Object Oriented Programming
Dr Robert Harle

IA CST, PBS (CS) and NST (CS)

Michaelmas 2014/15

The OOP Course

- So far you have studied **functional** programming (ML)
- Now we consider **imperative** programming (Java primarily but not exclusively).
- You have practicals 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*!

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

Outline

1. Types, Objects and Classes
2. Pointers, References and Memory
3. Creating Classes
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

- 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/

Lecture 1:
Types, Objects and Classes

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. "double x" might be a declarative instruction that you want the variable x doubled somehow. Imperatively we could have "x=x*2" or "x=x+x"

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.

Aside: Some things to note on Java

- Programs are compiled and then run (i.e. not interpreted 'live' as per ML).
  1) Write your .java code (e.g. Test.java)
  2) Compile using javac Test.java
  3) Run using java Test
- You need an explicit start point. This is called the main() function
- You have to name your .java files exactly the same as the class they define (this will make more sense later)

More Boilerplate than ML

HelloWorld.ml:

```
print “Hello World”;
```

HelloWorld.java:

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

Types and Variables

- Most imperative languages don't have type inference

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

Explicit types assigned

- 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

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

Two's complement

See Easter

See Workbook 1

...vs ML

- bool – 1 bit (true, false)
- int – arbitrary-precision integer
- real – arbitrary-precision floating point

Much simpler to work with

but incurs a cost over working

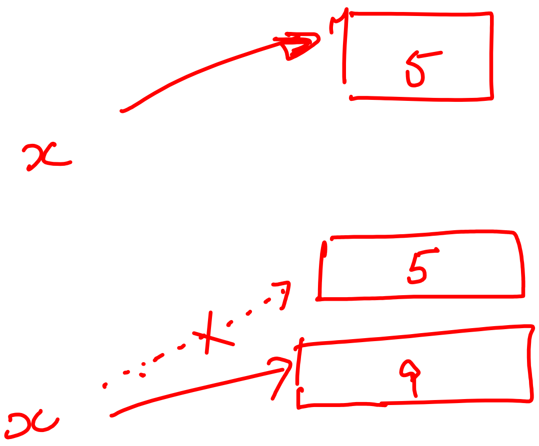
with fixed precision

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
```

Comparison



Java

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

int x=9;
```

assignment

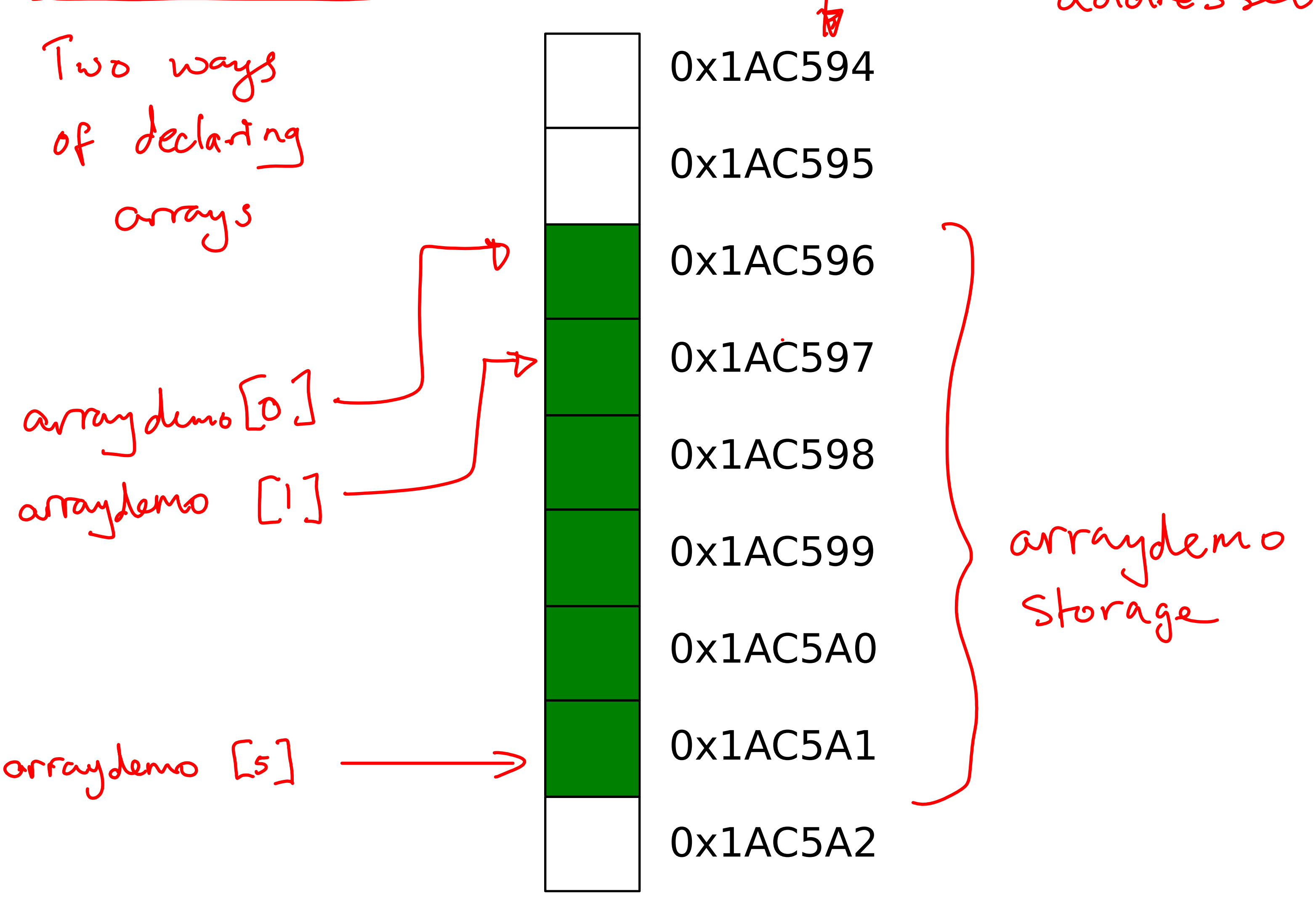
~~int x=9;~~ won't compile

(x == 9) false

(x == 5) true

Arrays

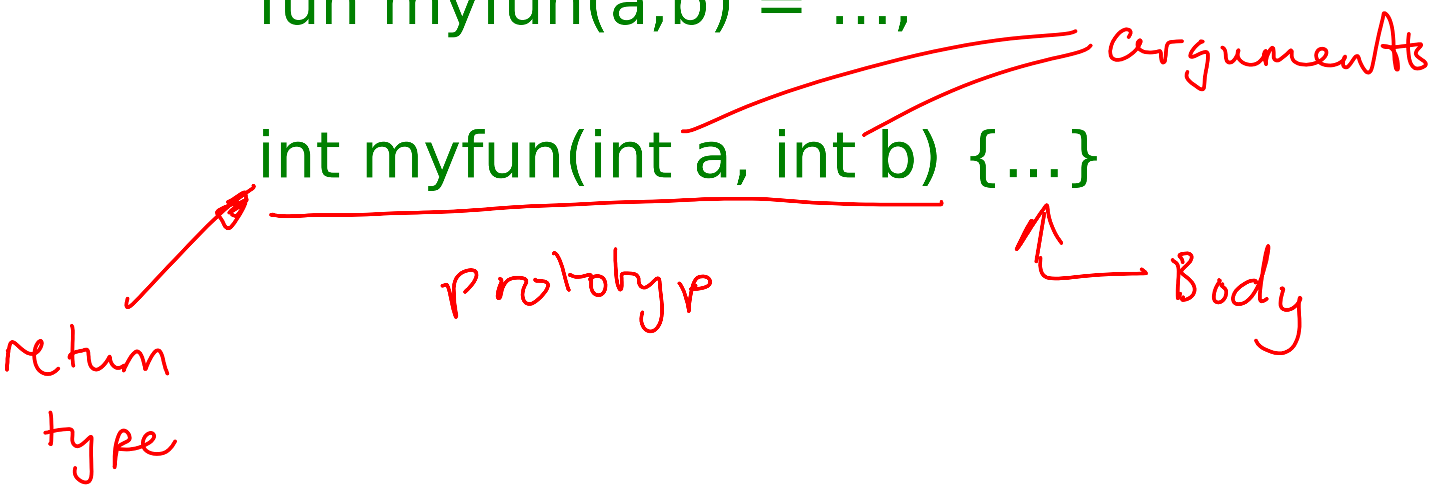
```
byte[] arraydemo = new byte[6];
byte   arraydemo2[] = new byte[6];
```



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) {...}
```



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 not just a different return type

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

X

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;

Proper functions (output depends only on arguments)

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

Procedure (output or behaviour depends on some external state)

void Procedures

- A void procedure returns nothing:

```
int count=0;

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

Meaningful only for procedures (i.e. not in ML)

Control Flow: Decision Making

```
if (boolean_expression) {
    do_something()
}
```

```
if (boolean_expression) {
    do_something()
}
else {
    do_something_else()
}
```

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--) ...
```

while(*boolean_expression* **)**

```
int i=0;  while (i<8) { i++; ...}

int j=7; while (j>=0) { j--; ...}
```

i++
post-increment
equivalent to
i; i=i+1;

Control Flow: Looping Examples

Shorthand for new int[5];

Built in length value for arrays

```
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;
}
```

O(1) search of array

Control Flow: Branching II

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

boolean

```
~~void~~ 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;
}
```

← Allows us to keep just one return point

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]);
   }
}
```

Custom Types

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

Custom type

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

Primitive building blocks

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;
   }
}
```

1 grouped concept

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.

class (custom type)

Reference to a MyCoolClass object

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

creation of an object

  makes two instances of class MyCoolClass.
- An instance of a class is called an **object**

Loose Terminology (again!)

State

Fields

Instance Variables

Properties

Variables

Members

Behaviour

Functions

Methods

Procedures

Parameterised Classes

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

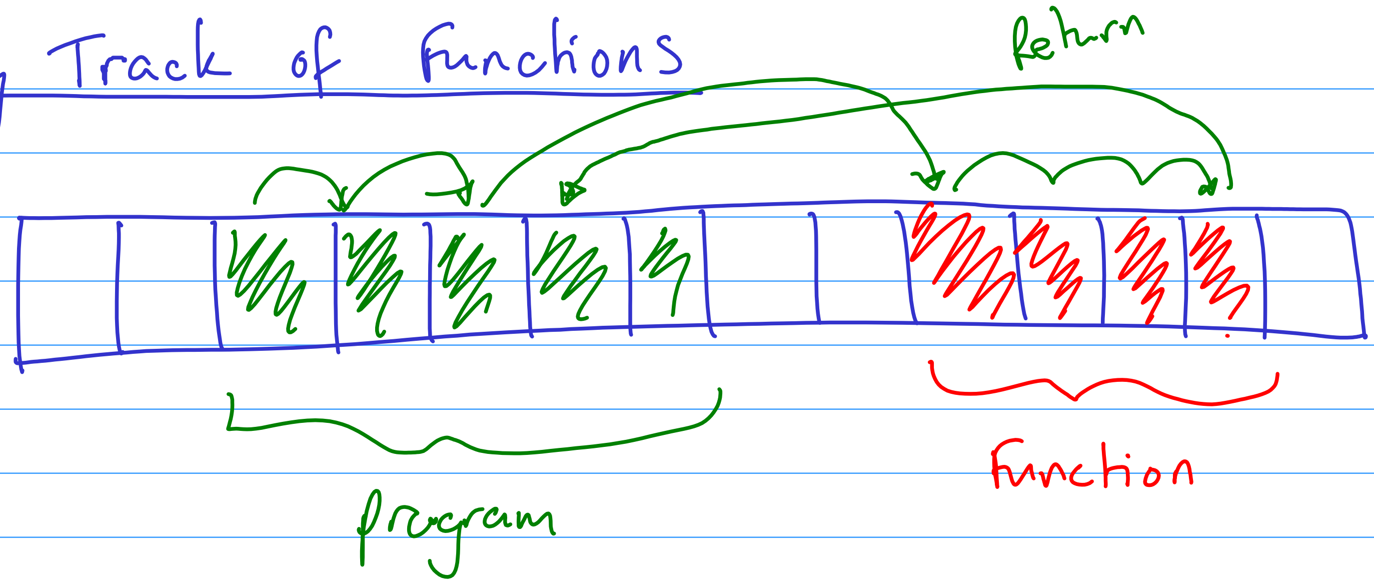
```
> fun self(x)=x;
val self = fn : 'a -> 'a
```

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

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

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

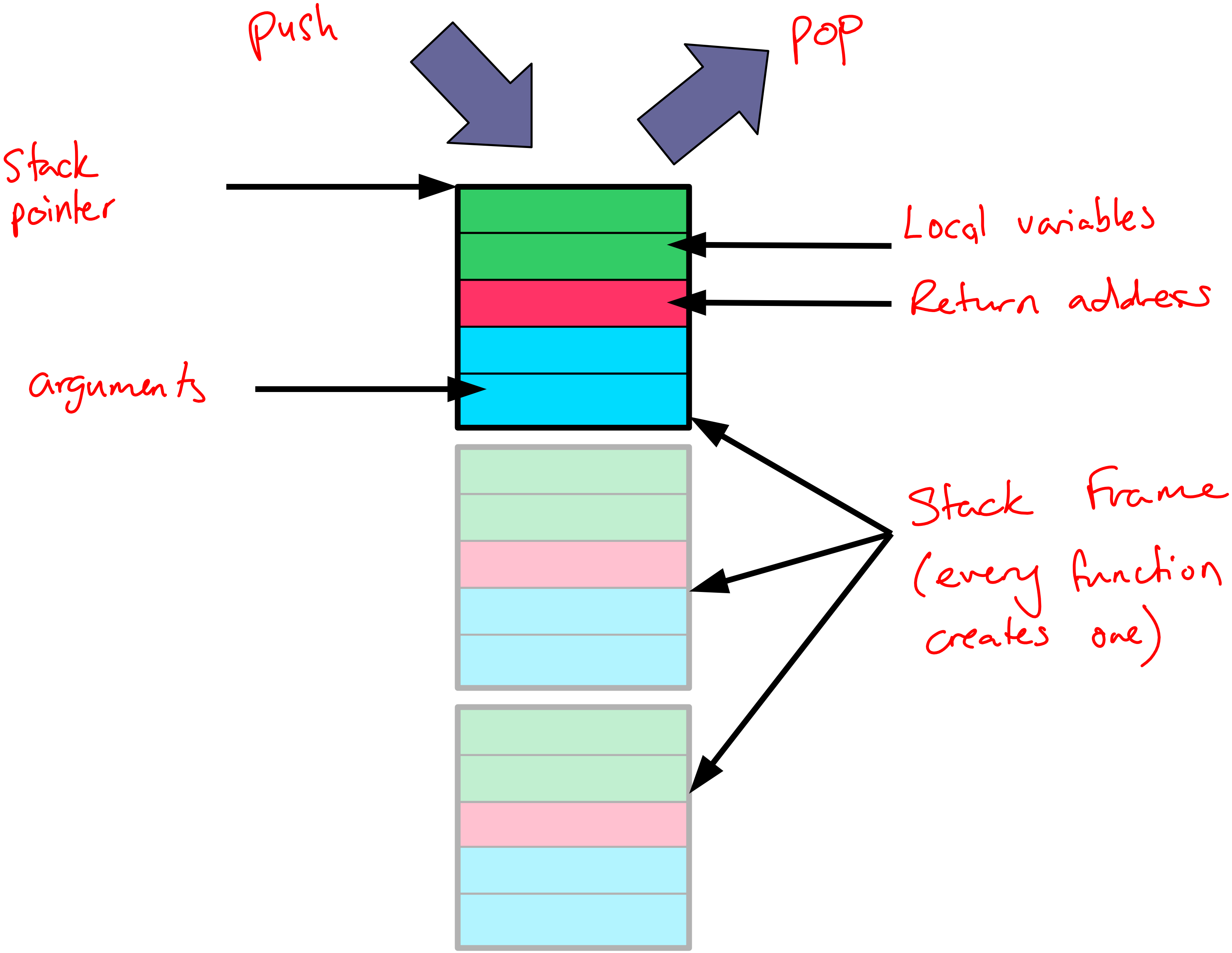
Lecture 2:
Pointers, References and Memory

Keeping Track of Functions



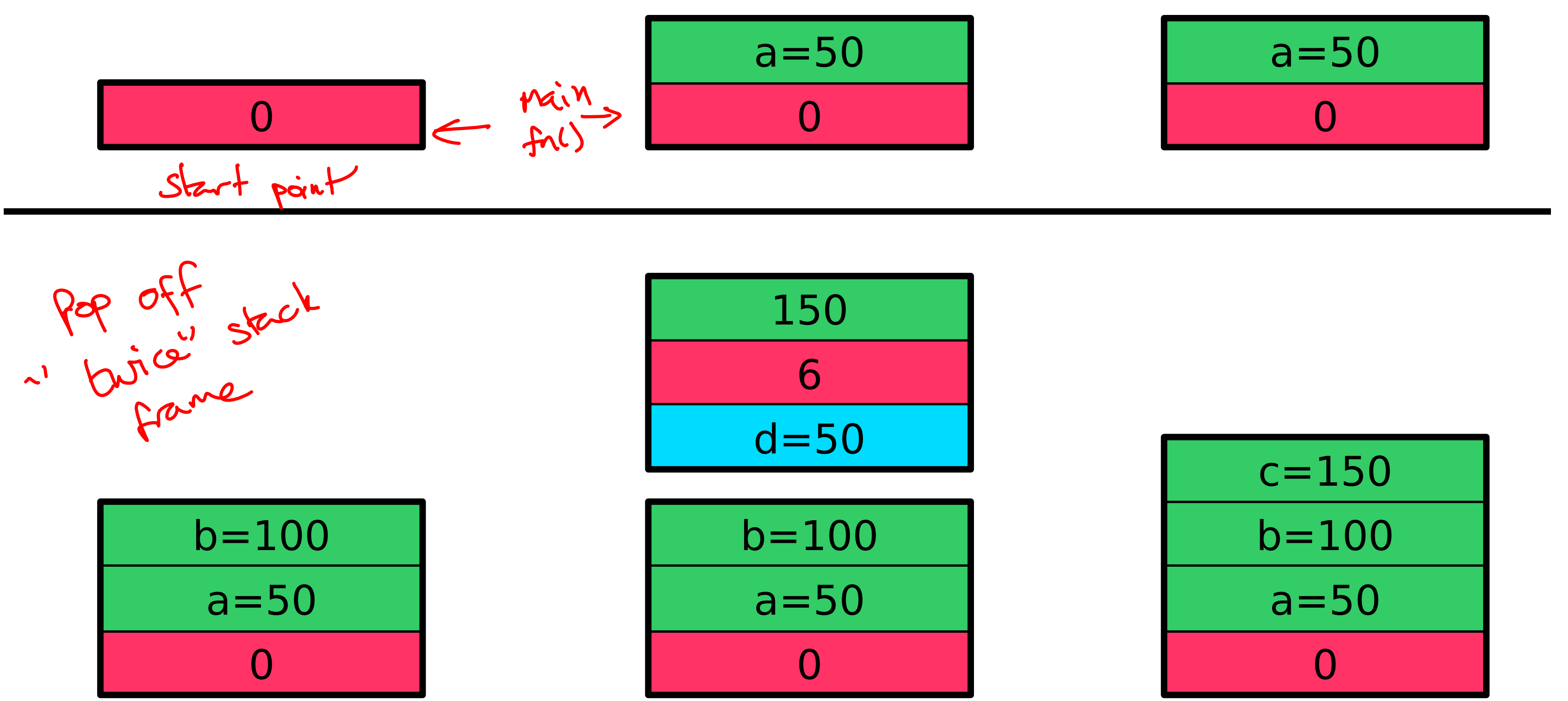
Somehow remember where to return to

The Call Stack



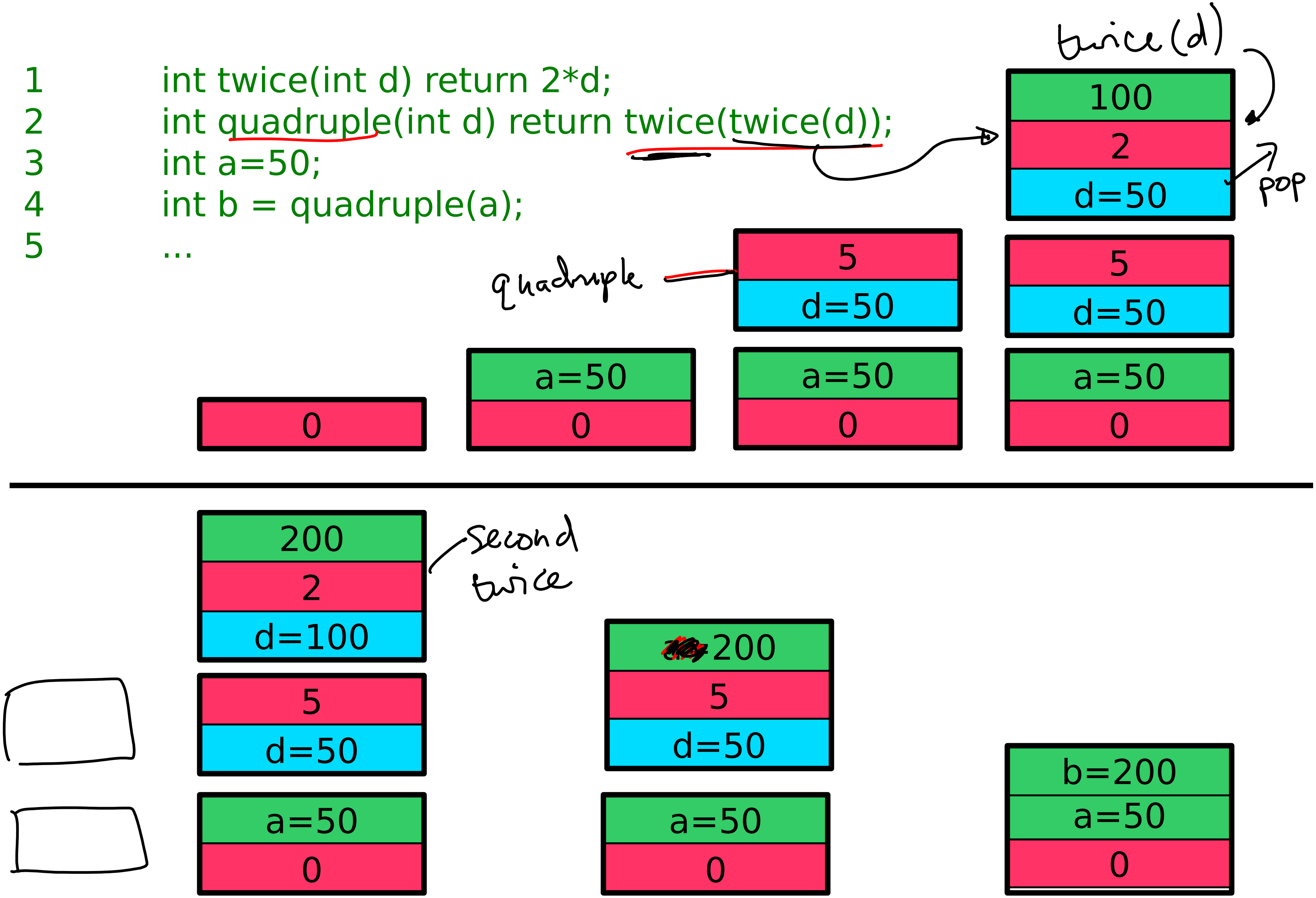
The Call Stack: Example

