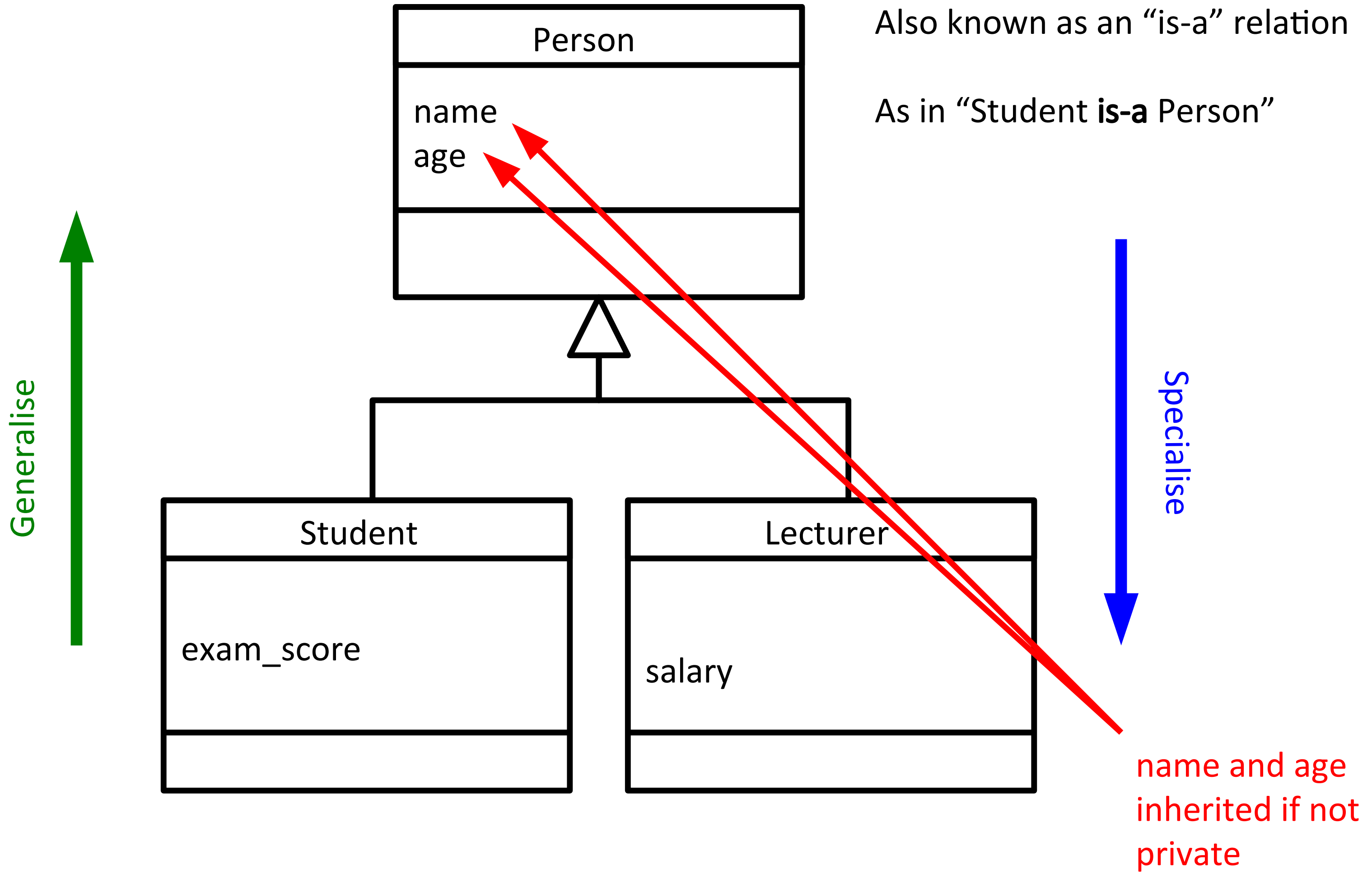
- We create a *base class* (Person) and add a new notion: classes can *inherit* properties from it
  - Both state and functionality and type!
- We say:
  - Person is the *superclass* of Lecturer and Student
  - Lecturer and Student *subclass* Person

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

Note: Java is a nominitive type language (rather than structurally typed)

important word!

Representing Inheritance Graphically



Also known as an “is-a” relation

As in “Student **is-a** Person”

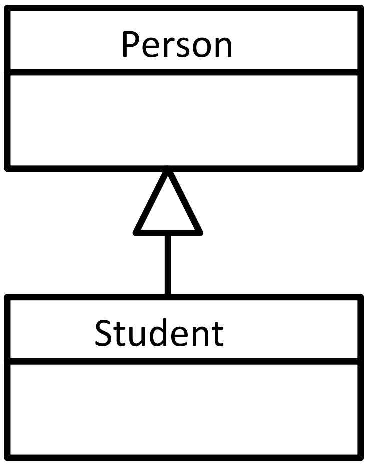
Casting

- 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 = (Person) s;
```

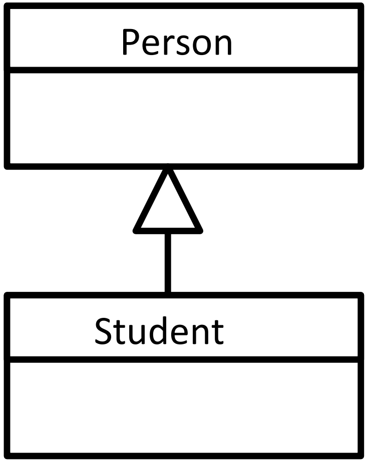
redundant cast

“Casting”

```
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

redundant cast

dangerous cast

```
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;
  }

}
```

doesn't compile

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

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 get class A extends Object {}

Object a = new A();

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();
  }
}
```

Person defines a 'default' implementation of dance()

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

Student overrides the default

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

Lecturer just inherits the default 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.

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

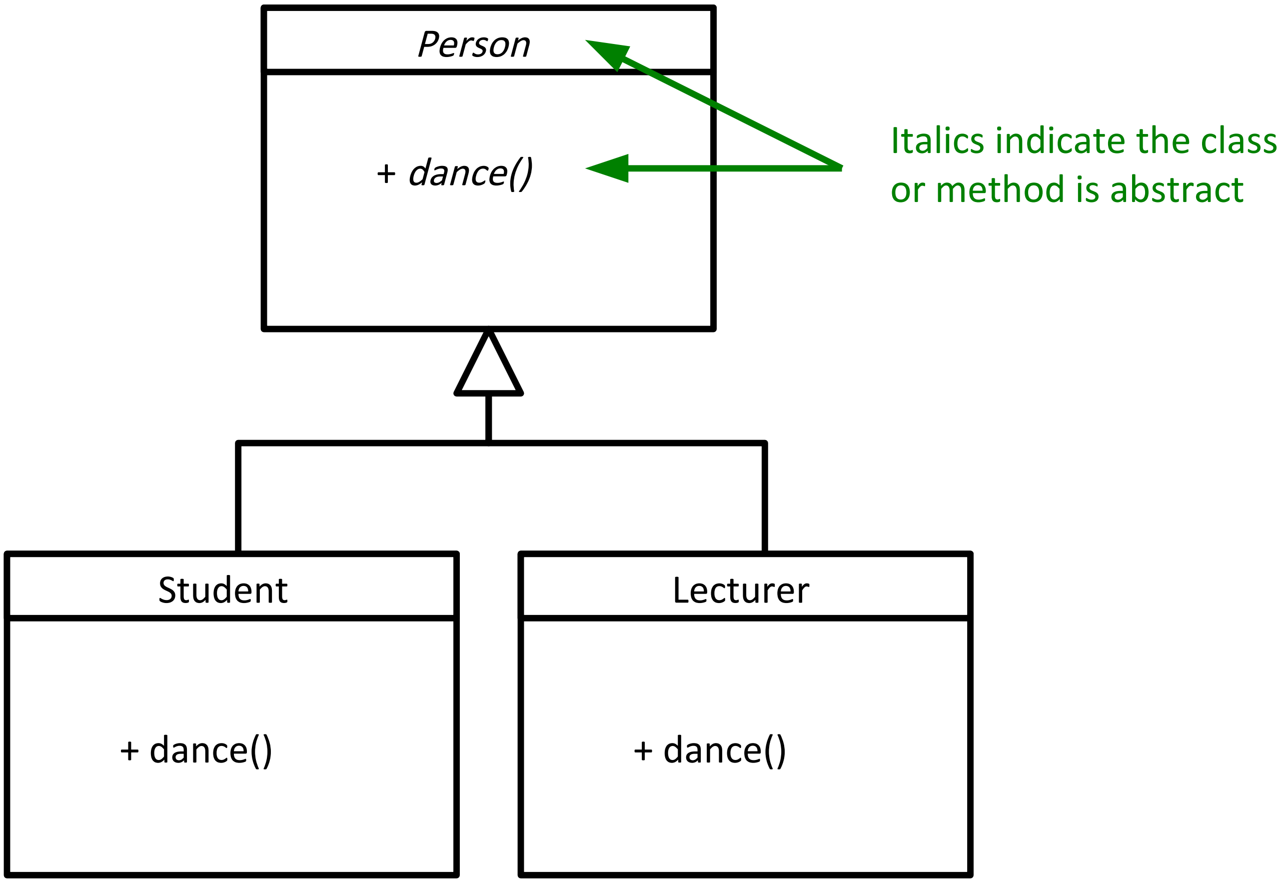
Java