```
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);
...
```



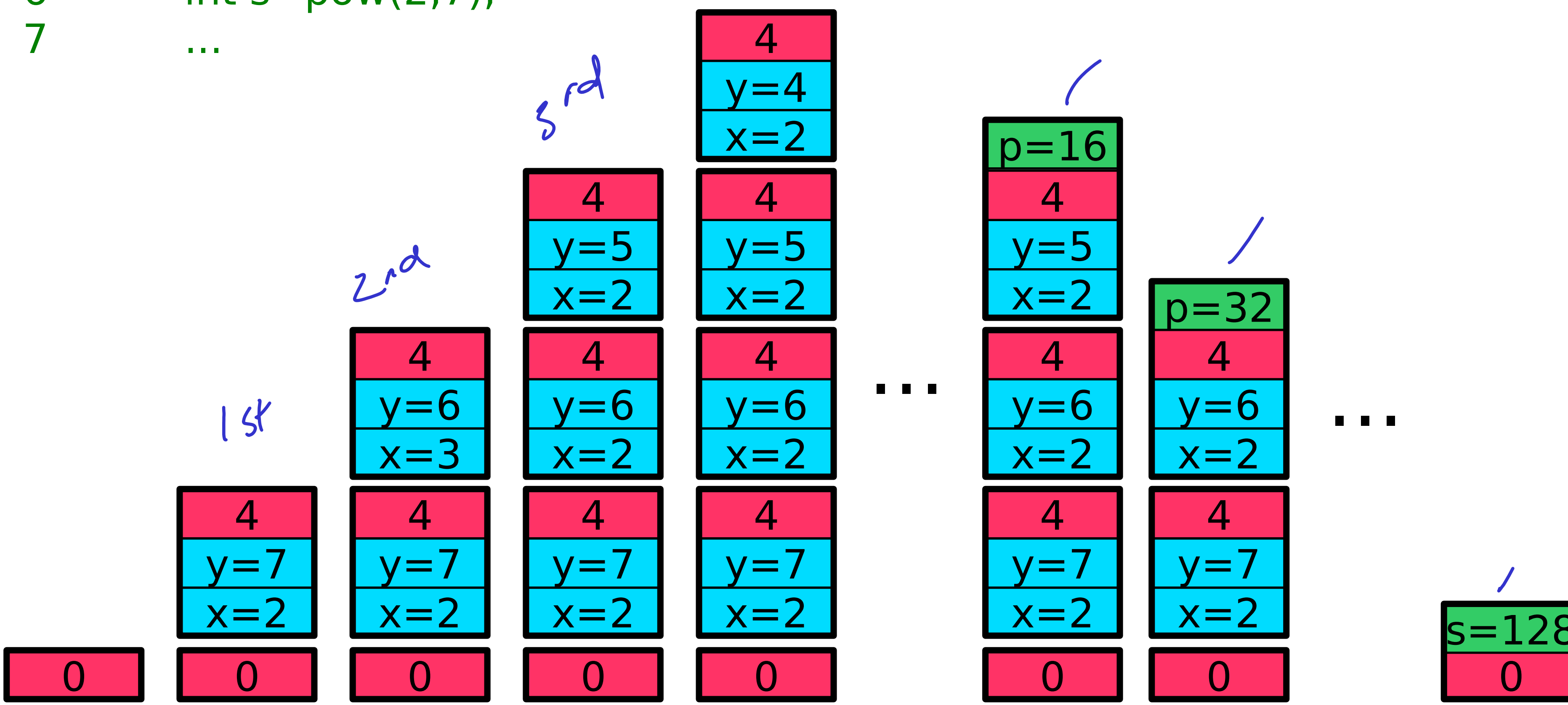
Nested Functions

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



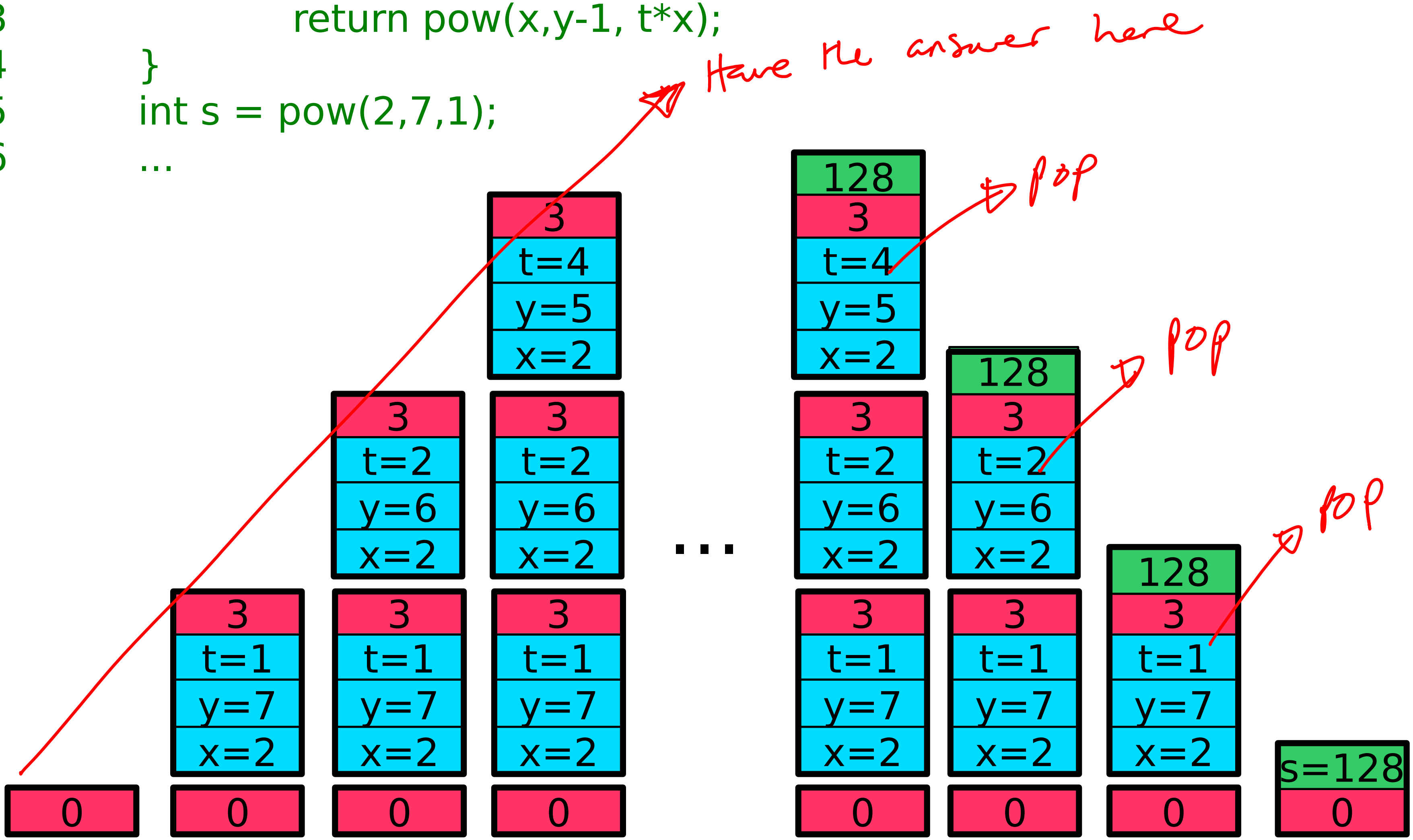
Recursive Functions

```
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);
...
```



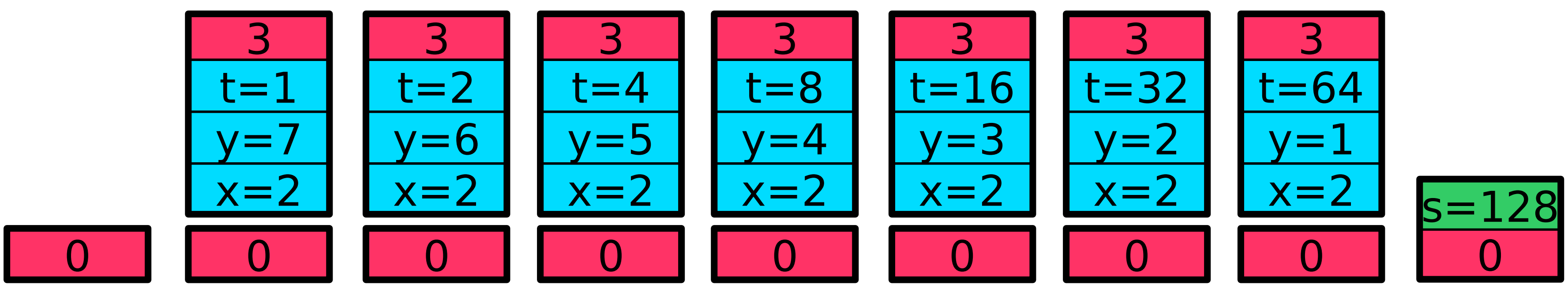
Tail-Recursive Functions I

```
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

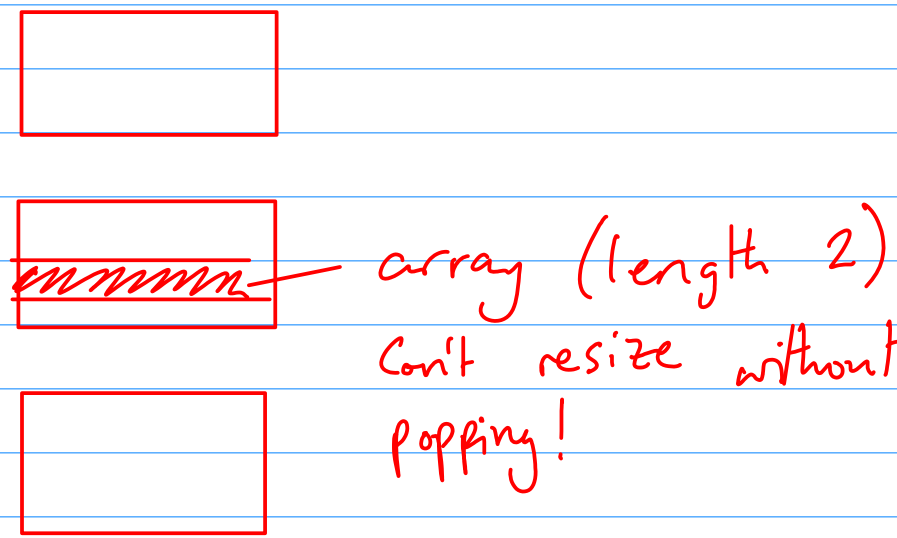
```
int pow (int x, int y, int t) {
        if (y==0) return t;
        return pow(x,y-1, t*x);
}
int s = pow(2,7,1);
...
```



The problem with the stack

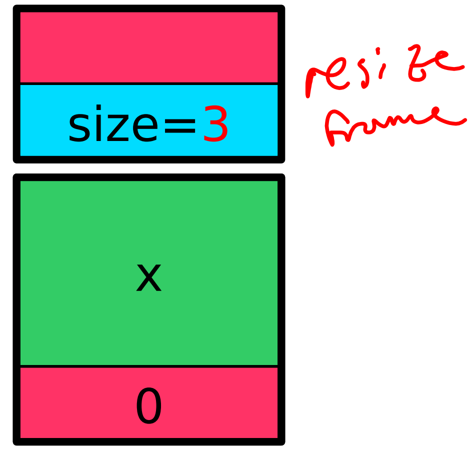
You can only operate on the top.

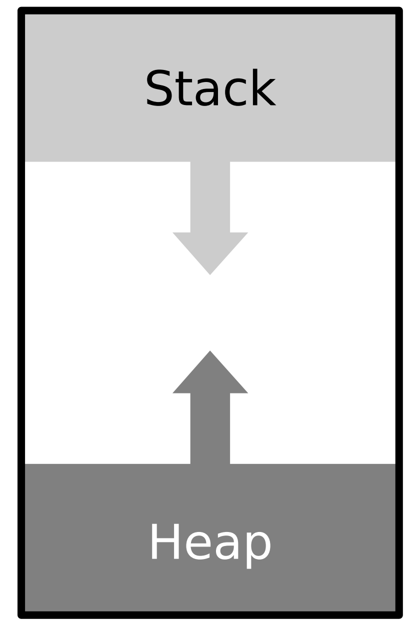
So resizing an entity (e.g. array) elsewhere not poss.

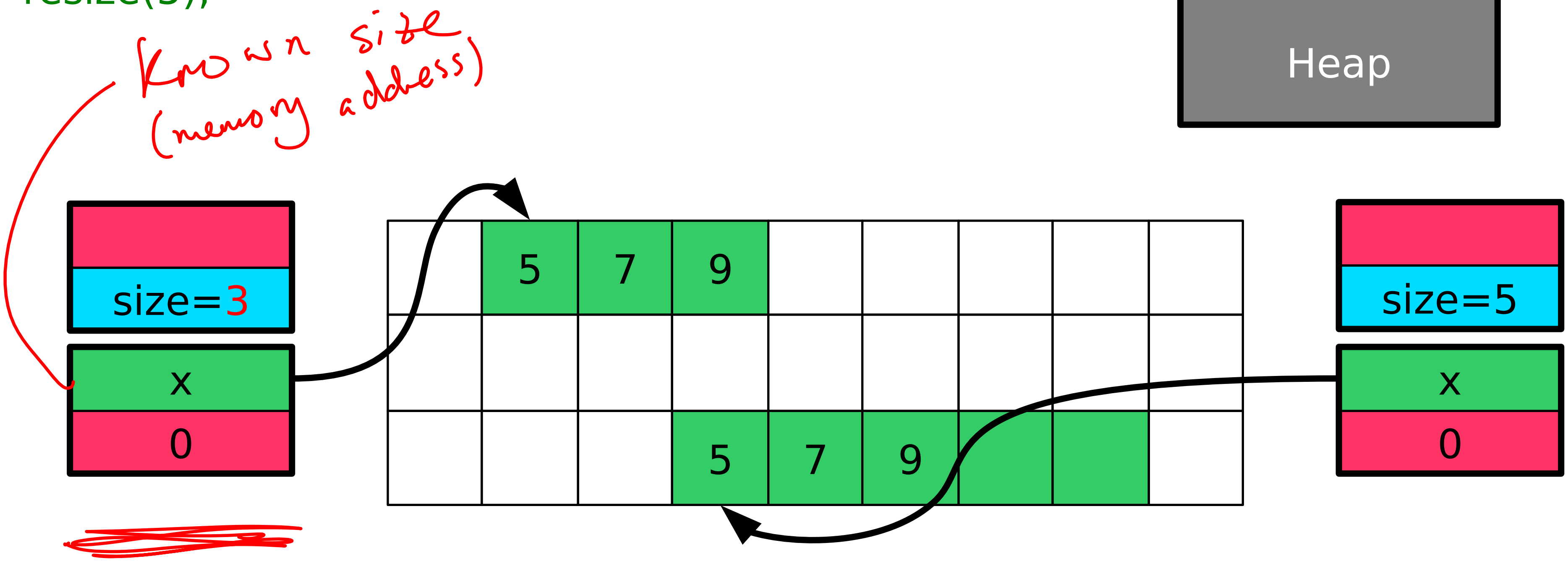


The Heap

```
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);
```







Memory and Pointers

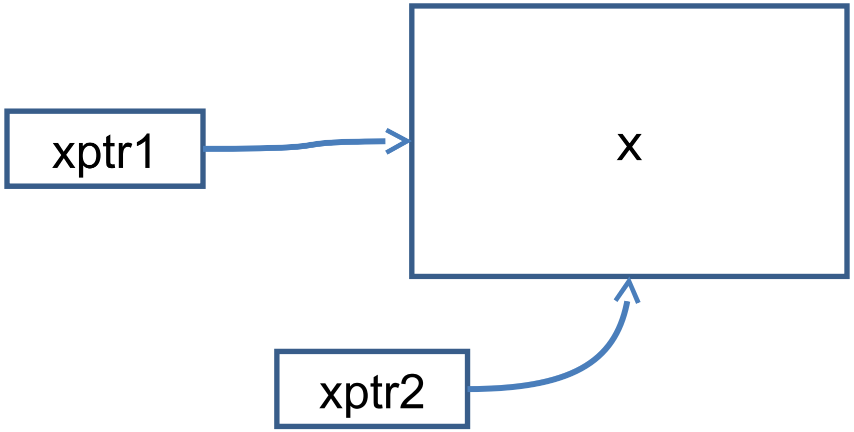
- In reality the compiler stores a mapping from variable name to a specific memory address, along with the type so it knows how to interpret the memory (e.g. *"x is an int so it spans 4 bytes starting at memory address 43526"*).
- Lower level languages often let us work with memory addresses directly. 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' .