```cpp
class Person {
  public:
    virtual void dance()=0;
}
```

C++

- 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



Objectives: 1) recap on abstract from last time
2) dynamic and static polymorphism
3) problems that arise from multiple inheritance
4) Java interfaces (type inheritance)

Lecture 5:
Polymorphism and Multiple Inheritance

Polymorphic Methods

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

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

demo: 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 ~~parent~~ method based on its static type
  - ~~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

C++ can do this. Java cannot

Polymorphic Concepts II

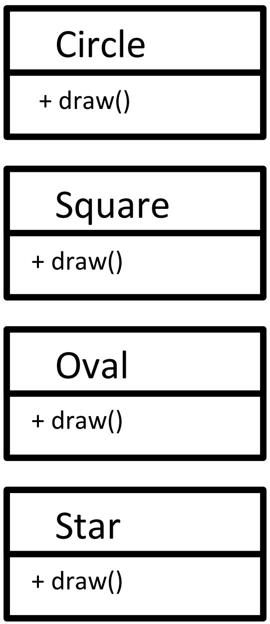
- **Dynamic** polymorphism
  - Run the method in the child
  - Must be done at run-time since that's when we know the child's type
  - ~~Type errors cause run-time faults (crashes!)~~

```
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

C++ can do this, so does Java

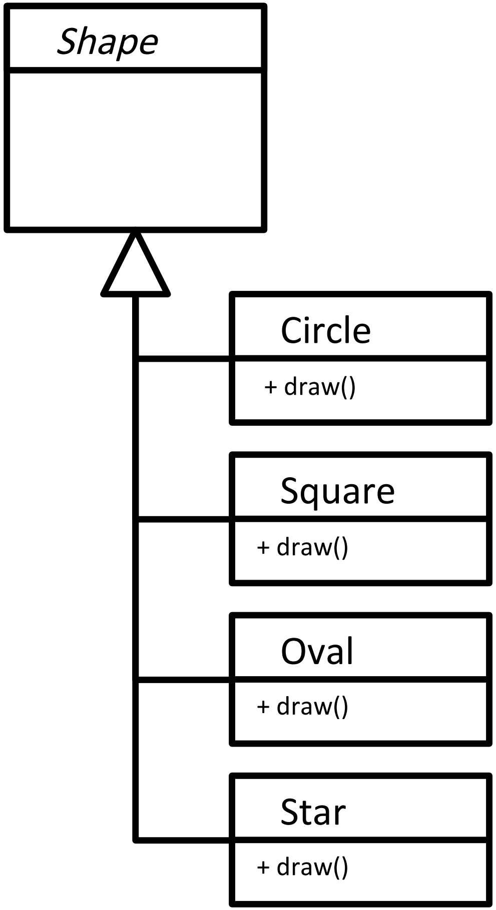
The Canonical Example I



- 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?

demo

The Canonical Example II



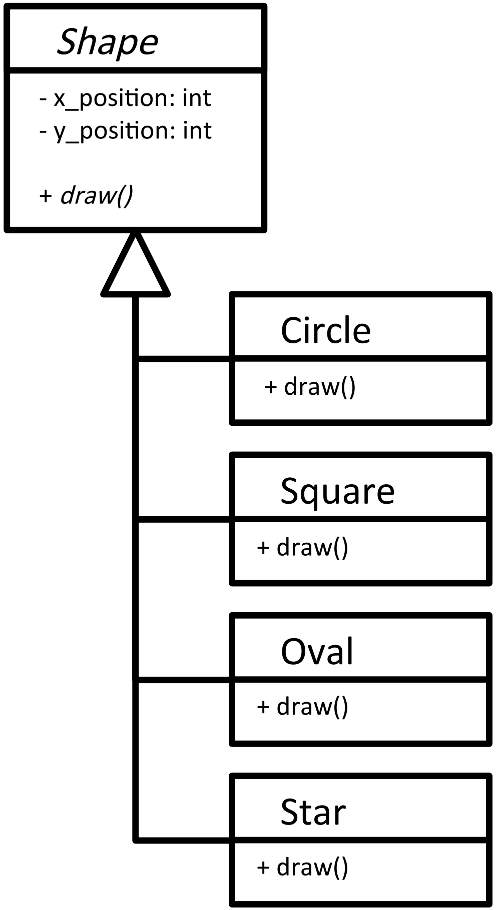
- **Option 2**
  - Keep a single list of Shape references
  - Figure out what each object really is, narrow the reference and then draw()

```
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?

demo

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

demo

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

  - What if we want to add a new shape?

this is called 'dynamic dispatch'

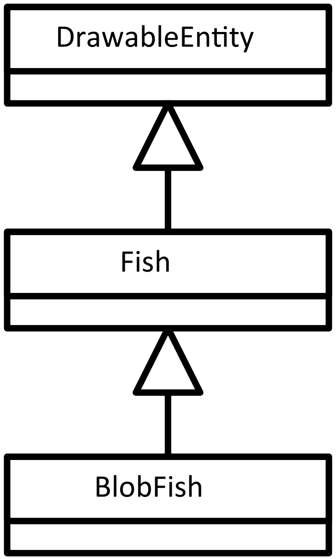
Implementations

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

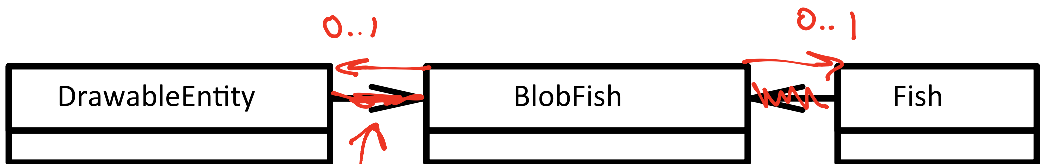
- Polymorphism in OOP is an extremely important concept that you need to make <u>sure</u> you understand...

Harder Problems

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



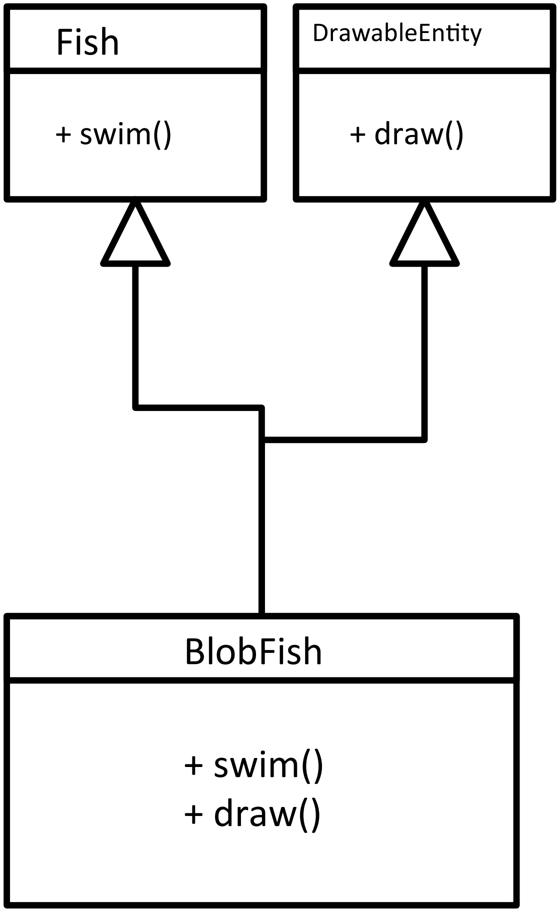
X Dependency between two independent concepts



X Conceptually wrong

remember what the open arrow head means?

Multiple Inheritance

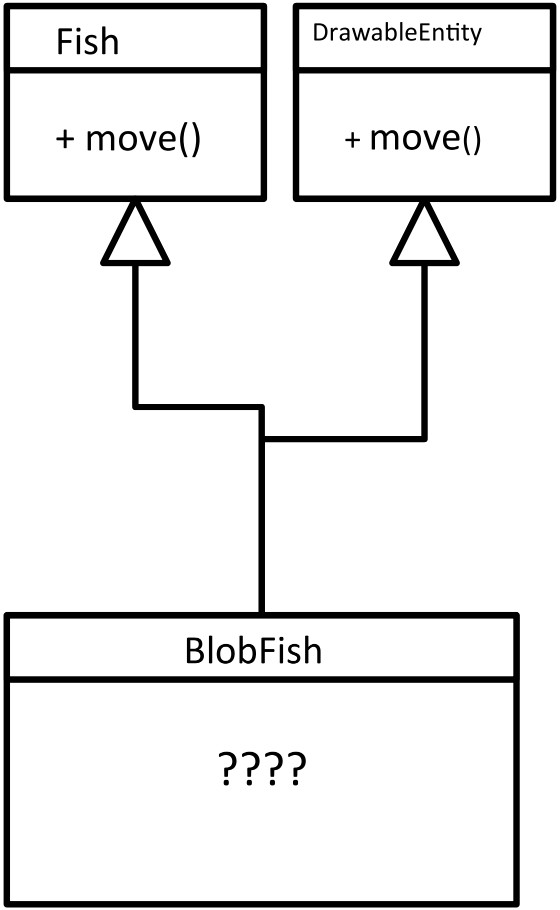


- If we multiple inherit, we capture the concept we want
- BlobFish inherits from both and is-a Fish and is-a DrawableEntity
- C++:

```
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++:

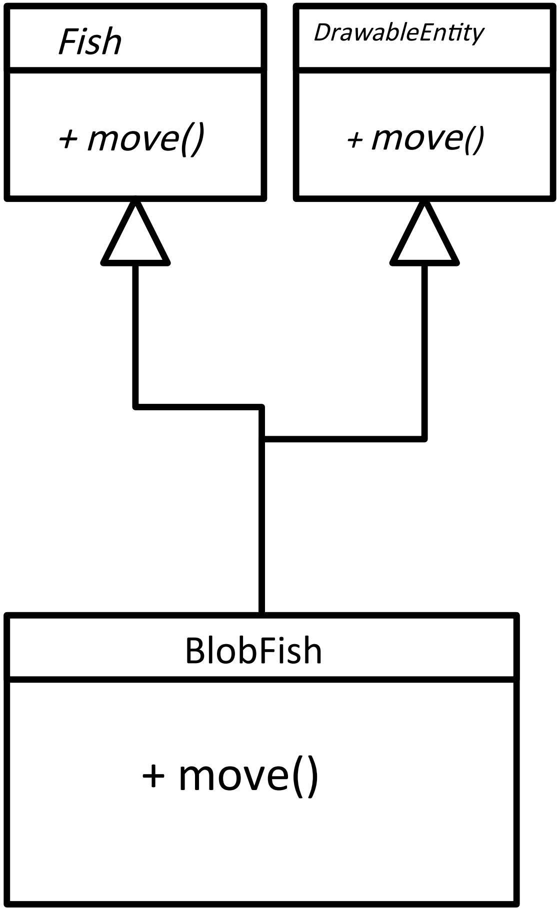
```
BlobFish *bf = new BlobFish();
bf->Fish::move();
bf->DrawableEntity::move();
```

- Yuk.

This is like shadowing e.g.

```
class A {
  int x;
}

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

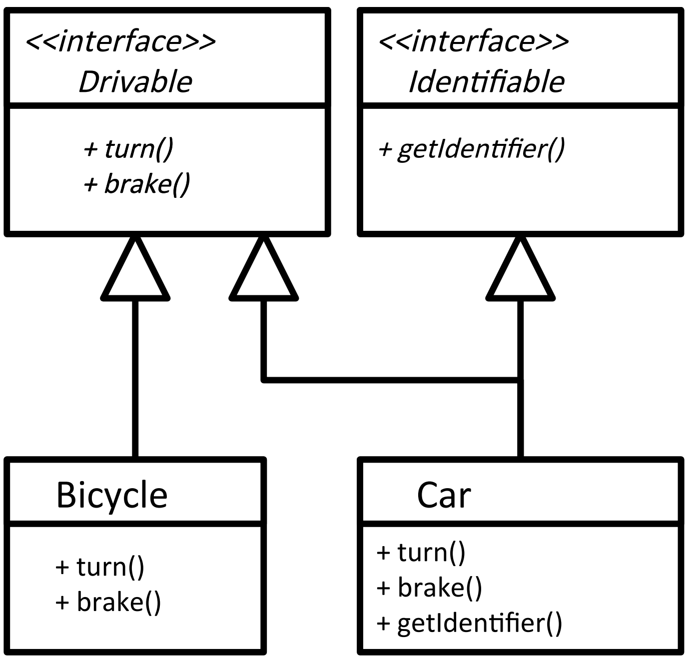
Fixing with Abstraction



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

this is type inheritance (not code)

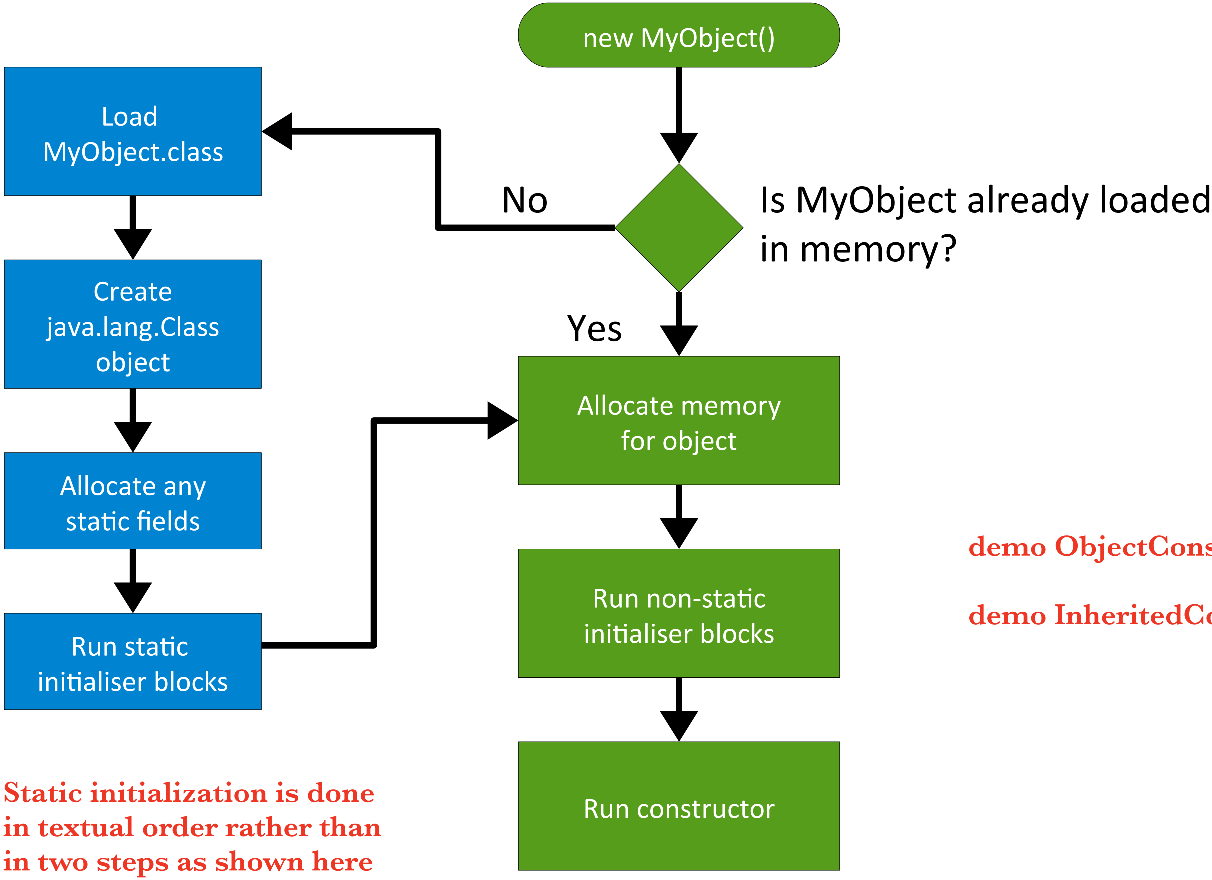
adjective

Objectives:

- **All fields in an Interface are static**
- **Know the procedure for object initialisation**
- **Difference between destructors and finalisers**
- **RAII and TWR**
- **High level idea of how a garbage collector works**

Lecture 6:
Lifecycle of an Object

Creating Objects in Java



demo ObjectConstruction

demo InheritedConstruction

Static initialization is done in textual order rather than in two steps as shown here

Initialisation Example

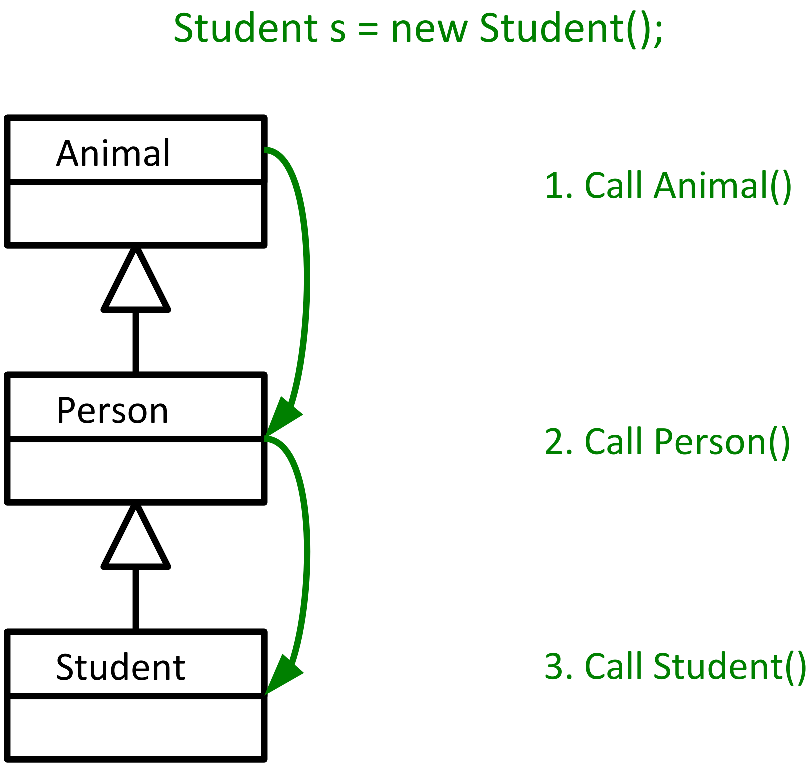
```
public class Blah {
  private int mX = 7;
  public static int sX = 9;