Pointers: Box and Arrow Model

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

C++

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

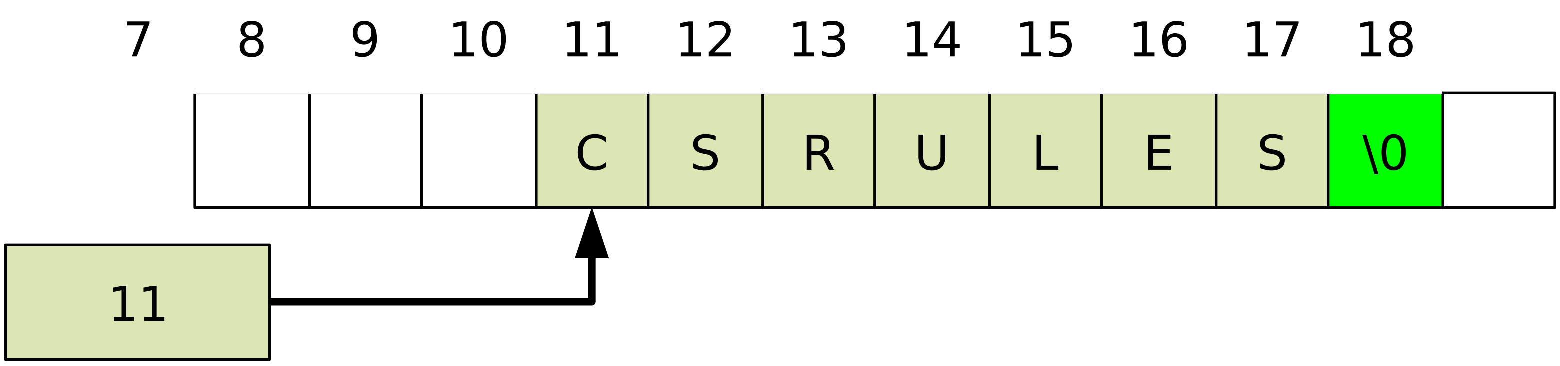


* = pointer (memory address)

& get the memory address

Example: Representing Strings I

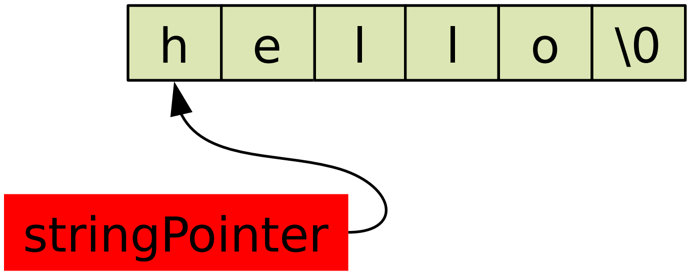
- A single character is fine, but a text string is of variable length – how can we cope with that?
- We simply store the start of the string in memory and require it to finish with a special character (the NULL or terminating character, aka '\0')
- So now we need to be able to store memory addresses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



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

Example: Representing Strings II

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

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

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

letterArray[3]='\0';

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



int x = 0x01020304

Memory address ←

75	76	77	78	79	80	81	82	83	84	85	86
			01	02	03	04					

char * xp = (char *) & x; // get a pointer to x i.e. xp = 78

⇒ xp has value 01

xp	has	value	01
xp+1	"	"	02
xp+2	"	"	03
xp+3	"	"	04
xp+4	"	"	unknown !

References

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

References vs Pointers

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

1. Valid address
2. NULL
3. Random (invalid)

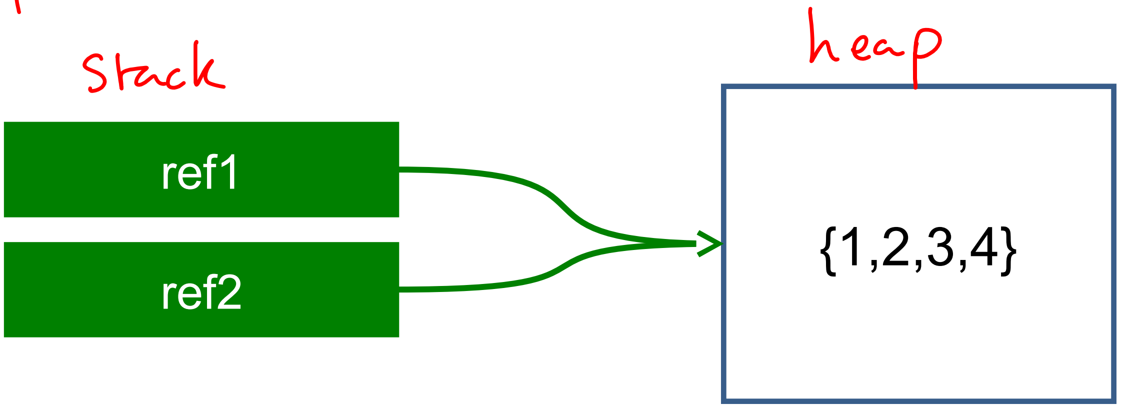
1. Valid address
2. Null

References Example (Java)

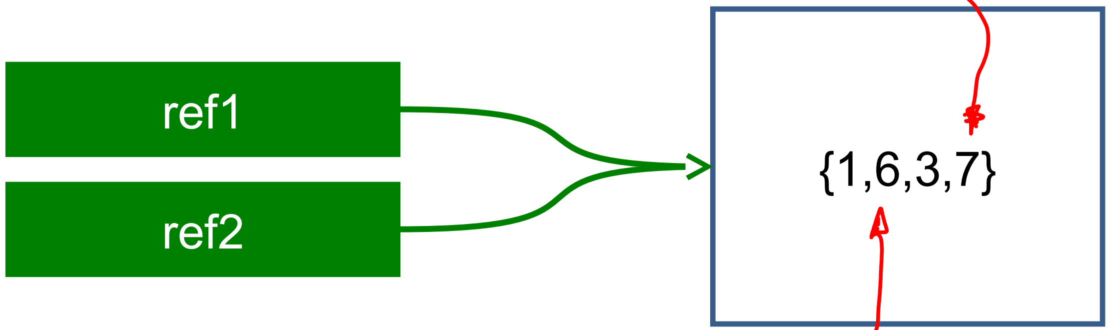
Arrays are handled by reference

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

Allocate on the heap



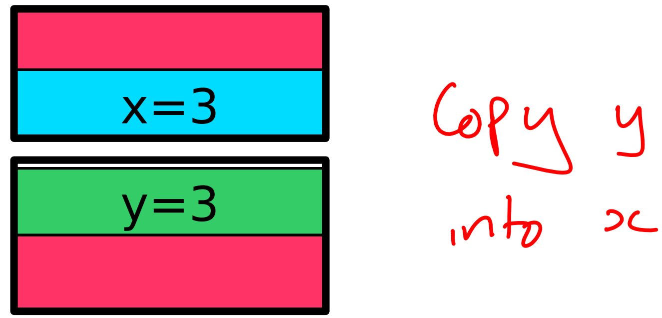

```
ref1[3]=7;
ref2[1]=6;
```



Argument Passing

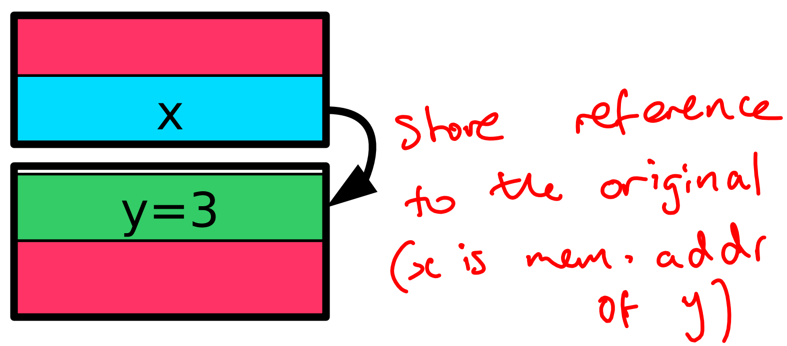
- **Pass-by-value**. Copy the object into a new value in the stack

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



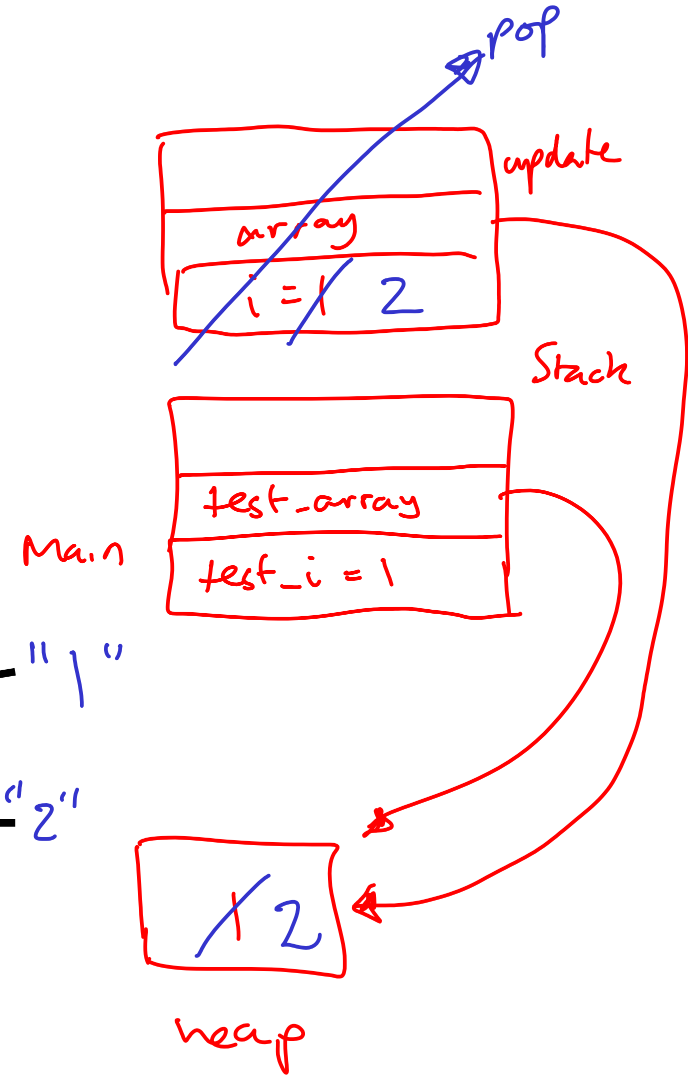
- **Pass-by-reference**. Create a reference to the object and pass that.

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



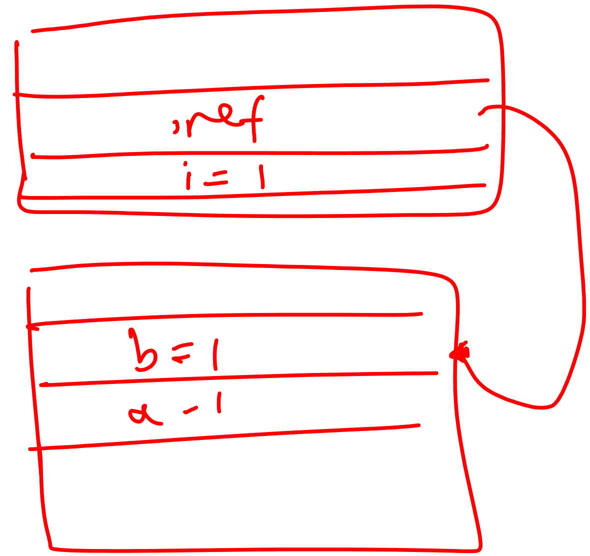
Passing Procedure Arguments In Java

```
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]);
   }

}
```



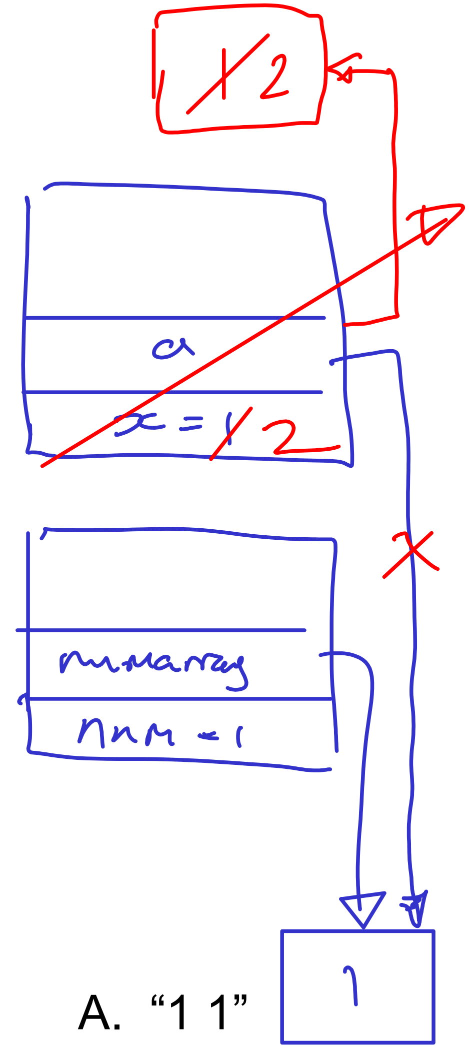
Passing Procedure Arguments In C++

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

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



Check...

```
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]);
}
```



A. “1 1”
B. “1 2”
C. “2 1”
D. “2 2”

Lecture 3:
Creating Classes

What Not to Do

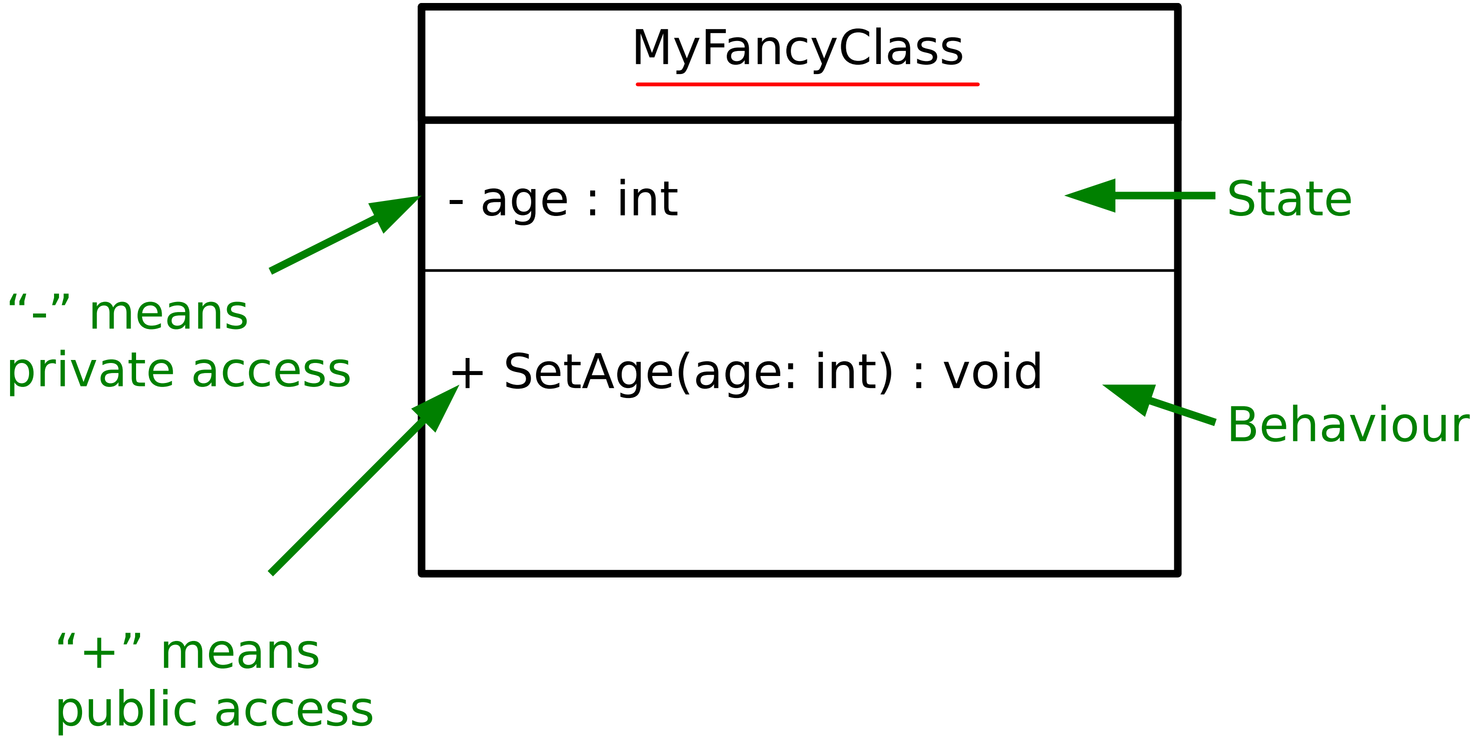
- Your ML has doubtless been one big file where you threw together all the functions and value declarations
- Lots of C programs look like this :-(
- We *could* emulate this in OOP by having one class and throwing everything into it
- We can do (much) better

Identifying Classes

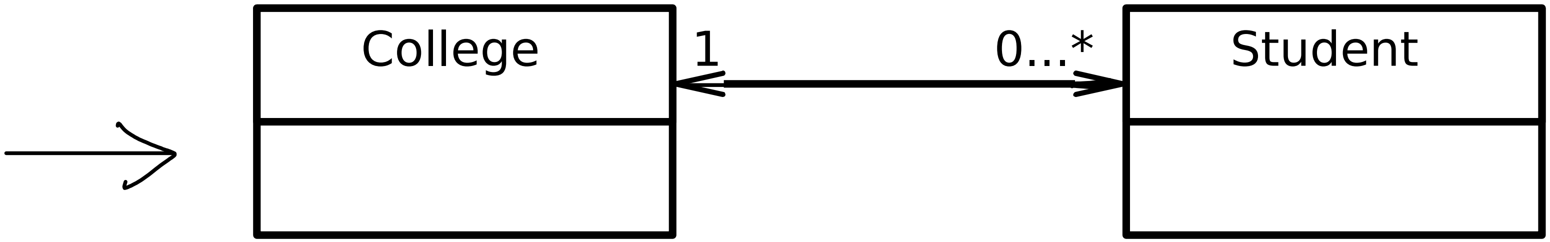
- We want our class to be a grouping of conceptually-related state and behaviour
- One popular way to group is using grammar
  - Noun → Object
  - Verb → Method

“A simulation of the Earth's orbit around the Sun”

UML: Representing a Class Graphically

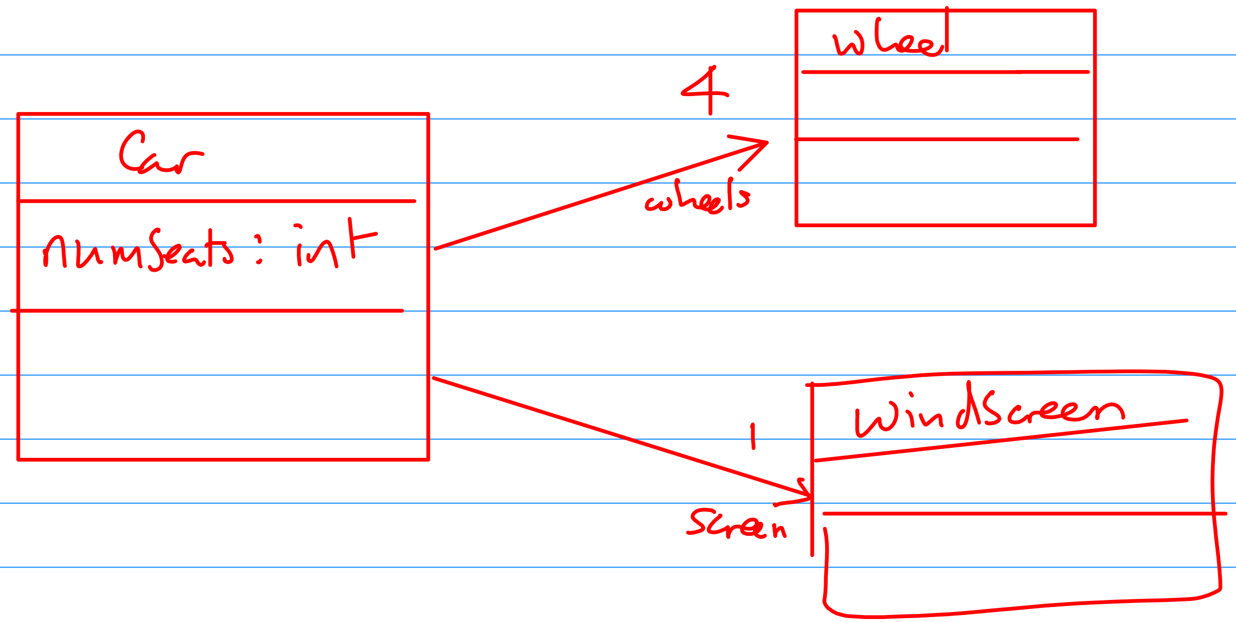


The has-a Association



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

```
public class Car {
    Wheel wheels[] = new Wheel[4];
    WindScreen screen = new WindScreen();
    int numSeats = 5;
}
```



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();
    }

}
```

State

Behaviour

Start

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

Class state

Class behaviour

```
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

Accessing the state and Behavior

```
public class Test {
    int x = 5;
    void setX(int y) {
        x = y;
    }
}
```

```
Test t = new Test();
t.x = 7;
t.setX(3);
```

dot operator

OOP Concepts

- OOP provides the programmer with a number of important concepts:
  - Modularity
  - Code Re-Use
  - Encapsulation
  - Inheritance
  - Polymorphism
- 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.