  {
     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

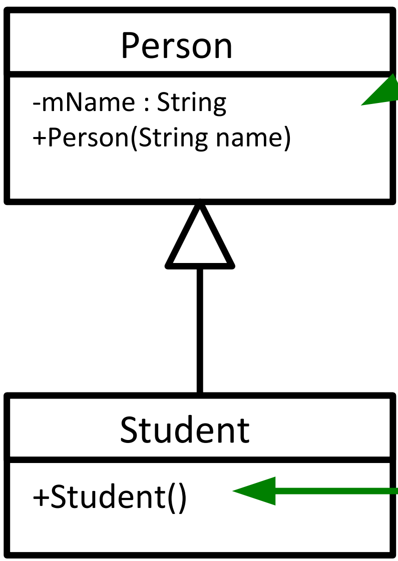
Constructor Chaining

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



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;
}
```

demo NoDefaultConstructor

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

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++):

In C++ this creates a new MyClass on the stack using the default constructor

```
void UseSmartPointer()
{
                        MyClass mc;
   unique_ptr<MyClass> *mc = new MyClass();
   // ...use mc...
}  // mc deleted here
```

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;

}
```

RAII = Resource Acquisition Is Initialization

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 ($\rightarrow$ memory leak) or we delete multiple times ($\rightarrow$ crash)
- **We can instead leave it to the system to figure out when to delete**
  - **“Garbage Collection”**
  - The system someohow 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 of short-term objects, you will soon notice the collector running
    - Can give noticeable pauses to your program!
    - But minimises memory leaks (it does not prevent them...)
- There are various algorithms: we'll look at two that can be found in Java
    - Reference counting
    - Tracing

demo Leak

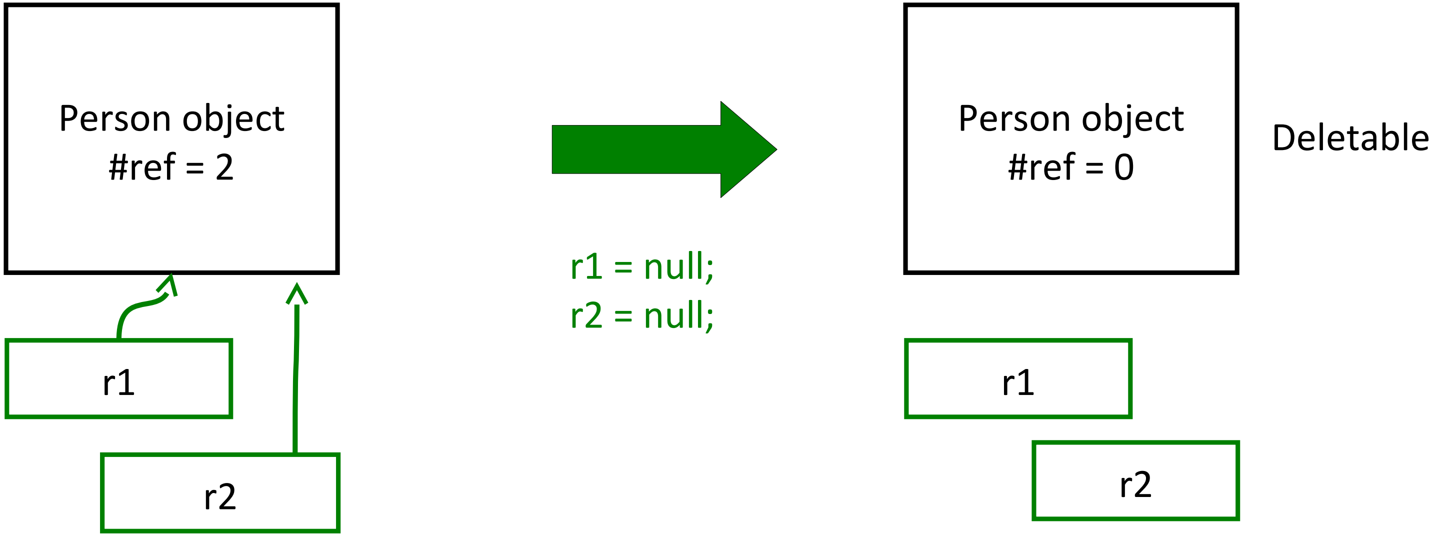
'Stop the world' - pause the operation of the program

'incremental' - garbage collect in multiple phases and let program run in the gaps

'concurrent' - no pause

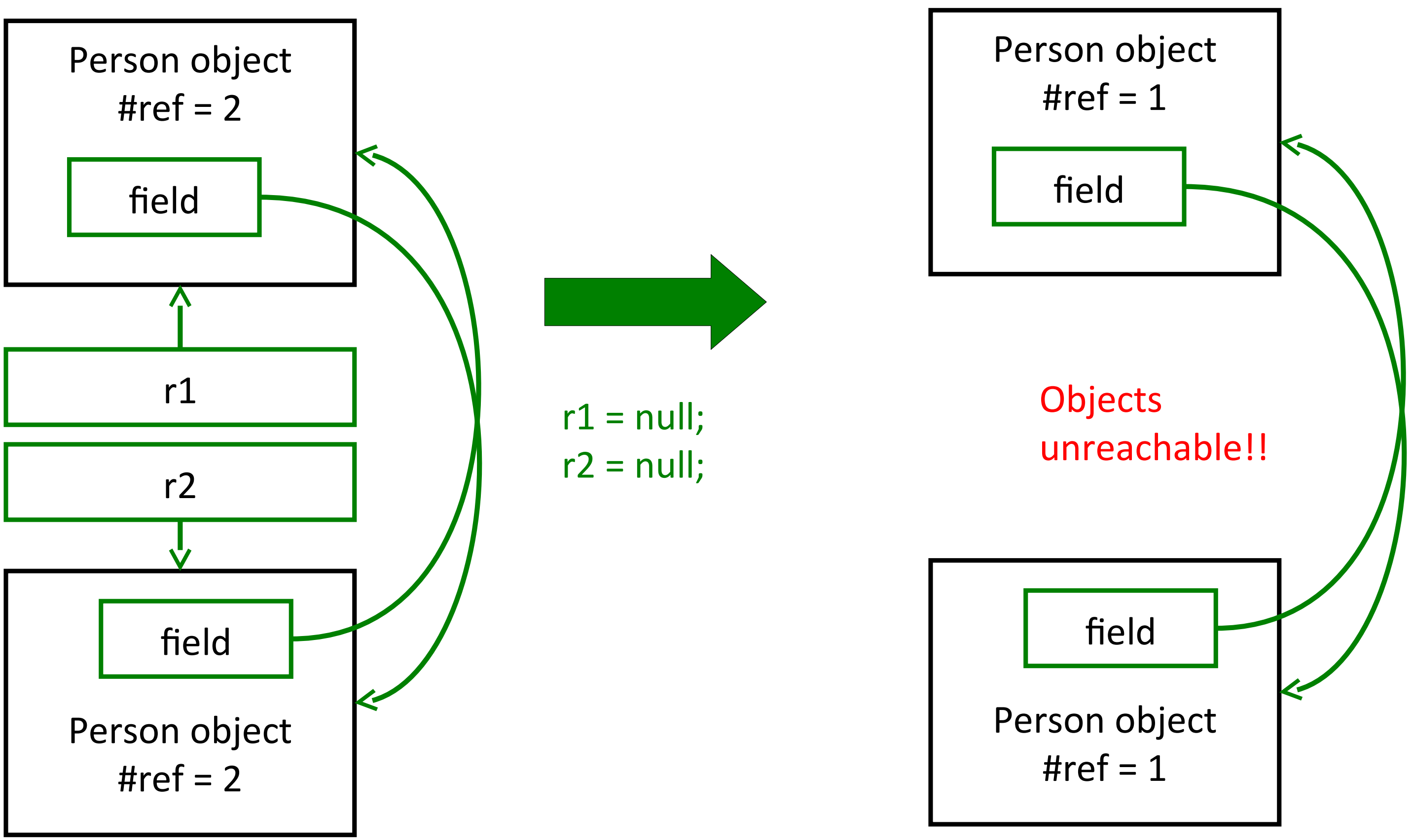
Reference Counting

- Java’s original GC. 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



Reference Counting Gotcha

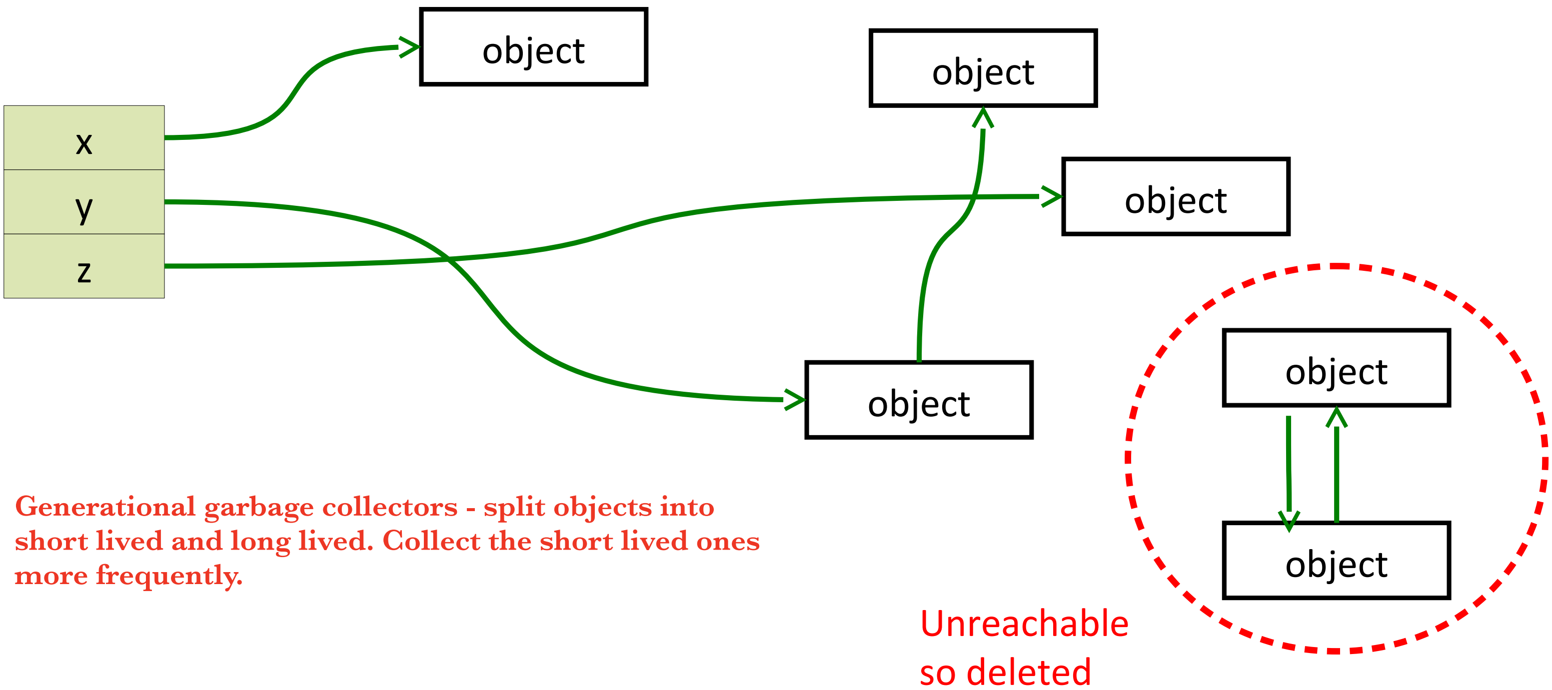
- Circular references are a pain



Tracing

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

This is called 'Mark and Sweep'



Generational garbage collectors - split objects into short lived and long lived. Collect the short lived ones more frequently.

Objectives: boxing and unboxing
set, list, queue and map
fail fast Iterators
comparing and comparable

Lecture 7:
Java Collections and Object Comparison

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...

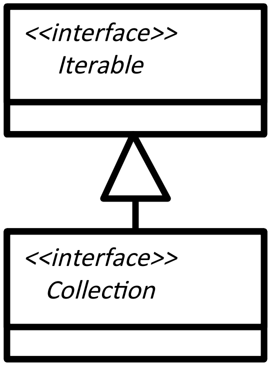
lots of this is
IB further Java

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

Digression: int and Integer

autoboxing

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")

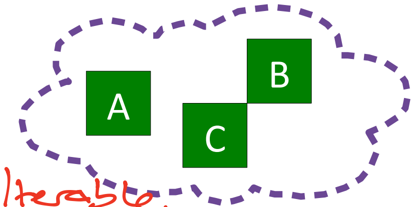
- The Collections framework has two main interfaces: Iterable and Collection. They define a set of operations that all classes in the Collections framework support
- add(Object o), clear(), isEmpty(), etc.

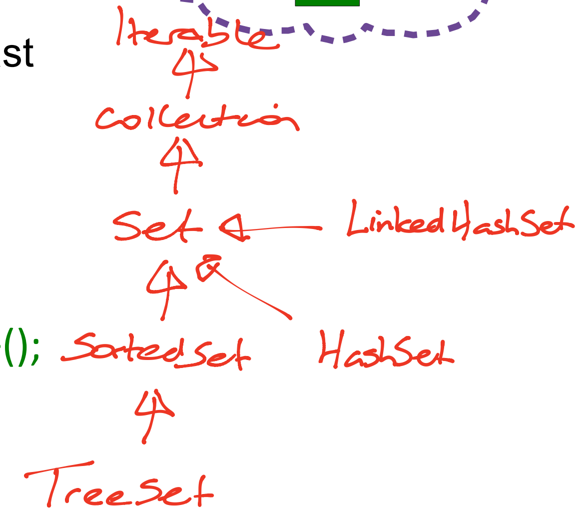
Sometimes the operation doesn't make sense - throw UnsupportedOperationException

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)

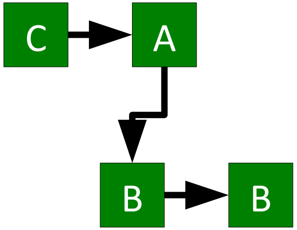





```
TreeSet<Integer> ts = new TreeSet<Integer>();
ts.add(15);
ts.add(12);
ts.contains(7);  // false
ts.contains(12); // true
ts.first(); // 12 (sorted)
```

Lists

<<interface>> List



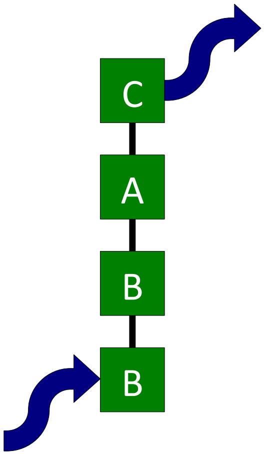
- An ordered collection of elements that may contain duplicates
- LinkedLIst: linked list of elements
- ArrayList: array of elements (efficient access)
- Vector: Legacy, as ArrayList but threadsafe

```
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)
```

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

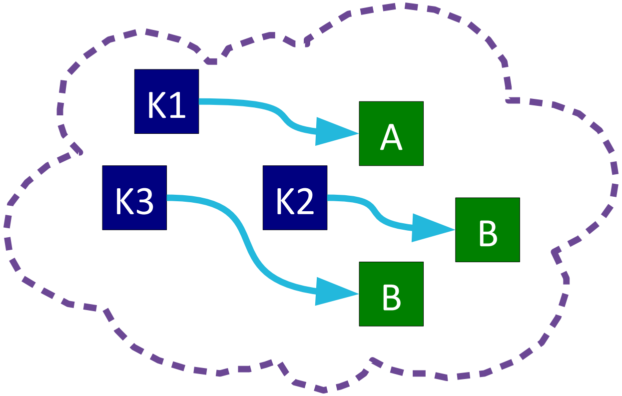



```
LinkedList<Double> ll = new LinkedList<Double>();
ll.offer(1.0);
ll.offer(0.5);
ll.poll(); // 1.0
ll.poll(); // 0.5
```

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 (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
```

Show summary table handout

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

demo

fail fast behavior

```
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();
}
```

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?

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);      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));
  }
}
```

demo: what's wrong with equals

learn the 'equals' contract

Aside: Use The Override Annotation

- It's so easy to mistakenly write:

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

   public boolean equals(EqualsTest e) {
      return (this.x==e.x);
   }

   public static void main(String args[]) {
      EqualsTest t1 = new EqualsTest();
      EqualsTest t2 = new EqualsTest();
      Object o1 = (Object) t1;
      Object o2 = (Object) t2;
      System.out.println(t1.equals(t2));
      System.out.println(o1.equals(o2));
   }
}
```

Aside: Use The Override Annotation II

- Annotation would have picked up the mistake:

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

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

   public static void main(String args[]) {
      EqualsTest t1 = new EqualsTest();
      EqualsTest t2 = new EqualsTest();
      Object o1 = (Object) t1;
      Object o2 = (Object) t2;
      System.out.println(t1.equals(t2));
      System.out.println(o1.equals(o2));
   }
}
```

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()

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

remember this with integers this - 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

ie. $x.\text{compareTo}(y) == 0 \iff x.\text{equals}(y)$

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++

People argue about whether this is good or bad

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


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

Objectives:

- finish last lecture: equals, comparing and comparable
- error handling approaches
- pros and cons of exceptions
- how to define your own exceptions

Lecture 8:
Error Handling Revisited

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

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
}
```

demo: catching multiple exceptions

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 arises due to another exception 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

discuss Throwable and Error

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) {
}
```

Always write something
If it can't happen throw a RuntimeException
If its ignored explain why

- ...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

Evil III: Circumventing Exception Handling

```
try{
  // whatever
}
catch(Exception e) {}
```

- Just don't.

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 - an exception could be thrown anywhere

Lends itself to single threads of execution

Unrolls control flow, doesn't unroll state changes

Assertions

SKIPPED

- Assertions are a form of error checking designed for debugging (only)
- 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!

SKIPPED

- 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

SKIPPED

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

Great for Postconditions SKIPPED

- 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 SKIPPED

- 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

SKIPPED

```
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 SKIPPED

```
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

SKIPPED

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

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

For the Last Word on Assertions...