[Ball example]

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);
}
```

Sanity checking of inputs

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;}
}
```

change →

```
class Location {

   private Vector2D v;

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

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

Access Modifiers

	Everyone	Subclass	Same package (Java)	Same Class
private				X
package (Java)			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

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;}
}
```

Class-Level Data and Functionality I

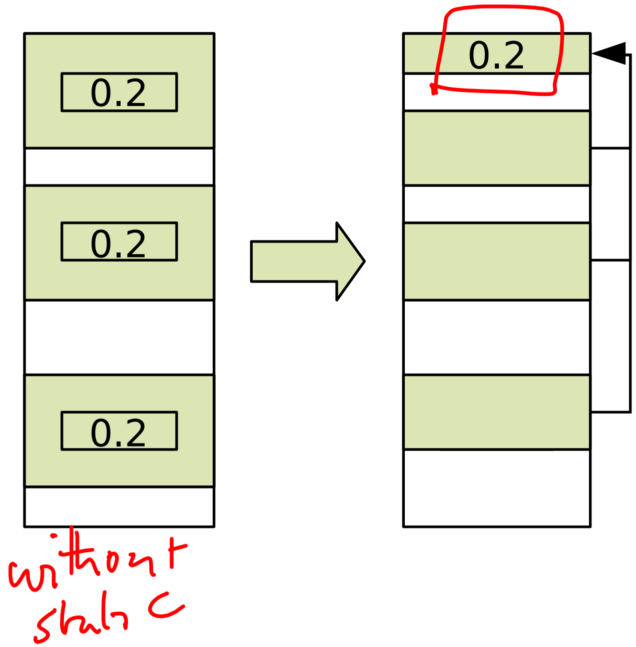
- A **static** field is created only once in the program's execution, despite being declared as part of a class

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

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

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

Class-Level Data and Functionality II



- Auto synchronised across instances
- Space efficient

- 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

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

vs

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

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

```
...
Math.sqrt(9.0);
...
```

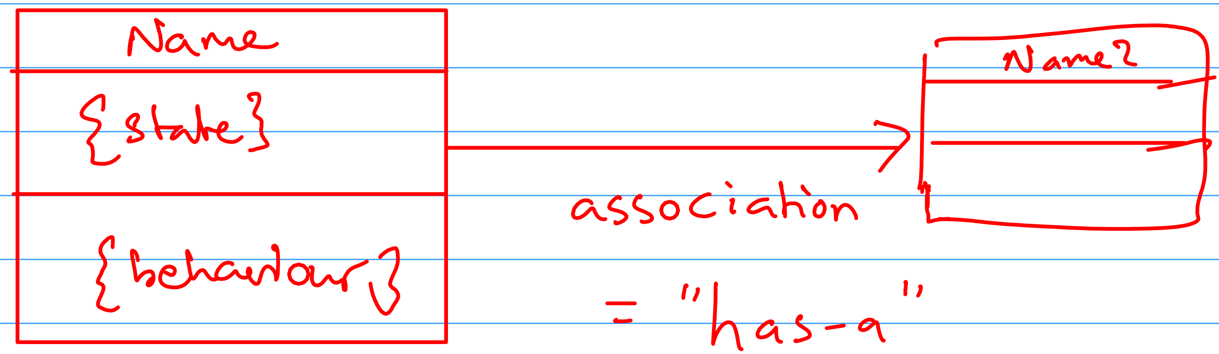
ugly

class name not object name

Recap

- OOP types (classes) are groupings of state and behaviour

- A good class represents a single concept



- We create multiple objects from each class

- Encapsulation is a key concept that encourages us to separate implementation from interface

- Encapsulation allows us to <u>refactor</u> (rewrite) our classes without everyone who uses them changing their code
- Encapsulation allows us to sanity check inputs.
- <u>Static</u> things are those that make sense for the concept but which are the same for every instance, i.e. instance-independent state or behaviour.

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

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

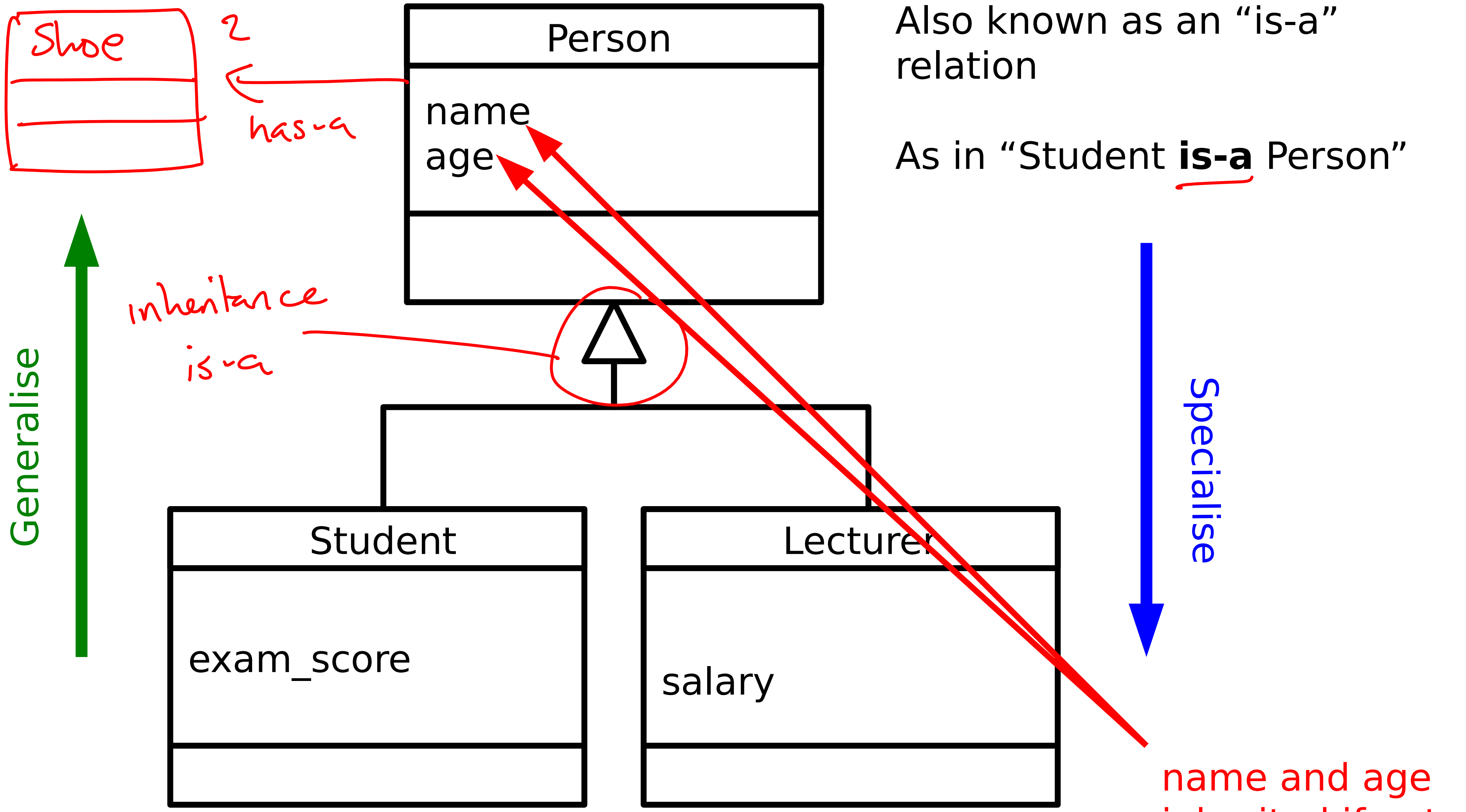
Inheritance II

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

class Student extends Person {
   public int grade;
}

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

inheritance

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

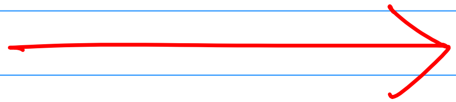
Representing Inheritance Graphically



Also known as an "is-a" relation

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

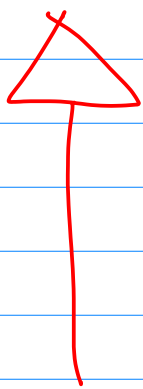
UML Arrows You Need To Know



Association

⇒ "has-a"

as in "X has-a Y"



Inheritance

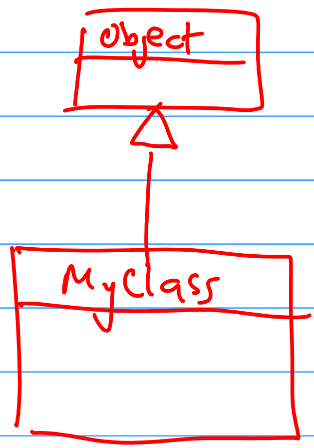
⇒ "is-a"

as in "X is a Y"

Java Oddity: Object

"Magic" class Object

Every class extends Object




```
public class MyClass {

}
```

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
```

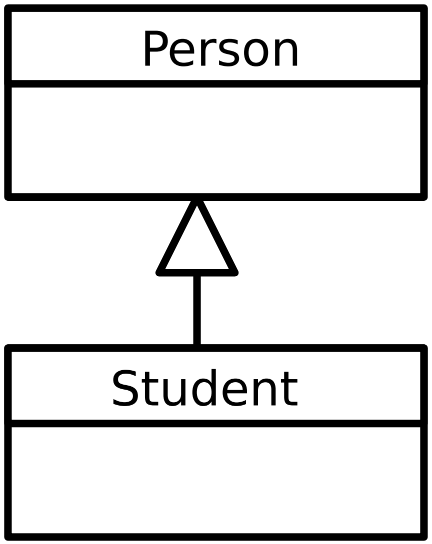
cast

upcast increase precision

downcast lose precision

- 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;
```

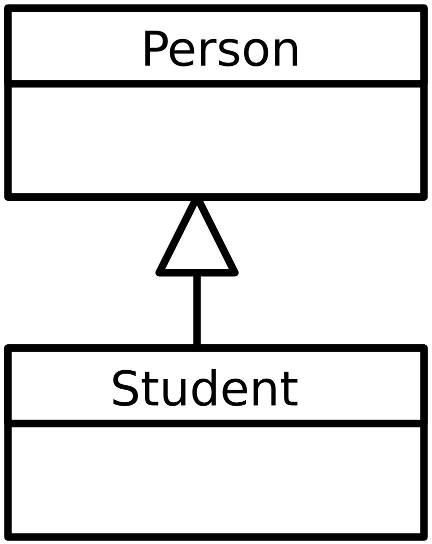
reference to a Person

"Casting" to a Person

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

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

Implicit cast

Narrowing



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


```
Person p = new Person();

Student s = (Student) p;
```

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

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

OK because underlying object
really is a Student

Why is this Useful?

E.g. Making a list of all people

```
LinkedList<Student> ls
    = new ....
```

```
LinkedList<Lecturer> ll
    = new ____
```

Loop over both lists

```
LinkedList<Person> lp
        = new LinkedList<Person>();

lp.add(new Student());
lp.add(new Lecturer());

for(int i=0; i<lp.size(); i++) {
Person p = lp.get(i)

}
```

Fields and Inheritance

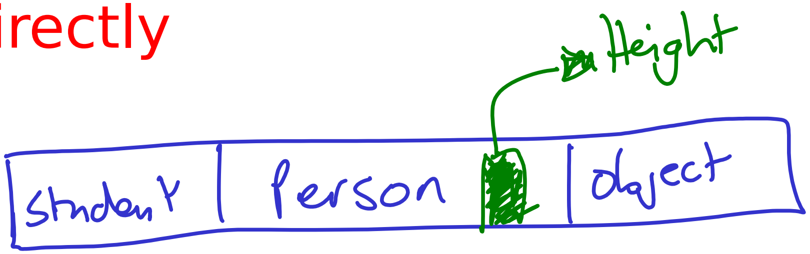
```
class Person {
   public String mName;
   protected int mAge;
   private double mHeight;
}

class Student extends Person {

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

}
```

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

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

Student inherits this but as a **private** variable and so cannot access it directly

won't compile



public - Everything external can access

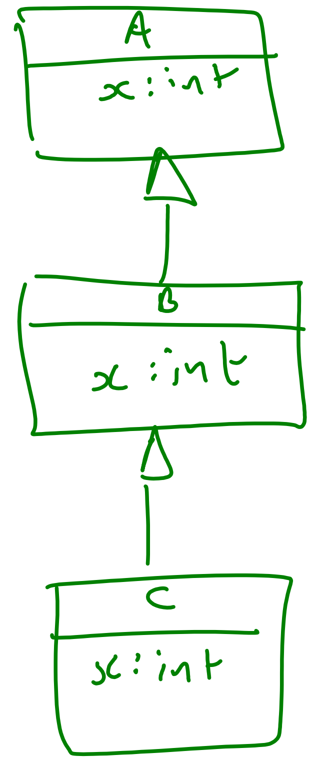
protected - Subclasses can access (+ anything in same package in Java)

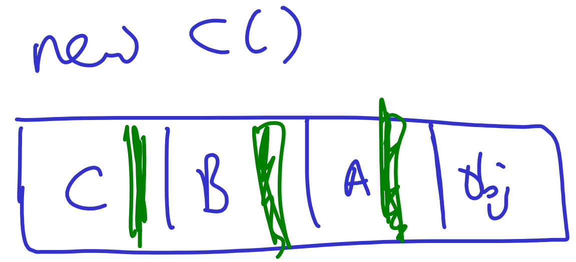
private - Nothing external can access

(package) - Only things in same package

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;
  }
}
```





super.super.super

NO!

this : The current object (ref to)

Super : Get a ref. to the superclass
object

Methods and Inheritance: Overriding

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

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

Person defines a 'default' implementation of dance()

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

Student overrides the default

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

Lecturer just inherits the default implementation and jiggles

Same signature

Annotations

```
@Override public void dance() {

}
```

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();
   }
}
```

No implementation

Cannot create a Person object

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();
}
```

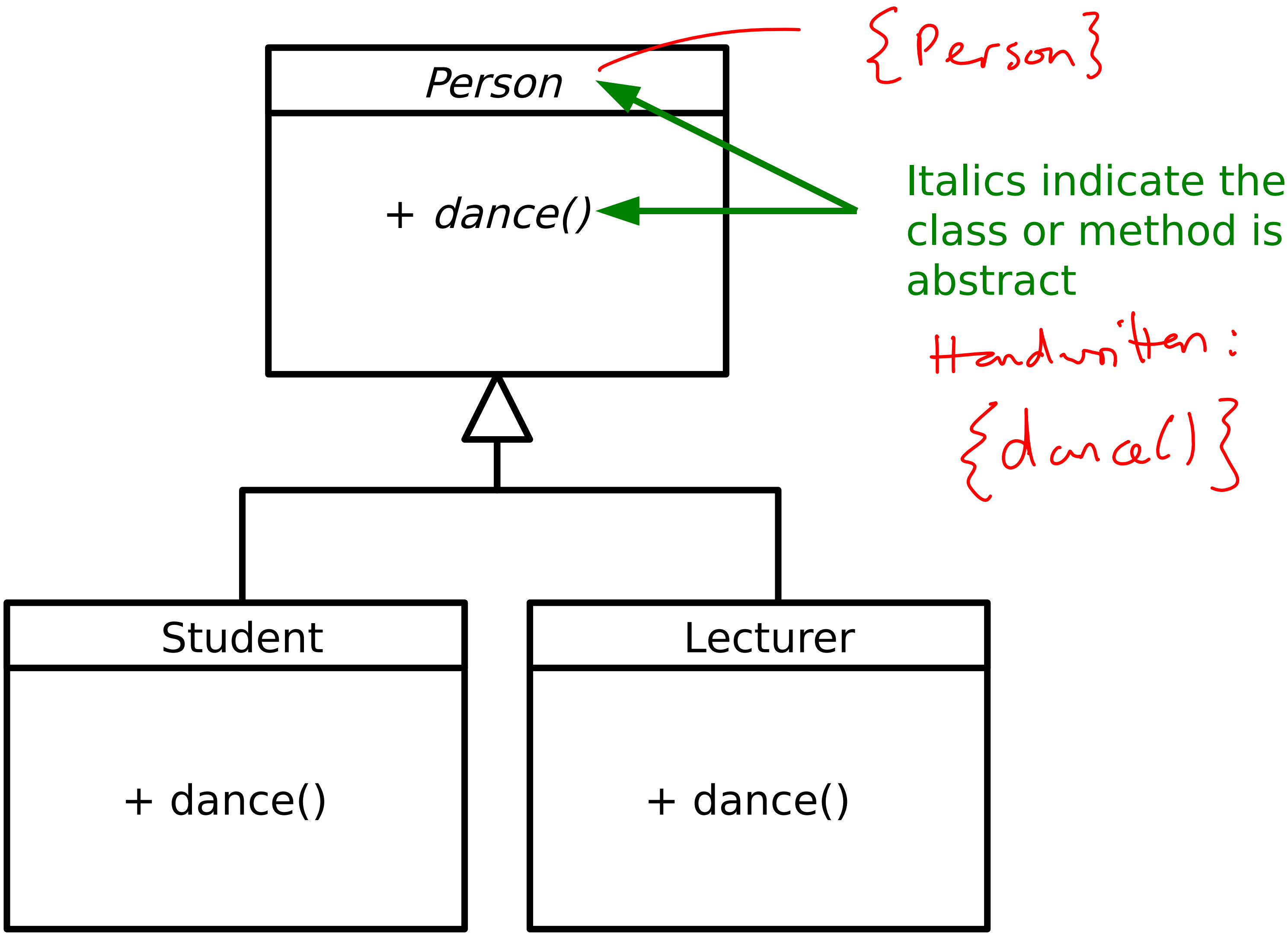
Java

```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



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??