SKIPPED

http://www.oracle.com/technetwork/articles/javase/javapch06.pdf

Objectives:

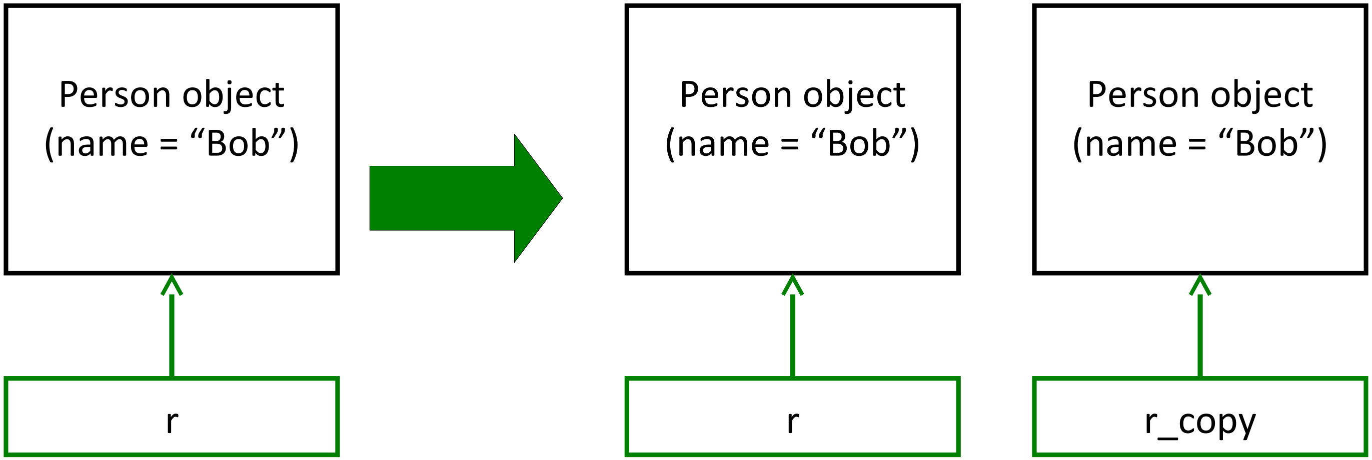
- **pros and cons of Exception handling**
- shallow vs deep copy
- covariance and contravariance
  principle of substitutability
- copy constructors

Lecture 9:
Copying Objects

Erratum: In lecture 4 I told you that Java has a nominative type system. It does. But I spelt nominative incorrectly!

Cloning I

- Sometimes we really do want to copy an object



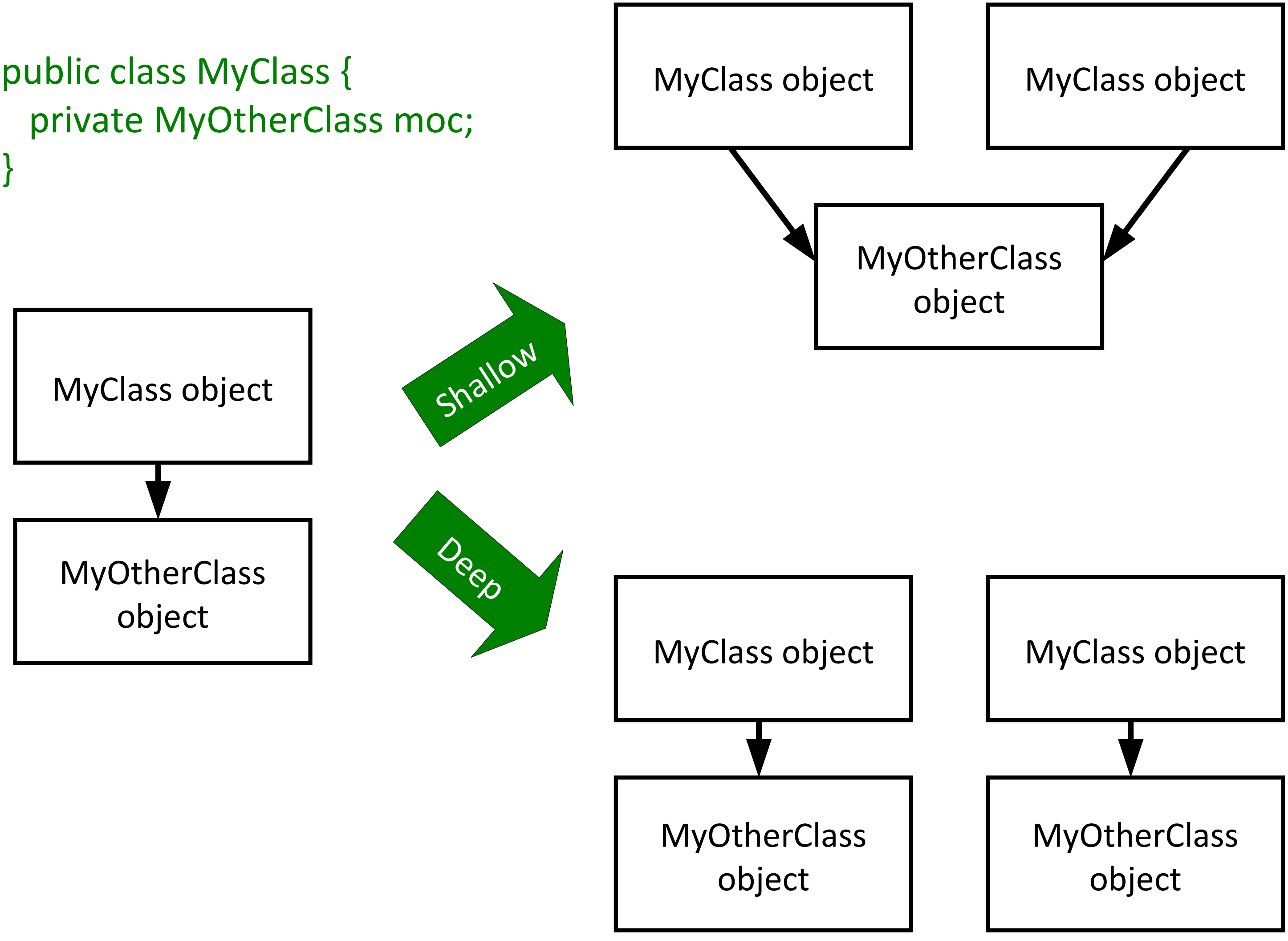
- Java calls this ***cloning***
- We need special support for it

Cloning II

- 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)

Shallow and Deep Copies

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



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

This is called a marker interface

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

demo: WeakeningAccess

When you override a method you can weaken the access modifier

```
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();
  }

};
```

Clone is protected in Object - this 'opens' up access to the method

This is the principle of substitutability

1 would return Vehicle here

demo covariance and contravariance

Clone Example III

public class Velocity implement Cloneable {

....   Velocity

public ~~Object~~ clone() {

return ~~super.clone~~(); (Velocity) super.clone();

}

};

public class Vehicle implements Cloneable {   'deep clone'

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();

cloned.vel = ~~(Velocity)~~vel.clone();

return cloned;

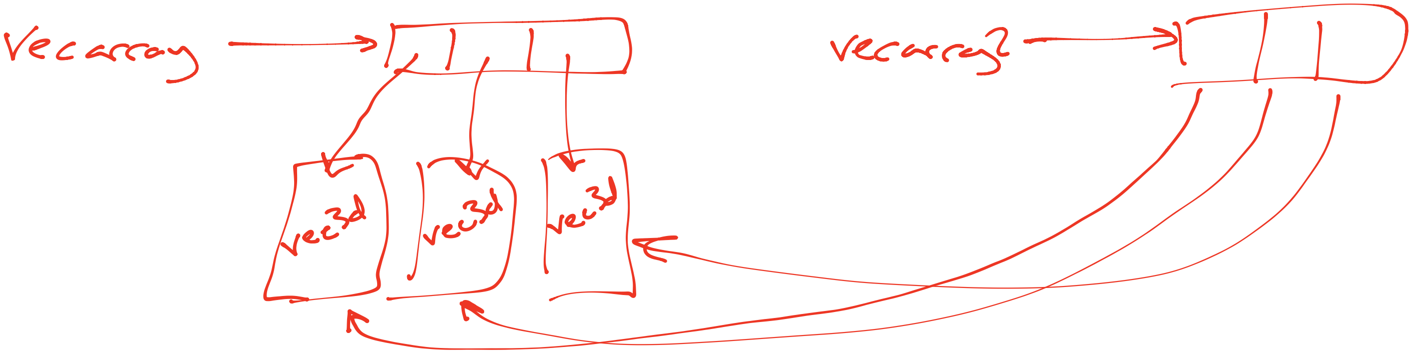
}

};

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();
```



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

Copy Constructors I

- Another way to create copies of objects is 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 Velocity vel;
   public Vehicle(int a, float vx, float vy) {
       age=a;
       vel = new Velocity(vx,vy);
   }
   public Vehicle(Vehicle v) {
              age=v.age;
              vel = v.vel.clone();
  }
}
```

Copy Constructors II

- Now we can create copies by:

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

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

- This is quite a neat approach, but has some drawbacks which are explored on the Examples Sheet

Objectives:

- **Why generics are not covariant**
- **Inner classes, anonymous inner classes, lambdas**
- **Functional interfaces**
- **Method references**
- **Streams**

Lecture 10:
Language Evolution

Evolve or Die

- Modern languages start out as a programmer “scratching an itch”: they create something that is particularly suitable for some niche
- If the language is to 'make it' then it has to evolve to incorporate both new paradigms and also the old paradigms that were originally rejected but turn out to have value after all
- The challenge is backwards compatability: you don't want to break old code or require programmers to relearn your language (they'll probably just jump ship!)
- Let's look at some examples for Java...

Vector

- The original Java included the Vector class, which was an expandable array

```
Vector v = new Vector()
v.add(x);
```

- They chose to make it *synchronised*, which just means it is safe to use with multi-threaded programs
- When they introduced Collections, they decided everything should *not* be synchronised
- Created ArrayList, which is just an unsynchronised (=better performing) Vector
- Had to retain Vector for backwards compatibility!

The Origins of Generics

```
// 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

The Origins of Generics II

```
// 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)

The Generics Solution

- Java implements *type erasure*
  - Compiler checks through your code to make sure you only used a single type with a given Generics object
  - Then it deletes all knowledge of the parameter, converting it to the old code invisibly

```
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 );
}
```

Generics has other clever stuff where you can include constraints on your generic type and also write ?'s in some places - not covered in this course

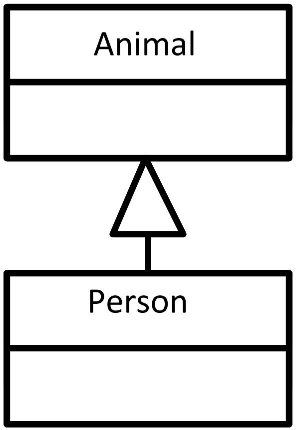
The C++ Templates Solution

- Compiler first generates the class definitions from the template

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

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

Generics and SubTyping




```
// Object casting
Person p = new Person();
Animal o = (Animal) p;

// List casting
List<Person> plist = new LinkedList<Person>();
List<Animal> alist = (List<Animal>)plist;
```

So a list of **Persons** is a list of **Animals**, yes?

```
class List<Animal> {
    Animal get() { … }
    void put(Animal a) {…}
}

    List<Animal> l =
        new List<Person>();
    Animal a = l.get();  // OK
    l.put(new Slug());   // NOT OK
```

```
class List<Person>
    extends List<Animal>{
    Person get() { … }
    void put(Person p) { … }
}
class List<Slug>
    extends List<Animal> {
    Slug get() { … }
    void put(Slug s) { … }
}
```

Adding Functional Elements...

- Java is undeniably imperative, but there is something seductive about some of the highly succinct and efficient syntax

```
result=map (fn x => (x+1)*(x+1)) numlist;
```

```
int[] result = new int[numlist.length];
for (int i=0; i<numlist.length; i++) {
   result[i] = (numlist[i]+1)*(numlist[i]+1)
}
```

Inner classes
Demo

- Enter Java 8...

Gui
GuiWithOuterClass
GuiWithInnerClass
GuiWithAnonymousInnerClass
GuiWithLambda

Lambda Functions

- Supports anonymous functions

this is a functional interface

```
()->System.out.println("It's nearly over...");
```

expression lambda

```
s->s+”hello”;

s->{s=s+”hi”;
     System.out.println(s);}

(x,y)->x+y;
```

statement lambda

```
interface Executor {
    int doSomethingGood( String a,
                         int b);
}

void run(Executor e) {
    e.doSomethingGood();
}

run( (p1,p2) -> p1 + " " + p2 );
```

Functions as ~~Values~~ instances of functional interfaces

```
// No arguments
Runnable r = ()->System.out.println("It's nearly over...");
r.run();

// No arguments, non-void return
Callable<Double> pi = ()->3.141;
pi.call();

// One argument, non-void return
Function<String,Integer> f = s->s.length();
f.apply("Seriously, you can go soon")
```

Method References

- Can use established functions too

```
System.out::println

Person::doSomething

Person::new
```

New forEach for Lists

```
List<String> list = new LinkedList<>();
list.add("Just a");
list.add("few more slides");

list.forEach(System.out::println);

list.forEach(s->System.out::println(s));

list.forEach(s->{s=s.toupperCase();
                 System.out::println(s);};
```

Sorting

- Who needs Comparators?

```
List<String> list = new LinkedList<>();

….

Collections.sort(list, (s1, s2) -> s1.length() - s2.length());
```