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

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
  - Type errors give compile errors

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

p is a Person
∴ run Person's dance()

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

Why can't the compiler just figure out the true type?

```
Person p = null
if (create_student) {
        p = (Person) new Student();
}
else {
        p = (Person) new TaxPayer();
}

p.dance();
```

Conditional branch (covers the two assignments to p)

← p.dance(): Compiler can't know it's really a Student or TaxPayer

Static Polymorphism You've Seen

ML

```
fun cons a xs = a::xs;
val cons = fn : 'a -> 'a list -> 'a list

cons 1 [2,3,4];
```

↑ Compiler creates a cons for ints

Java Generics

```
public class Stack<T> {
    ...
}

new Stack<Integer>();
```

↑ Compiler fills in the type

Polymorphic Concepts II

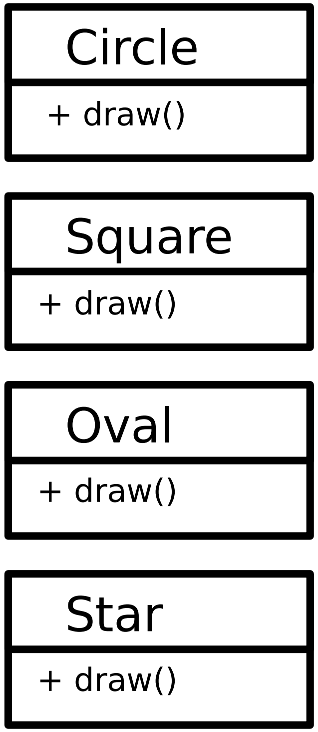
- **Dynamic** polymorphism
  - Run the method in the child
  - Must be done at <u>run-time</u> since that's when we know the child's type
  - Type errors cause run-time faults (crashes!)

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

Looks in memory
=> Sees a Student
=> runs Student's dance()

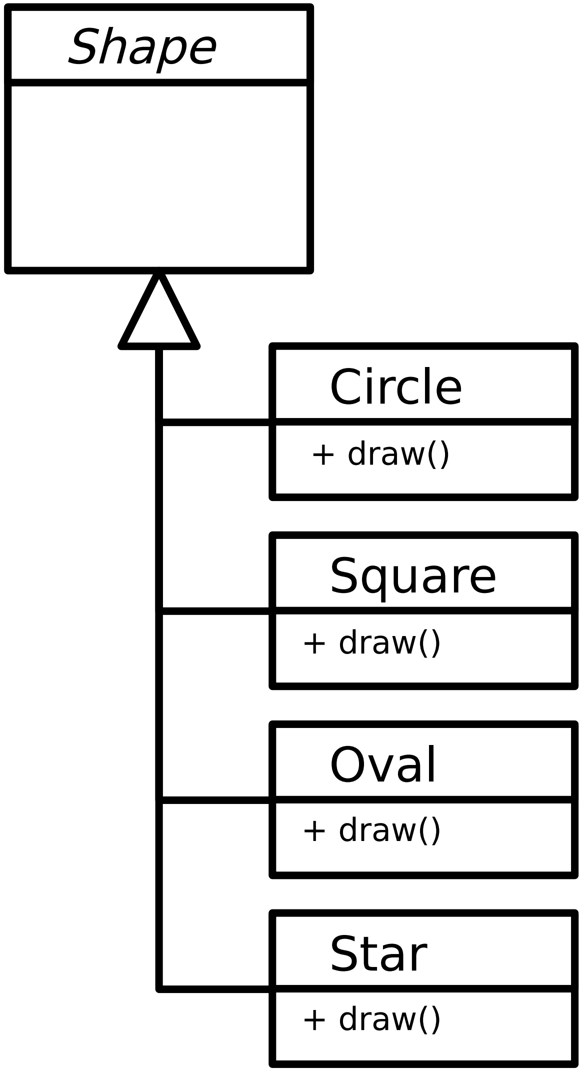
- Compiler looks in memory and finds that the object is really a Student
- So p.dance() runs the dance() action in <u>Student</u>

The Canonical Example I



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

The Canonical Example II

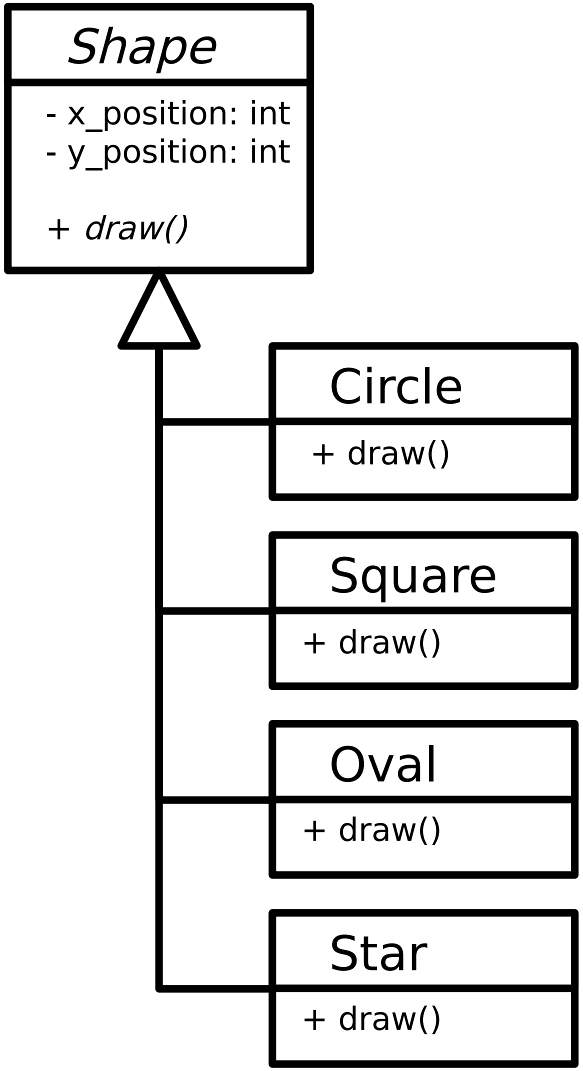


- **Option 2**
  - Keep a single list of Shape references
  - Figure out what each object really is, narrow the reference and then draw()

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

  - What if we want to add a new shape?

The Canonical Example III



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

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

"Polymorphic check"

  - What if we want to add a new shape?

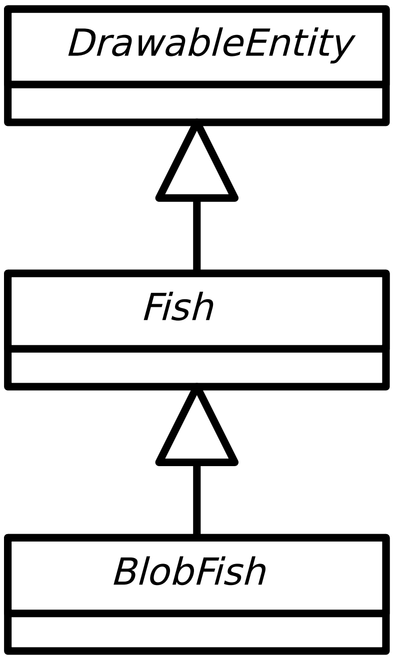
Implementations

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

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

Harder Problems

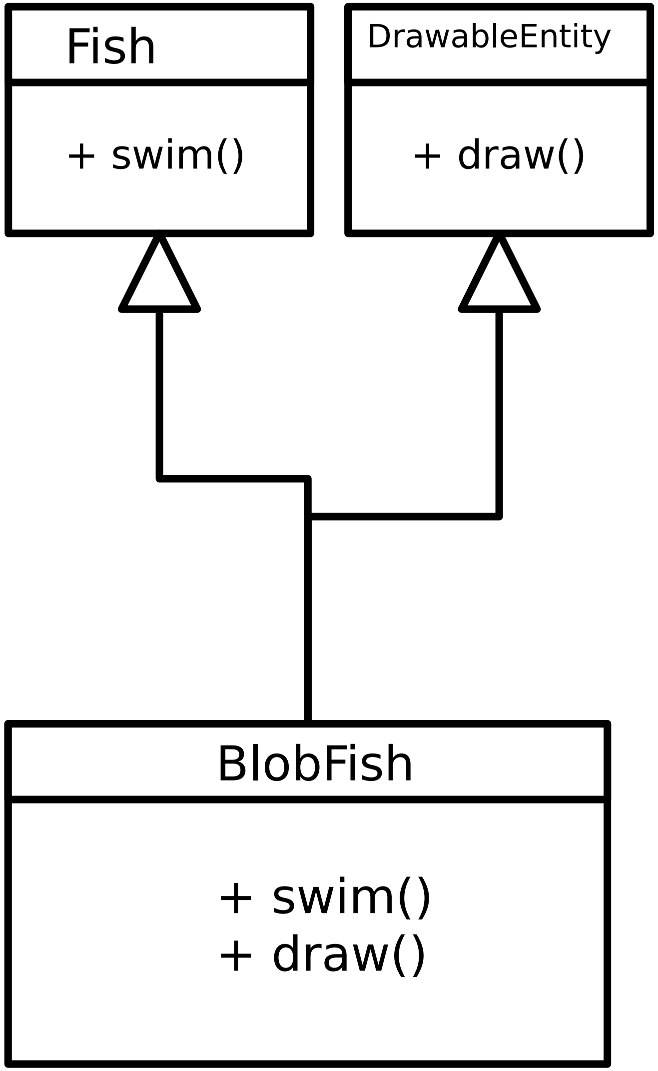
- Given a class Fish and a class DrawableEntity, how do we make a BlobFish class that is a drawable fish?



X Dependency between two independent concepts

X Conceptually wrong

Multiple Inheritance

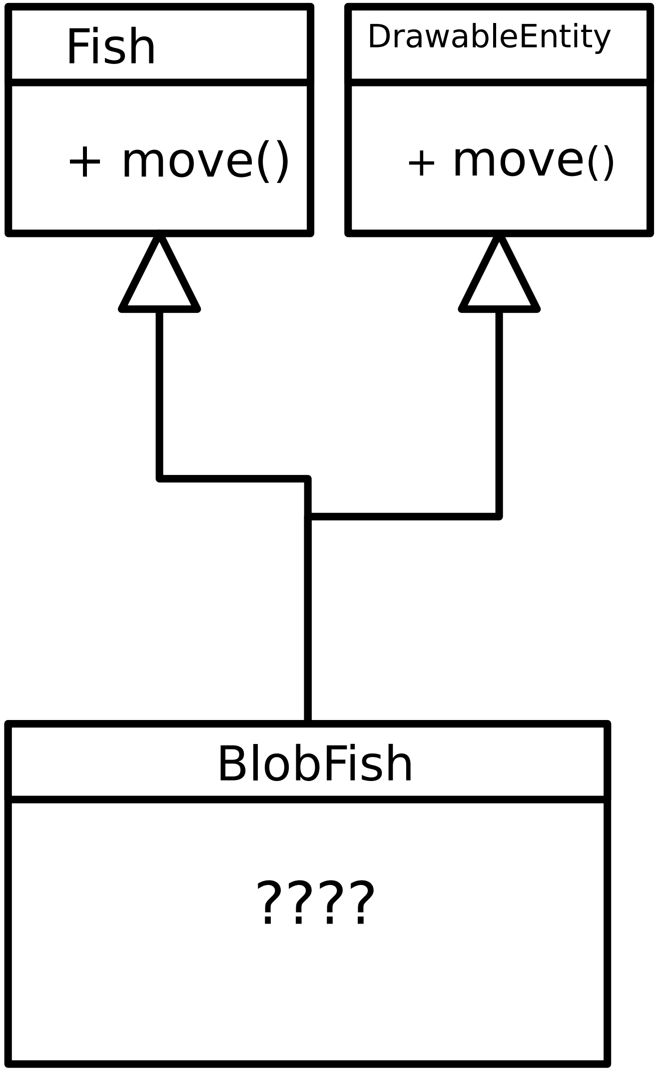


- If we multiple inherit, we capture the concept we want
- BlobFish inherits from both and is-a Fish and is-a DrawableEntity
- C++:

```
class Fish {…}
class DrawableEntity {…}

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

- But…

Multiple Inheritance Problems

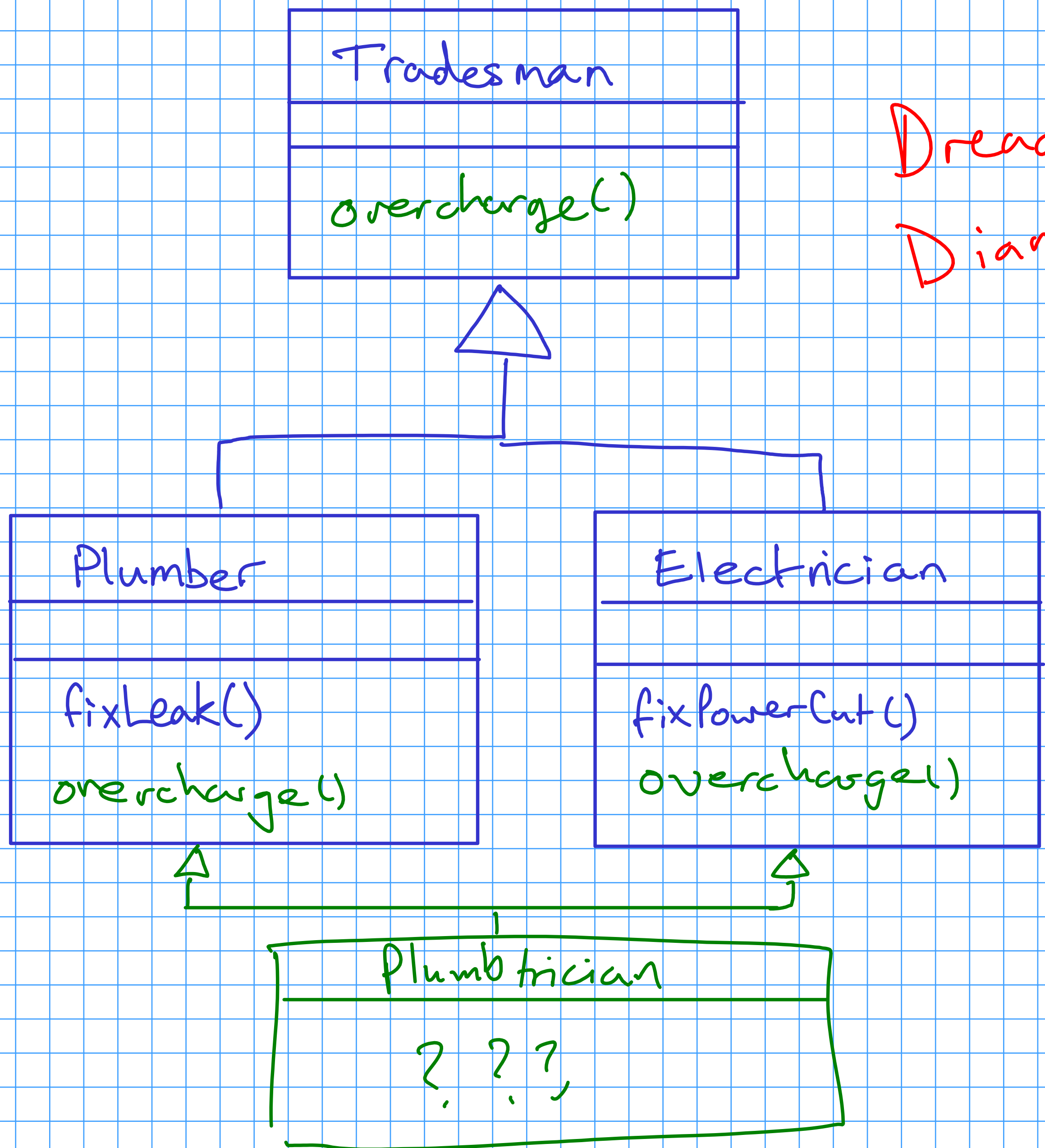


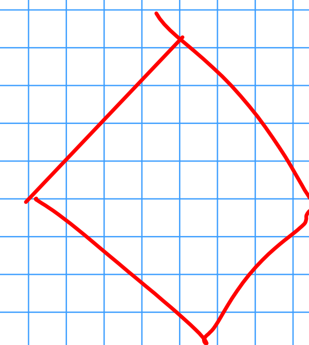
- What happens here? Which of the move() methods is inherited?
- Have to add some grammar to make it explicit
- C++:

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

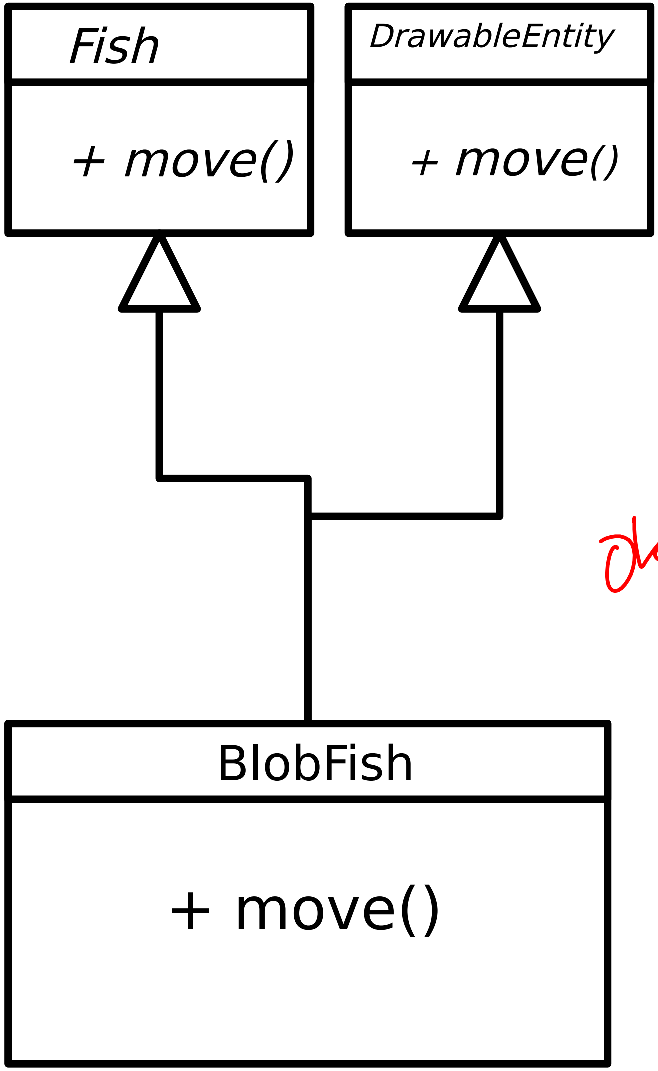
- Yuk.

Dreaded Diamond





Fixing with Abstraction



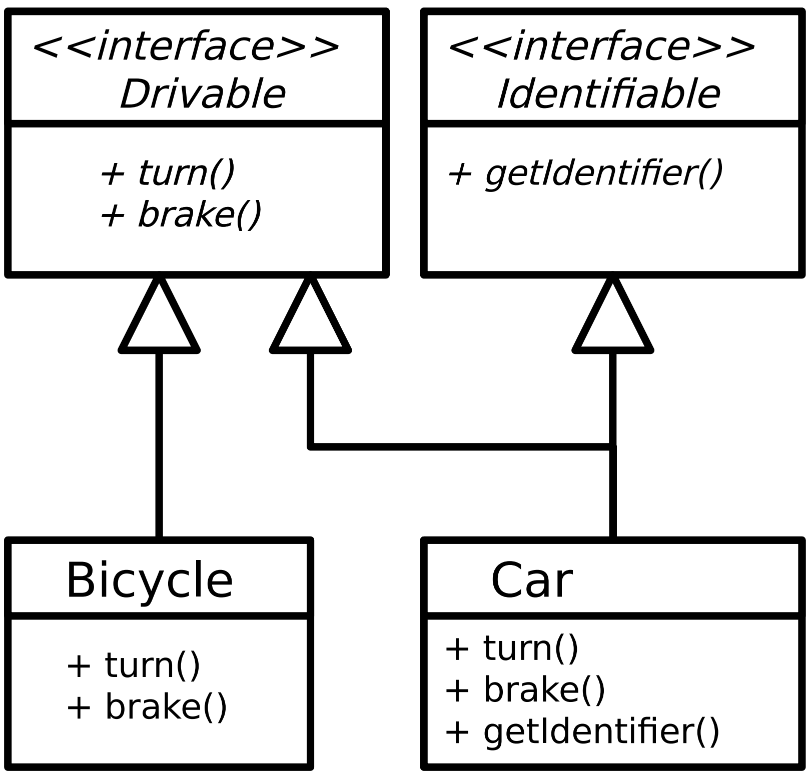
- Actually, this problem goes away if one or more of the conflicting methods is abstract

ok ⇒ Must have a move() method

⇒ No definition of move()

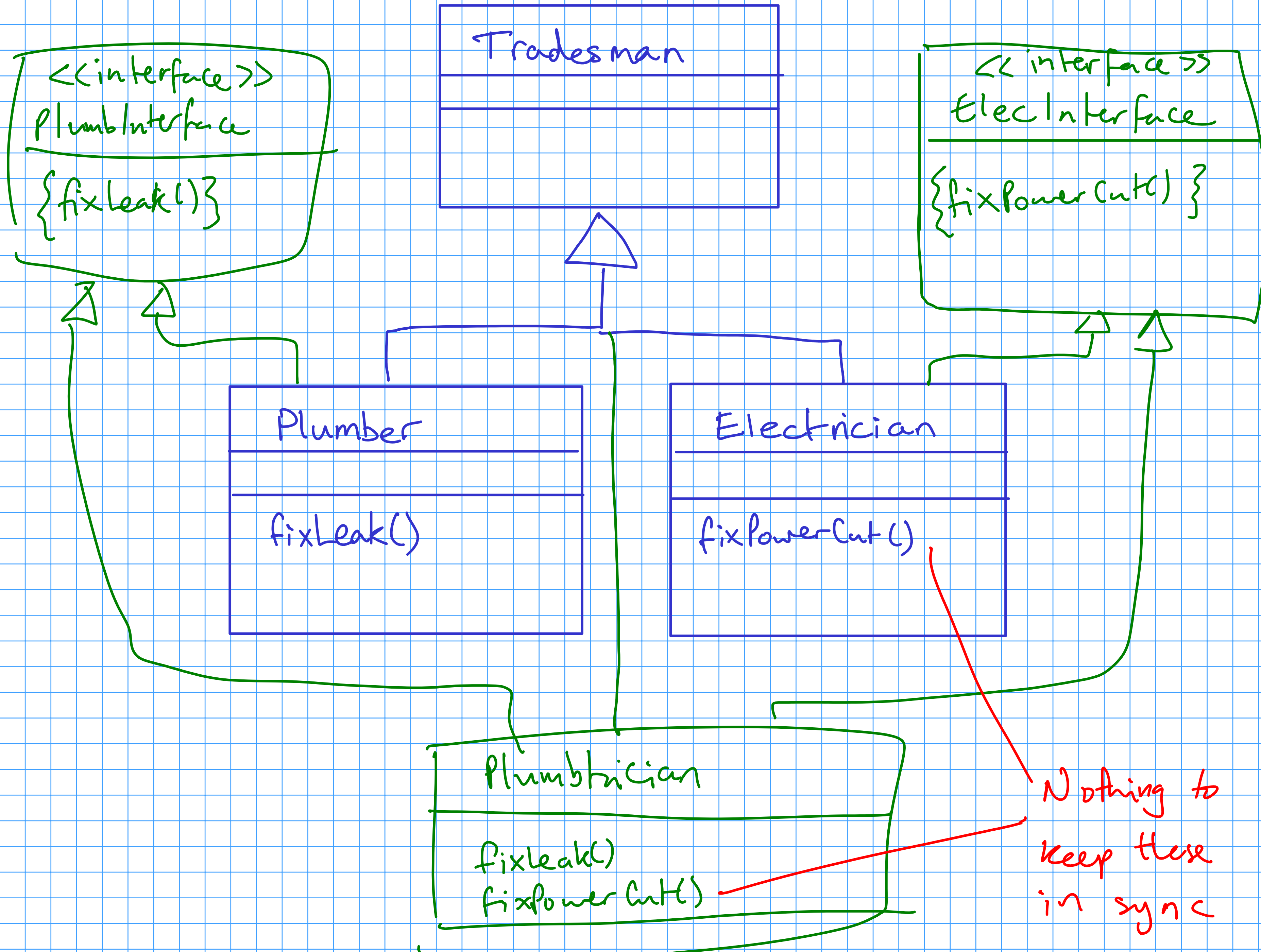
Java's Take on it: Interfaces

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

- extend a class
- implement interfaces



Tradesman
<<interface>>
PlumbInterface
{fixLeak()}
<< interface >>
ElecInterface
{fixPowerCut()}
Plumber
fixLeak()
Electrician
fixPowerCut()
Plumbtrician
fixleak()
fixPowerCut()
Nothing to keep these in sync

Interfaces (Java only)

- Only abstract methods
- Not required to write "abstract" (implicit)
- Only static AND final state

  ⇒ Constants
- Allows a limited (safer) form of multiple inheritance