Streams

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

demo:
streams

```
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

Objectives:

- understand simple usage of Streams
- what is a design pattern
- open-closed principle
- some example design patterns

Lecture 11/12: 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

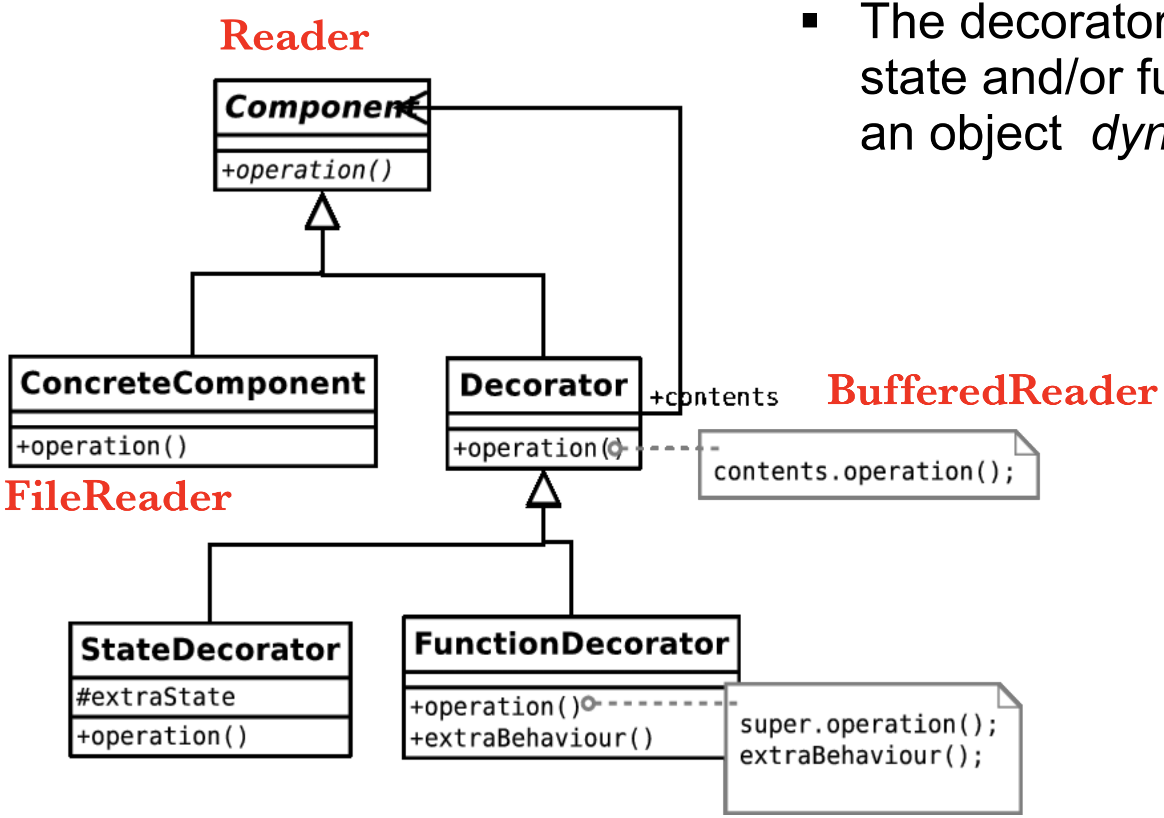
Decorator

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 object *dynamically*

Singleton

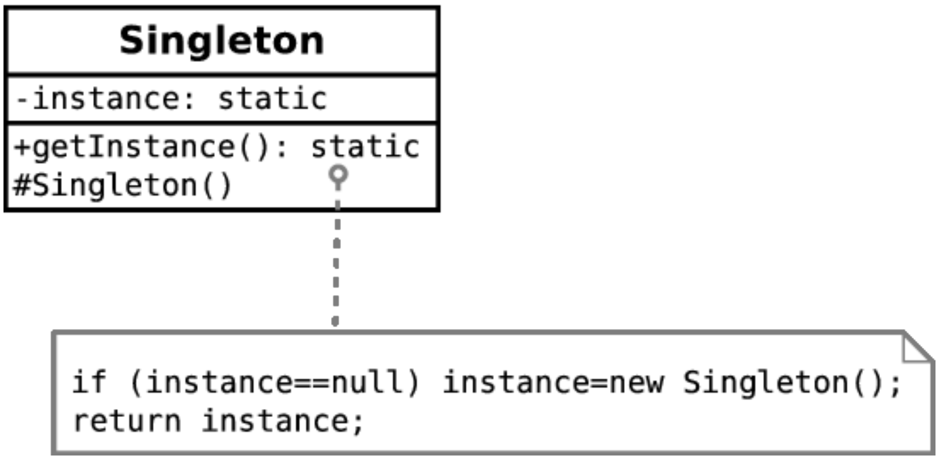
Abstract problem: How can we ensure only one instance of an object is created by developers using our code?

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



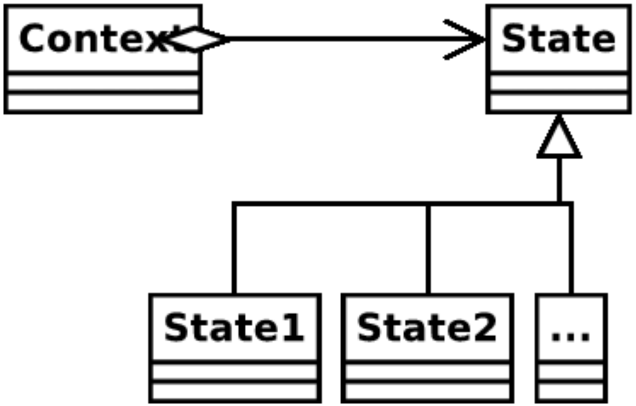
State

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

demo: FanSpeed

State in General



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

Strategy

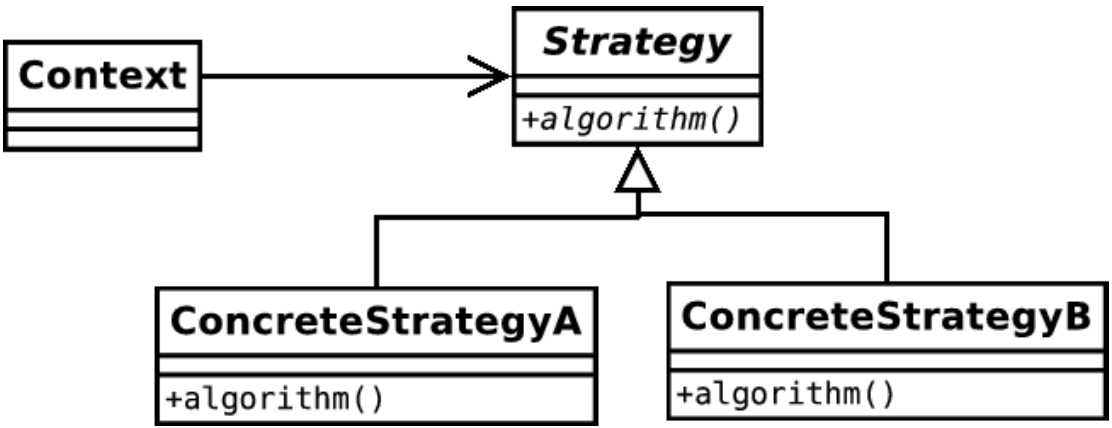
Abstract problem: How can we select an algorithm implementation at runtime?

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



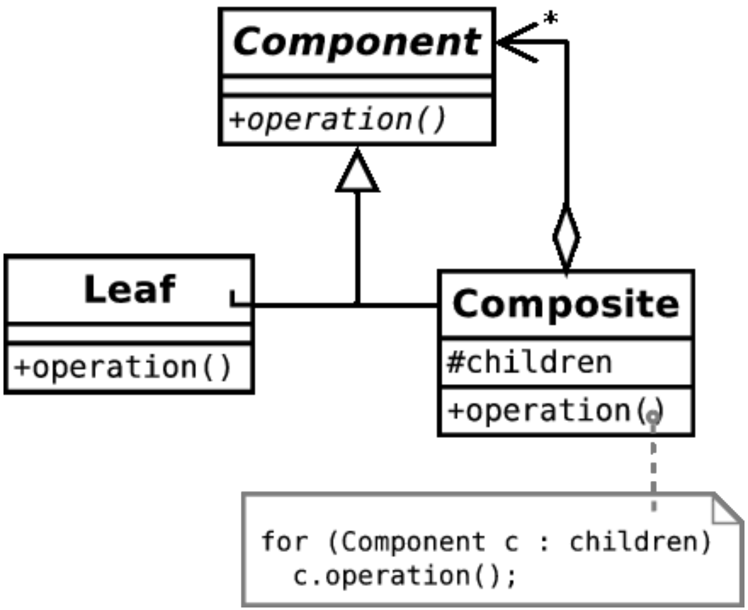
Composite

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

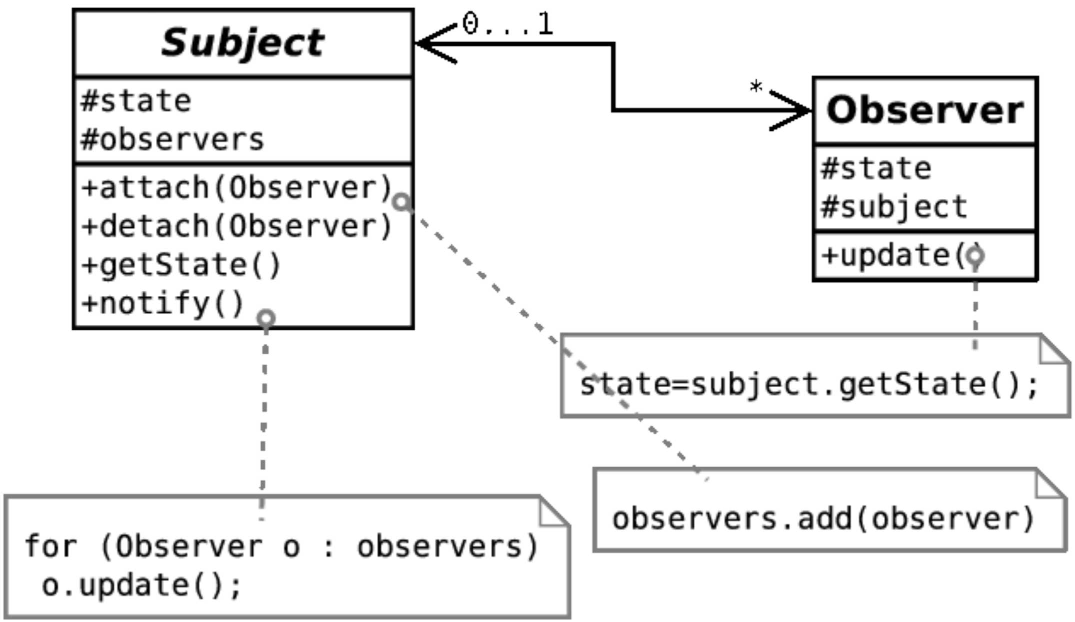
Abstract problem: When an object changes state, how can any interested parties know?

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

demo: ActionListener from last lecture

Observer in General

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



Interpreter to Virtual Machine

- *Java* was born in an era of internet connectivity. SUN wanted to distribute programs to internet machines
  - 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)?

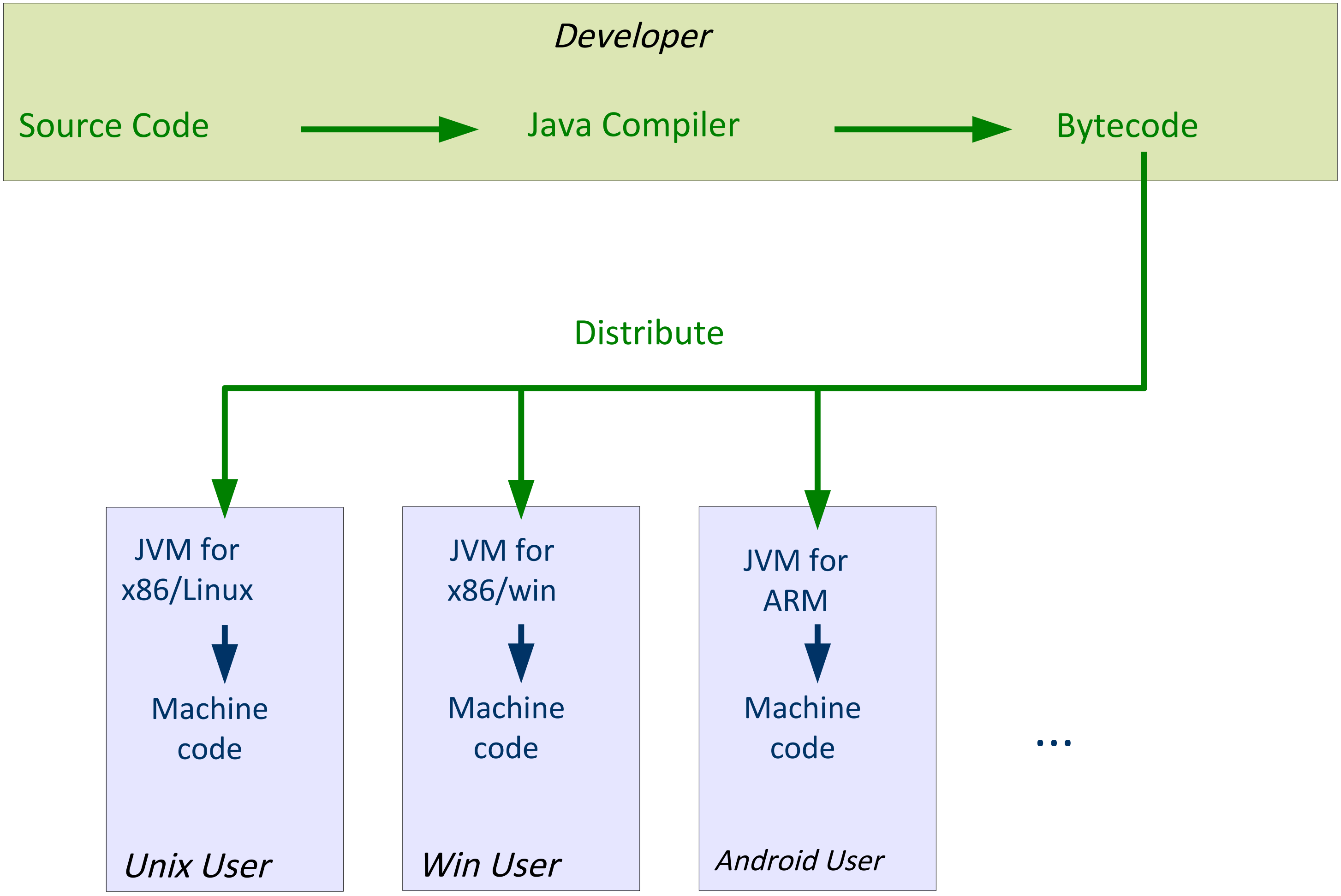
- Could use an interpreter (→ Javascript). But:
  - High level languages not very space-efficient
  - The source code would implicitly be there for anyone to see, which hinders commercial viability.

- Went for a clever hybrid interpreter/compiler

Java Bytecode I

- SUN envisaged a hypothetical Java Virtual Machine (JVM). Java is compiled into machine code (called bytecode) for that (imaginary) machine. The bytecode is then distributed.
- To use the bytecode, the user must have a JVM that has been specially compiled for their architecture.
- The JVM takes in bytecode and spits out the correct machine code for the local computer. i.e. is a bytecode interpreter

Java Bytecode II



Java Bytecode III

+ 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