```
public interface Car{
    public void drive();
}
```

But...

Lecture 6:
Lifecycle of an Object

Constructors

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

function

- 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

Constructor Examples

Java

```
public class Person {
  private String mName;

  // Constructor
  public Person(String name) {
     mName=name;
  }

  public static void main(
        String[] args) {
    Person p =
        new Person("Bob");
  }

}
```

Same name

No return type

C++

```
class Person {
  private:
    std::string mName;

  public:
    Person(std::string &name){
       mName=name;
    }
};


int main (int argc,
          char ** argv) {
  Person p ("Bob");
}
```

Initialiser Blocks

```
public class Test {
    private int x = 7;
    {
      x = 8;
    }
    public Test() {
        x = 9;
    }
```

final Revisited

final int mX = ...

← Sets the value of mX based on either constructor or the init. block

final LinkedList<Integer> mList = ...

← Prevent me from re-setting the reference but the <u>object</u> remains mutable

public final class X {...}

← Cannot extend the class

Default Constructor

auto generates

public Person() {

}

```
public class Person {
   private String mName;

   public static void main(String[] args) {
     Person p = new Person();
   }

}
```

- If you specify no constructor at all, Java fills in an empty one for you
- Here it creates Person() for us
- The default constructor takes no arguments (since it wouldn't know what to do with them!)

Multiple Constructors

```
public class Student {
   private String mName;
   private int mScore;

   public Student(String s) {
     mName=s;
     mScore=0;
   }

   public Student(String s, int sc) {
      mName=s;
      mScore=sc;
   }

   public static void main(String[] args) {
    Student s1 = new Student("Bob");
    Student s2 = new Student("Bob",55);
   }
 }
```

- You can specify as many constructors as you like.
- Each constructor must have a different signature (argument list)

Calling One Constructor From Another

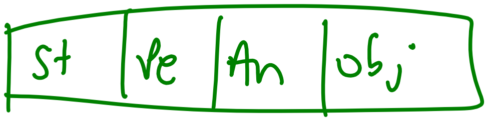
```
public Student (String name, int score) {
    mName = name;
    mScore = score;
}

public Student (String name) {
    this(name, 0);
}
```

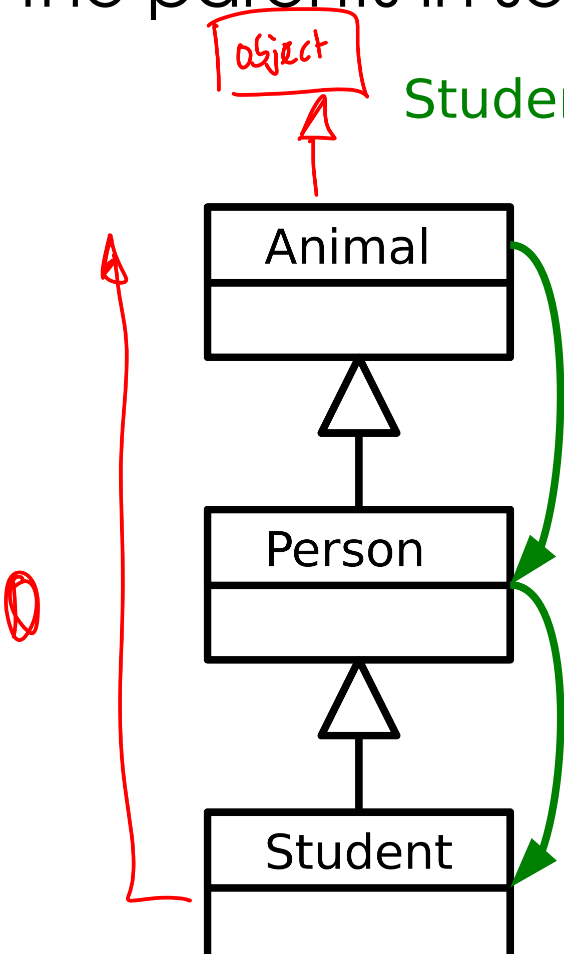
This must be the first line of your constructor

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();





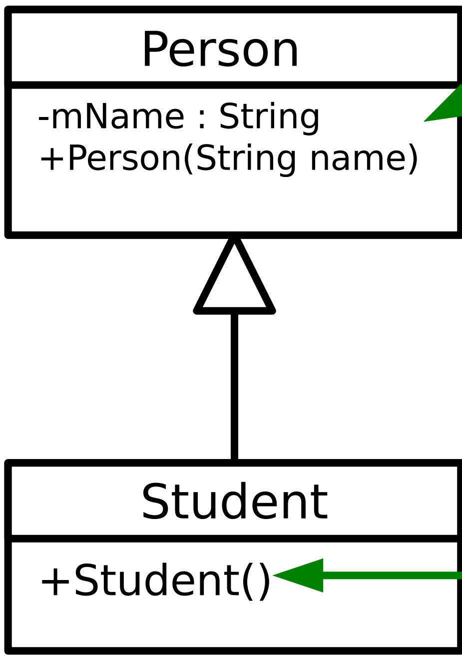
1. Call Animal()

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");
}
```

1 up the tree

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;
}
```

destructor

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

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

  // Use object here
  ...

  // Destruct the object
  delete f;

}
```

Cleaning Up

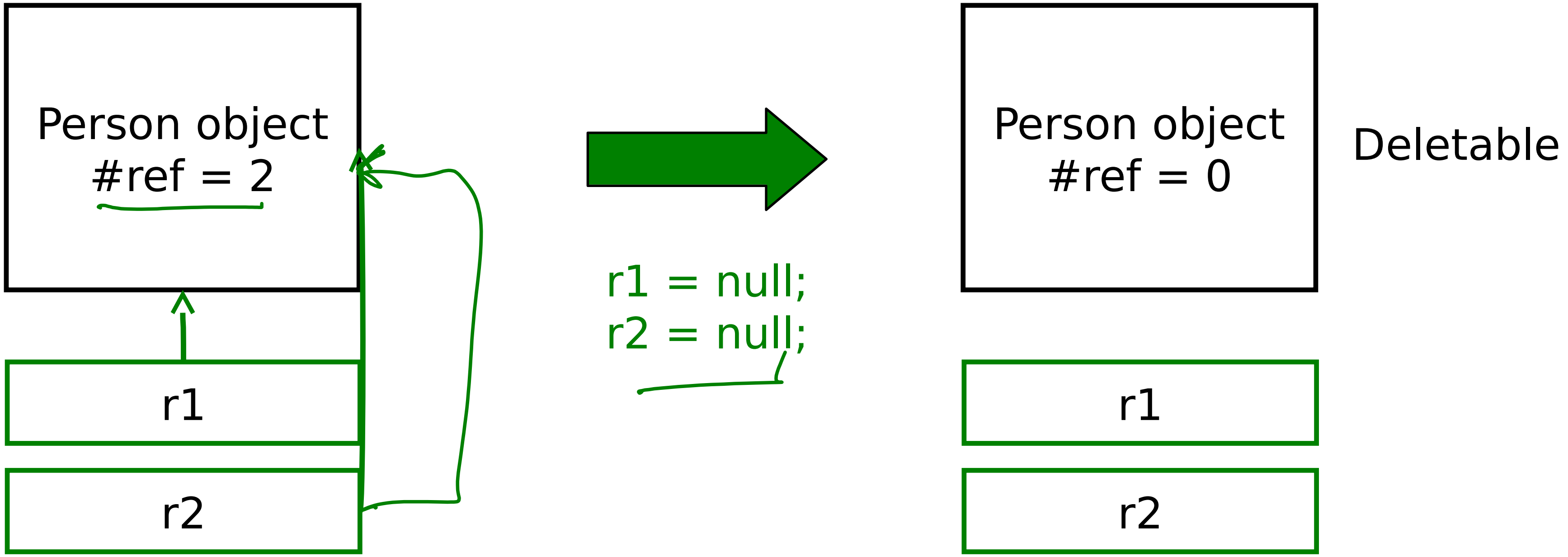
- A typical program creates lots of objects, not all of which need to stick around all the time

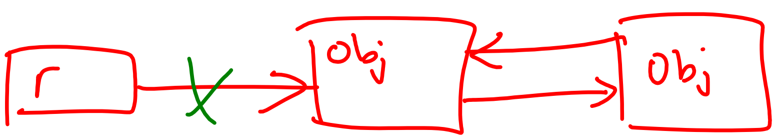
- **Approach 1:**
  - Allow the programmer to specify when objects should be deleted from memory
  - Lots of control, but what if they forget to delete an object?
    - A "memory leak"

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

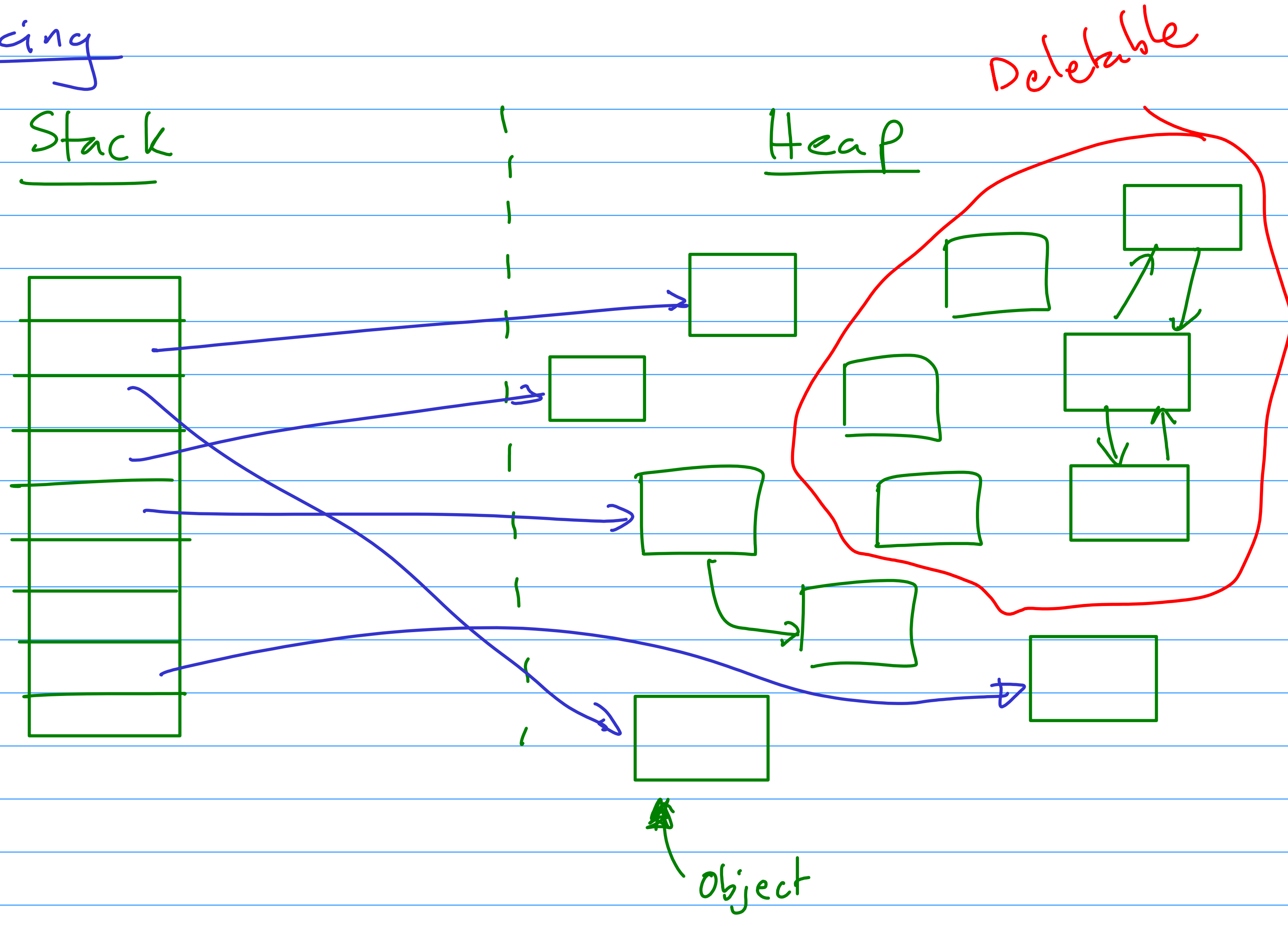
Cleaning Up (Java) I

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





Tracing



Cleaning Up (Java) II

- Actual deletion occurs through a **garbage collector**
  - A separate process that periodically scans the objects in memory for any with a reference count of zero, which it then deletes.
  - 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
    - Gives noticeable pauses to your application while it runs.
    - But minimises memory leaks (it does not prevent them…)

Cleaning Up (Java) III

- One problem with GC is we have no idea *when* an object will actually be deleted. The GC may even decide to defer the deletion until a future run.
- This causes issues for destructors – it might be ages before a resource is closed and available again!
- Therefore Java doesn't have destructors
- It does have **finalizers** that gets run when the GC deletes an object
    - BUT there's no guarantee an object will ever get garbage collected in Java...
    - Garbage Collection != Destruction

Lecture 7:
Error Handling

Return Codes

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

Error code

```
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!!");
```

Can't pass back

- 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

C++

Pass by reference

```
int divide(double n, double d, double & result){
    if (d == 0.0) return -1;
    else {
        result = n/d;
        return 0;
    }
}
```

```
double r = 0.0
divide(a, b, r)
```

Could also try to signal an error through magic values

```
double divide (double n, double d) {
    if (d == 0) return Double.MAX_VALUE;
    }
}
```

```
double d = divide(a,b);
       d = divide(d,c);
```

goes to

```
double r = 0.0;
if (divide(a,b,r) <0){
    // handle error
}
if (divide(r,c,r)<0){
    // handle error
}
```

1. Ignore the return values
2. Forget

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;
}
```

```
public Vector2D pos = null;

Vector2D norm (Vector2D v) {
    if (v.getX() == 0.0 && v.getY() == 0.0) {
        return null;
    }
    else {
        double mag = Math.sqrt (v.getX() * v.getX() +
                                v.getY() * v.getY())
        return new Vector2D (v.getX()/mag,
                             v.getY()/mag);
}
```

return null to indicate error or absence

Non-Exam

Optional<T>

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
}
```

"Try block"

"catch block"

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);
```

This is skipped

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);
    }
}
```

- 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)

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
}
```

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
}
```

Matches in order

Types of Error

- Defects – Bug
  - Not handle-able

- Invalid conditions out of our immediate control
  - These are potentially fixable as we run

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

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();
}
```

Evil I: Exceptions for Flow Control

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

No test

```
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) {
}
```

- ...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

```
public void func throws Exception {

}
```

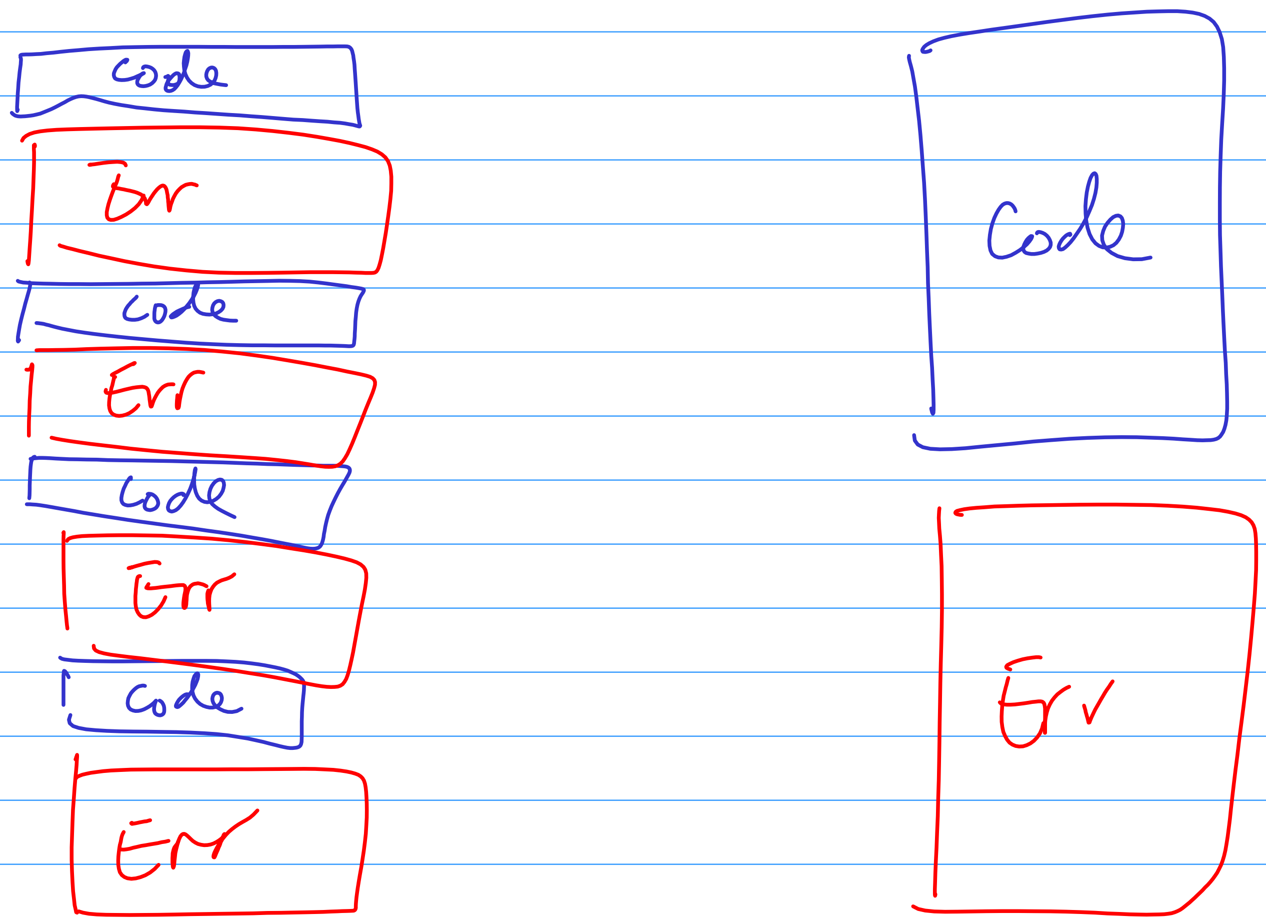
Advantages of Exceptions

- 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

checked

1. Separate Error Handling and Regular Code

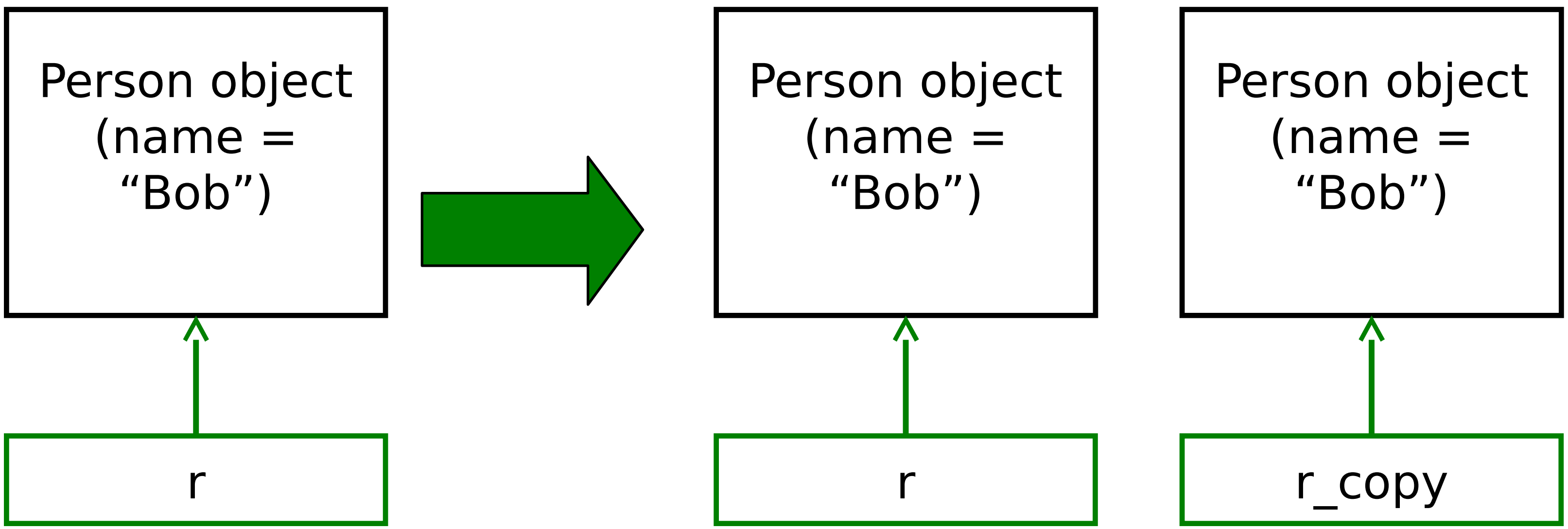
Return codes | Exceptions



Lecture 8:
Copying Objects

Cloning I

- Sometimes we really do want to copy an object



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

Person p = new Person();

Person copy = p;

Copy Constructors

1. Create a constructor that takes a ref. to an object to be copied.
2. You call the copy cons. of the parent (using super)
3. Copy the state in your class
    - → Copy shallow
    - → Copy deep

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)

The Java clone() Recipe

1. Implement Cloneable
2. Override public Object clone();
3. Call super.clone() on the first line
4. Deep copy any referenced objects as appropriate

The Java clone() Recipe

1. Make your class implement Cloneable
2. Override public Object clone()
3. Must must must call super.clone() on the first line of clone()
4. Copy state
   → copy deep

The clone() logic

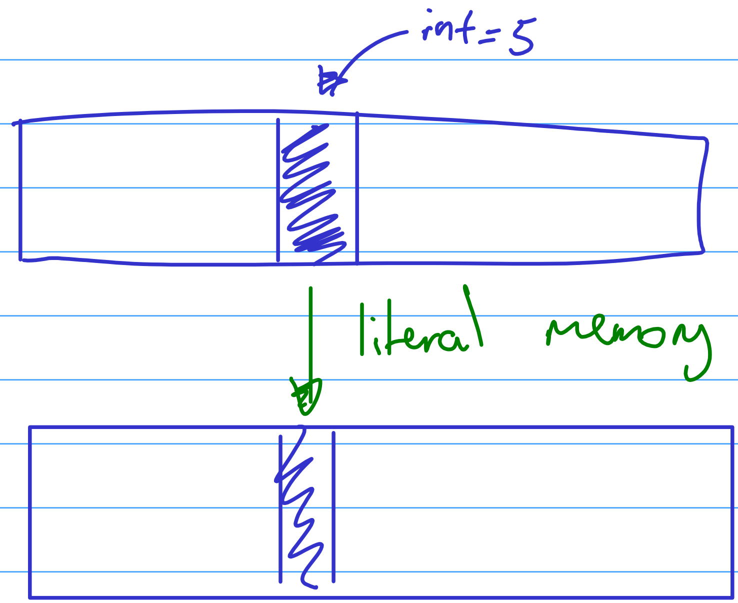
← in Object

if this class is not Cloneable

  throw CloneNotSupportedException

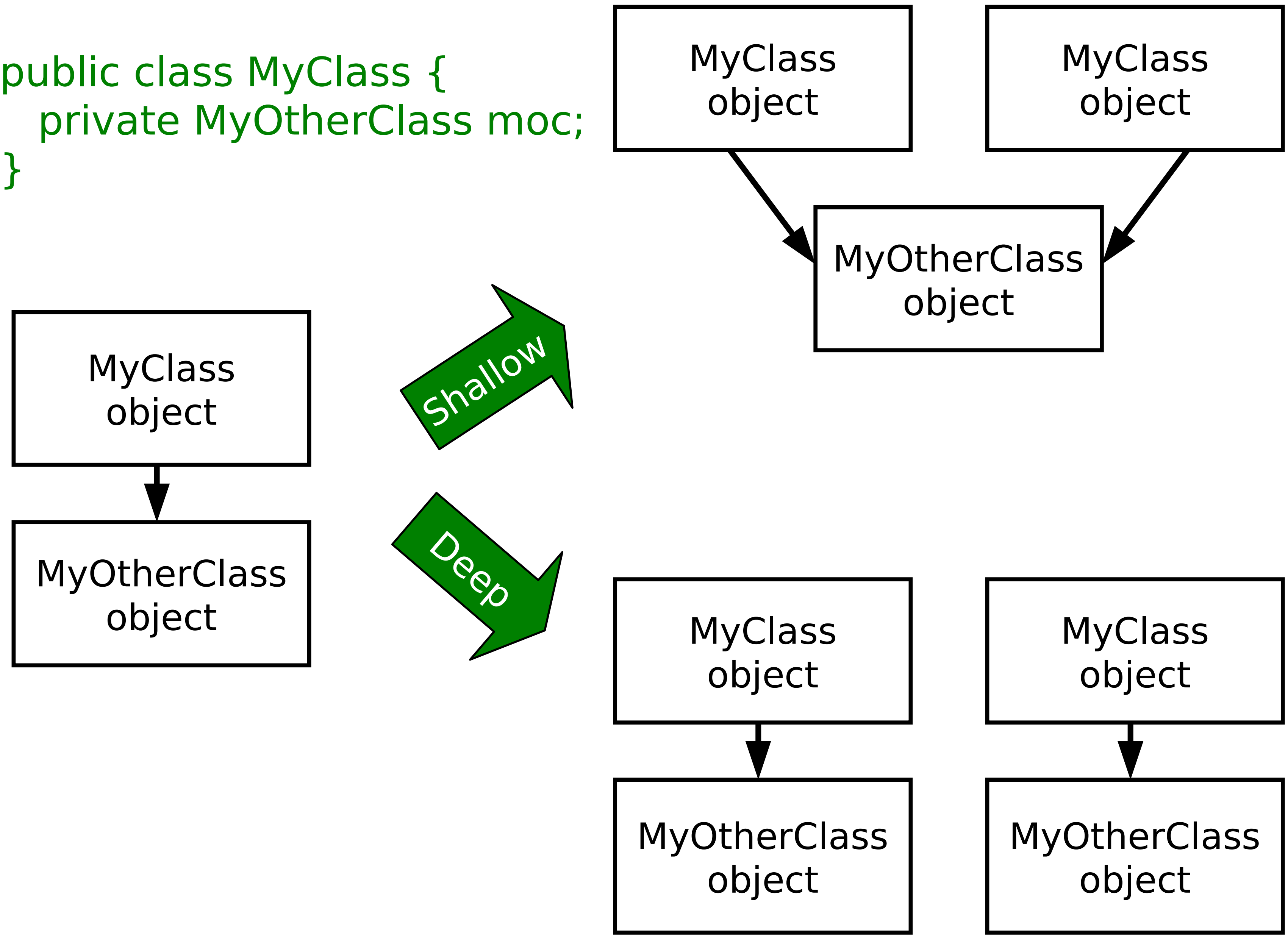
else

  Do a bit-for-bit copy



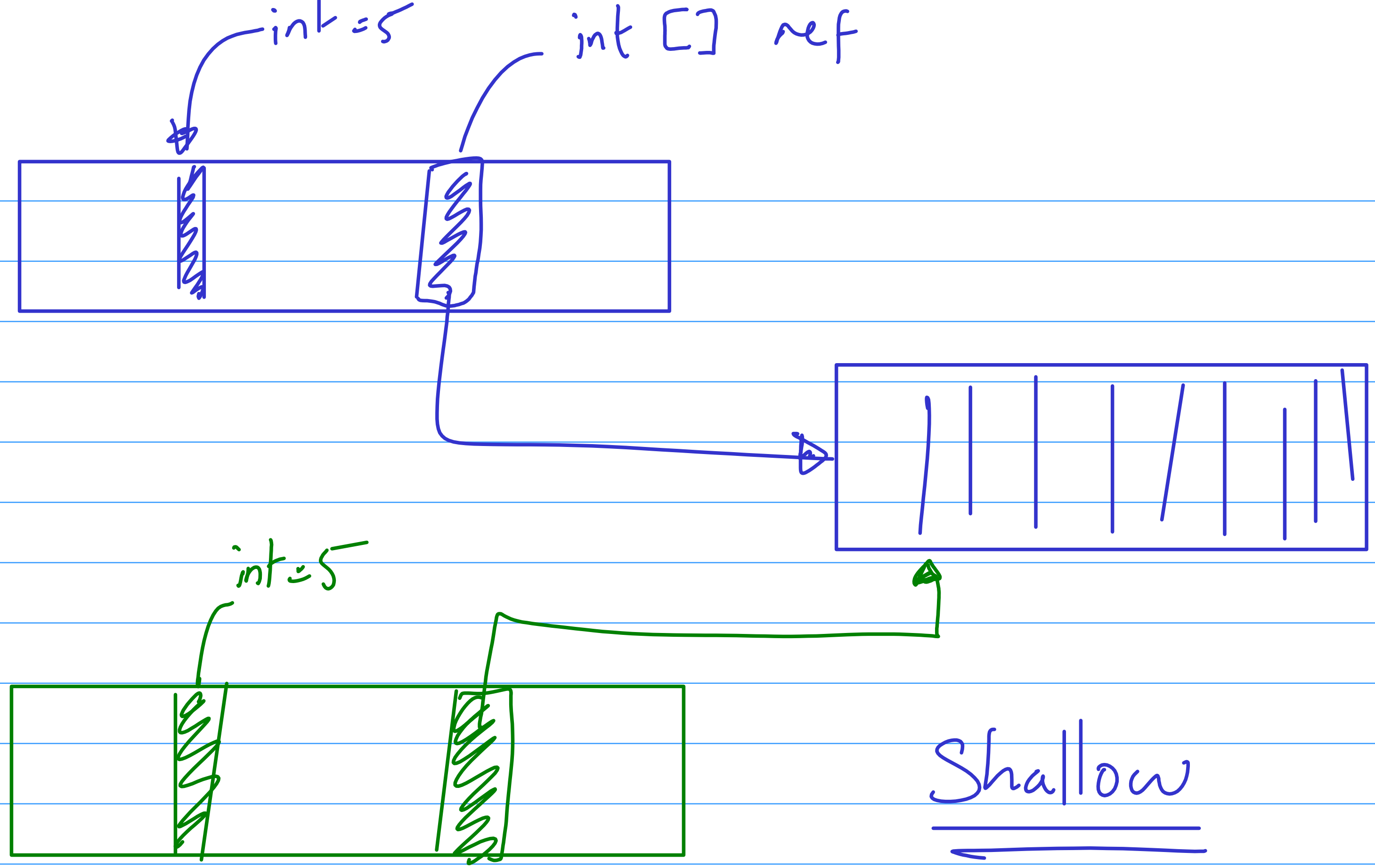
Shallow and Deep Copies

```
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 <u>sure</u> 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



Shallow

default result from super.clone()

Clone Example I

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

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

Clone Example II

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

  public Object clone() {
     return super.clone();
  }

};
```

Clone Example III

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

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

  public Object clone() {
      Vehicle cloned = (Vehicle) super.clone();
      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

- 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
- See examples sheet

```
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();
  }
}
```

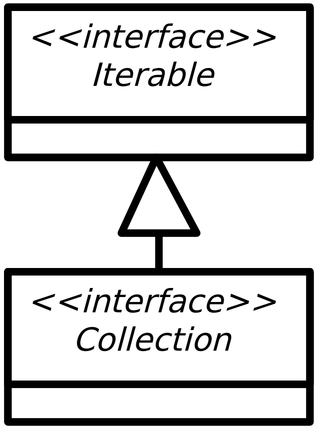
Lecture 8:
Java Collections

Java Class Library

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

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

Java's Collections Framework



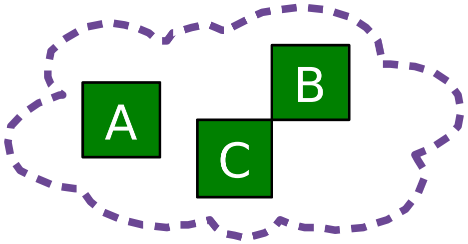
- Important chunk of the class library
- A collection is some sort of grouping of things (objects)
- Usually when we have some grouping we want to go through it ("***iterate*** over it")

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

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)



No duplicates

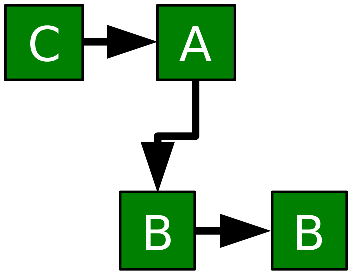
Sorted

```
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

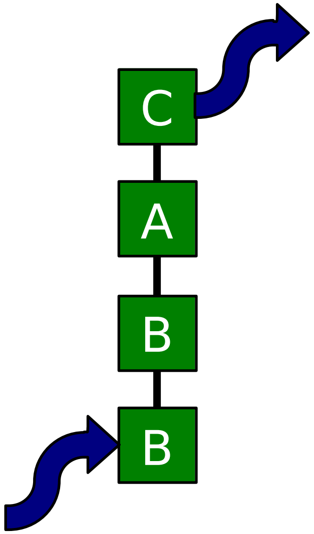



```
LinkedList<Double> ll = new LinkedList<Double>();
ll.add(1.0);
ll.add(0.5);
ll.add(3.7);
ll.add(0.5);
ll.get(1);  // get element 2 (==3.7)
```

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

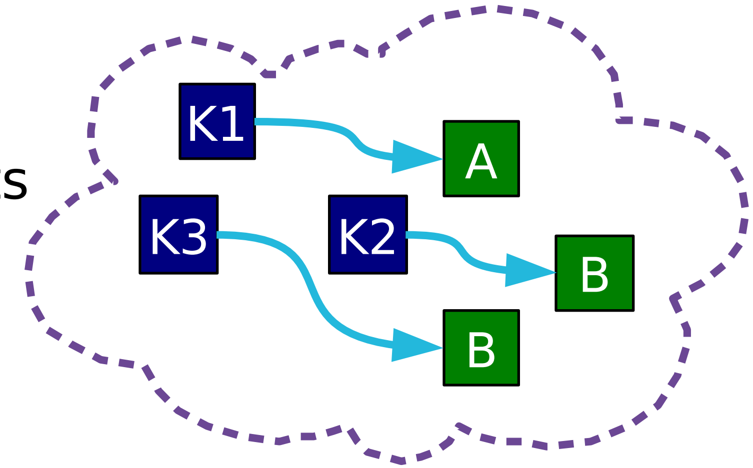



```
LinkedList<Double> ll = new LinkedList<Double>();
ll.offer(1.0);
ll.offer(0.5);
ll.poll(); // 1.0
ll.poll(); // 0.5
```

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
```

Iteration

- for loop

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

Traditional

- foreach loop (Java 5.0+)

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

Syntactic sugar

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();
}
```

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;    // ugly
}
```

- 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 );
}
```

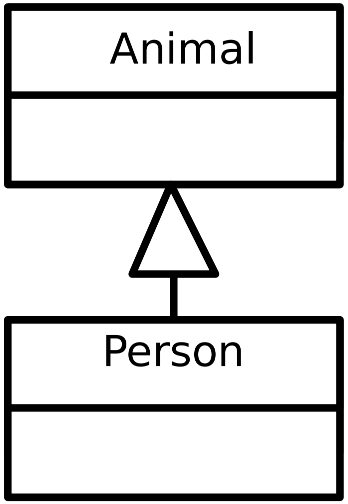
The C++ Templates Solution

- 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?

alist.add(new Hippo());

```java
import java.util.LinkedList;

public class Naughty {

	public static void main(String[] args) {
		LinkedList temp = new LinkedList<Double>();
		temp.add(9.0);
		printAll(temp);  // Should fail
	}

	public static void printAll(LinkedList<Object> list) {
		for (Object d : list)
			System.out.println((Double)d);
	}

}
```

Lecture 10:
Comparing Objects

Comparing Primitives

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

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

Reference Equality

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

```
Person p1 = new Person(“Bob”);
Person p2 = new Person(“Bob”);

(p1==p2);    ← 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));
    }
}
```

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()

Comparable<T> Interface I

```
int compareTo(T obj);
```

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

Comparable<T> Interface II

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

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

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

Comparator<T> Interface I

int compare(T obj1, T obj2)

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

Comparator<T> Interface II

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

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

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

Operator Overloading

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

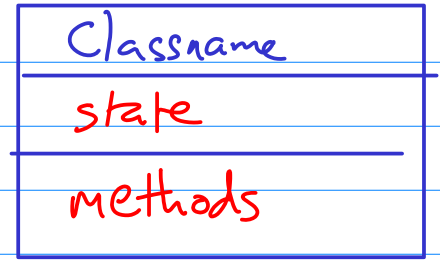
```
class Person {
  public:
    Int mAge
    bool operator==(Person &p) {
        return (p.mAge==mAge);
    };
  }


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

Lecture 11:
Design Patterns

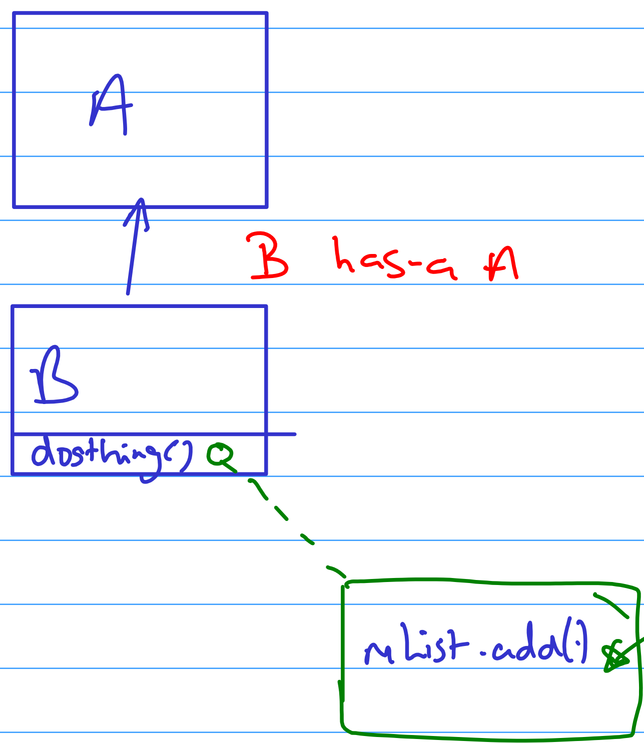
Design Patterns

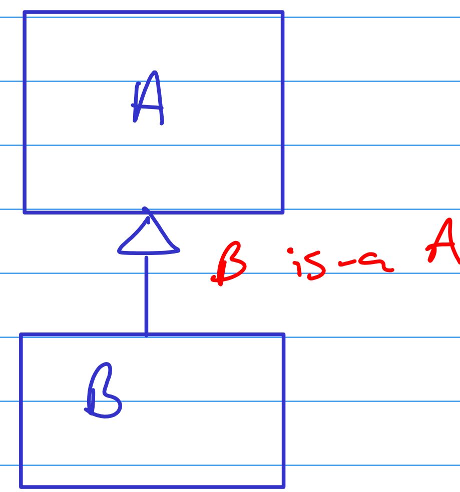
- A Design Pattern is a general reusable solution to a commonly occurring problem in software design
- Coined by Erich Gamma in his 1991 Ph.D. thesis
- Originally 23 patterns, now many more. Useful to look at because they illustrate some of the power of OOP (and also some of the pitfalls)
- We will only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

UML Revisited



− private
+ public
# protected





The Open-Closed Principle

Classes should be open for extension but closed for modification

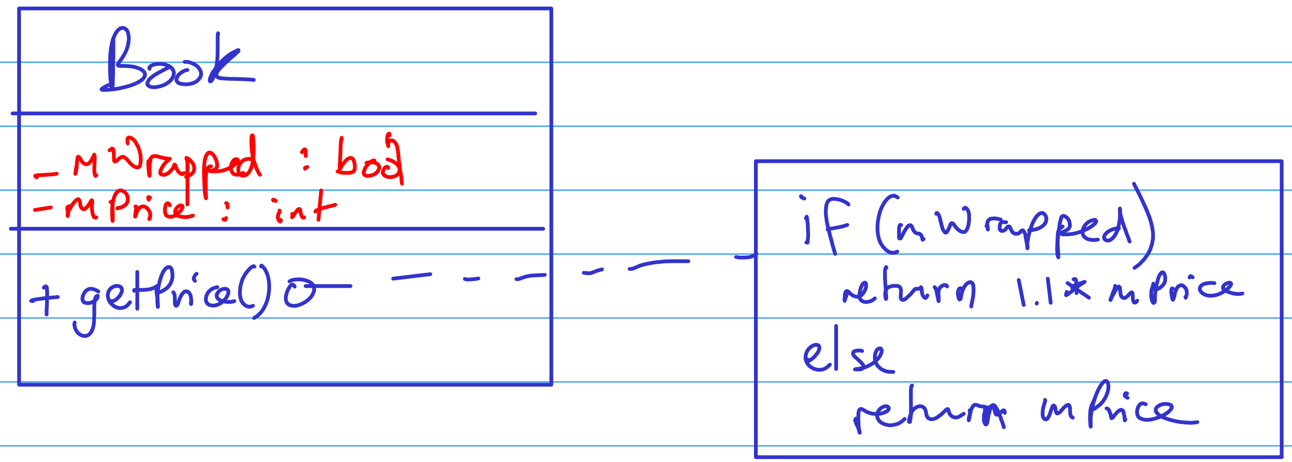
- i.e. we would like to be able to modify the behaviour without touching its source code
- This rule-of-thumb leads to more reliable large software and will help us to evaluate the various design patterns

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?

Solution 1: Add variables to the established Book class that describe whether or not the product is to be gift wrapped.

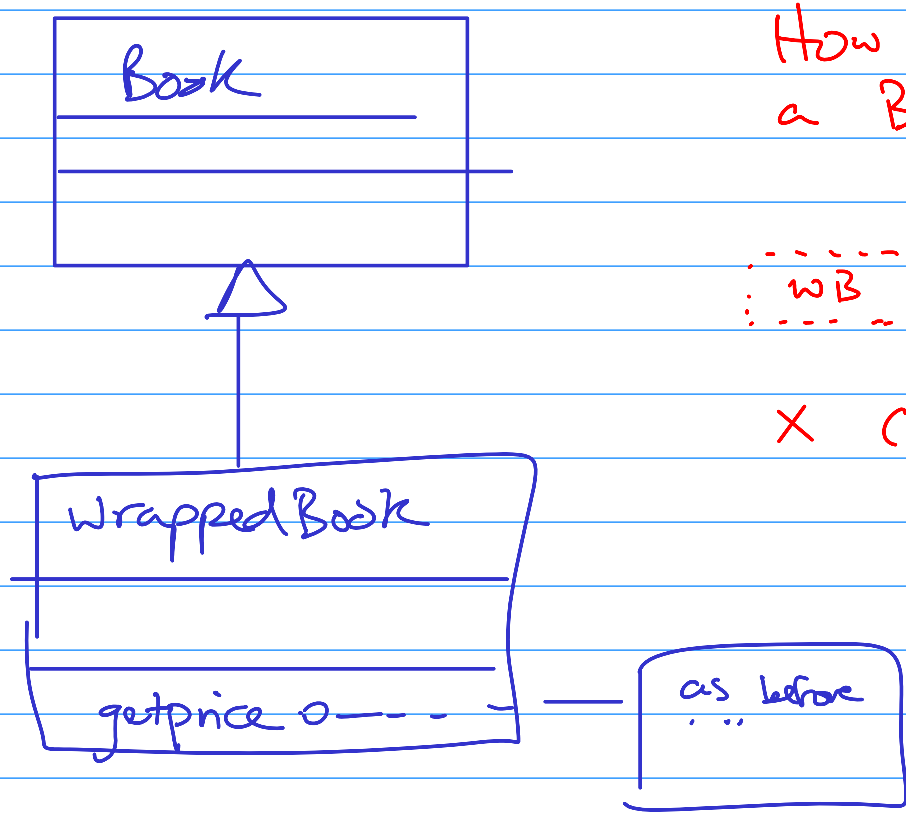


✗ violates open-closed

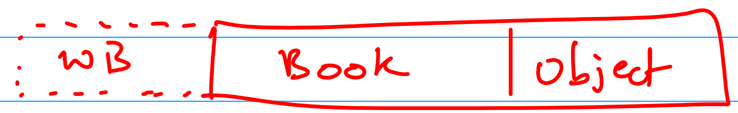
✗ Wasteful - every book wastes a boolean

✗ Hard to extend

Solution 2: Extend Book to create WrappedBook.

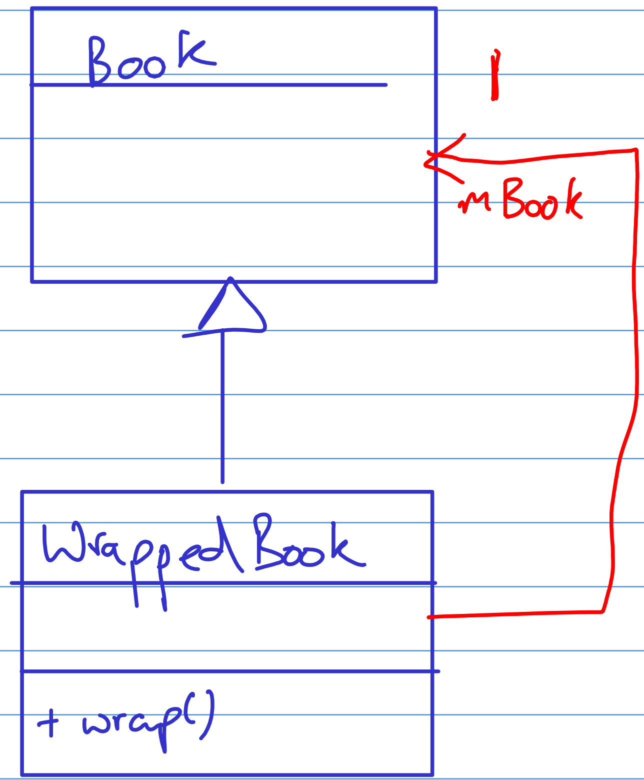


How do we convert a Book to a WrappedBook?



X Casting out

Solution 3: (Decorator) Extend Book to create WrappedBook and also add a member reference *to* a Book object. Just pass through any method calls to the internal reference, intercepting any that are to do with shipping or price to account for the extra wrapping behaviour.



wrapping around the object

Buffered Reader

Decorator class

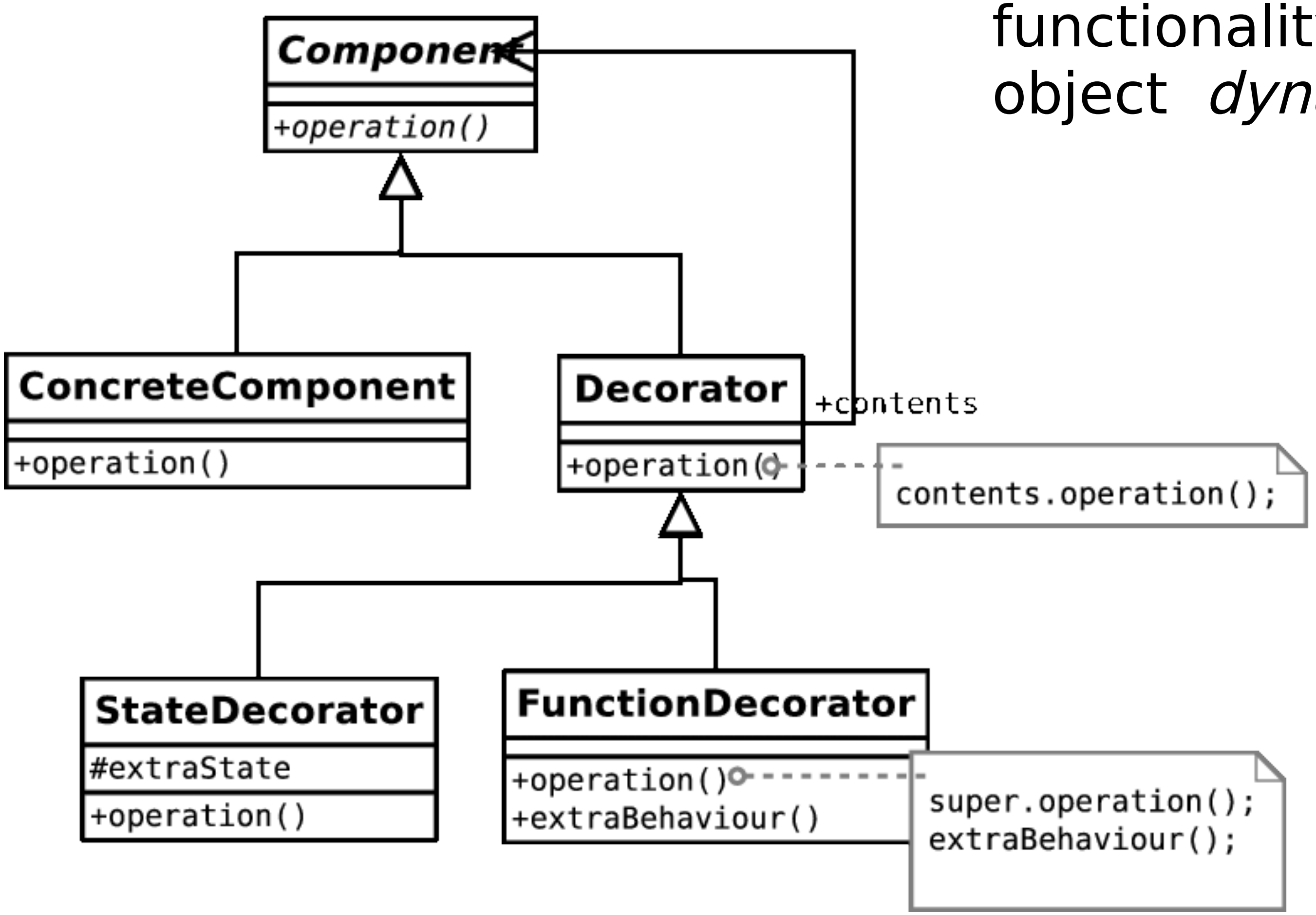
```
BufferedReader br = new BufferedReader
                        (new FileReader("myfile.txt"));
```

1. FileReader follows instructions exactly + gets things byte by byte. Visit the disk for every byte

   BR gives you a lookahead buffer ⇒ more efficiently

2. Extra functionality ⇒ readLine()

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.

Singleton

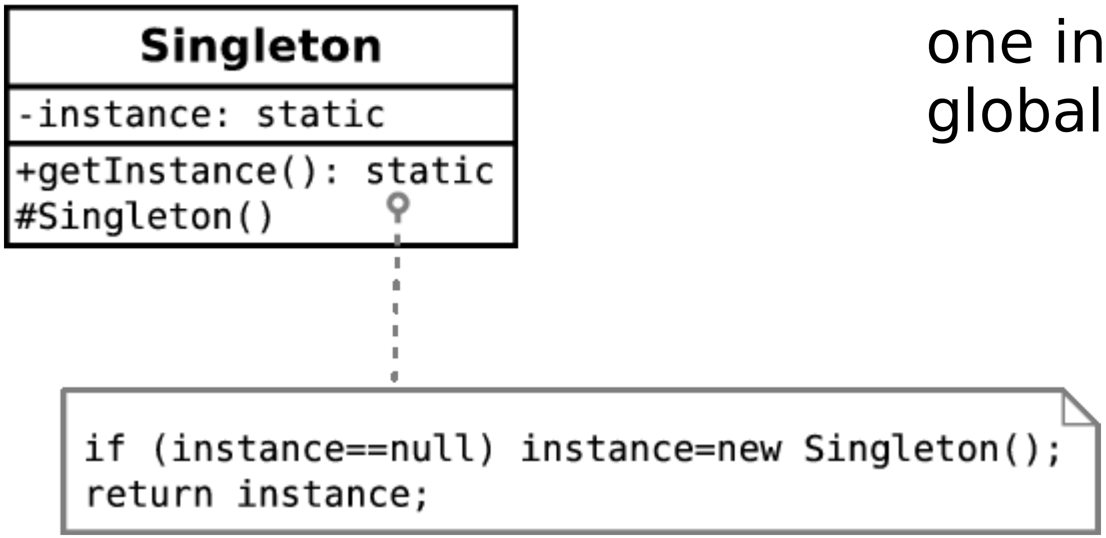
1. Make the constructor private
2. Create a single **static** instance
3. Make a **static** getter for your instance

java.util.Runtime

Runtime.getRuntime()

freeMemory()

Singleton in General



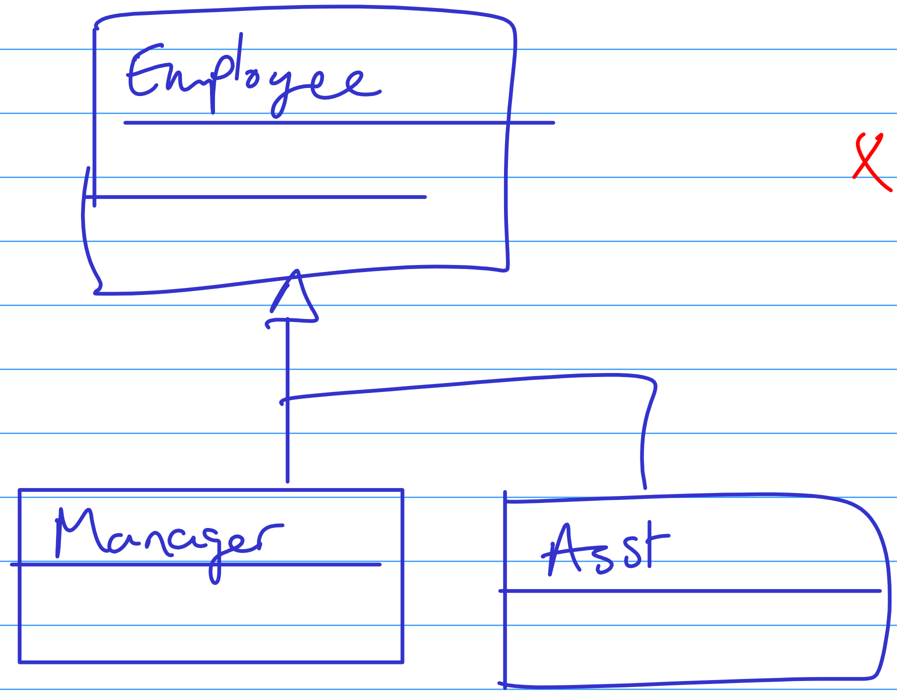
- The singleton pattern ensures a class has only one instance and provides global access to it

State

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

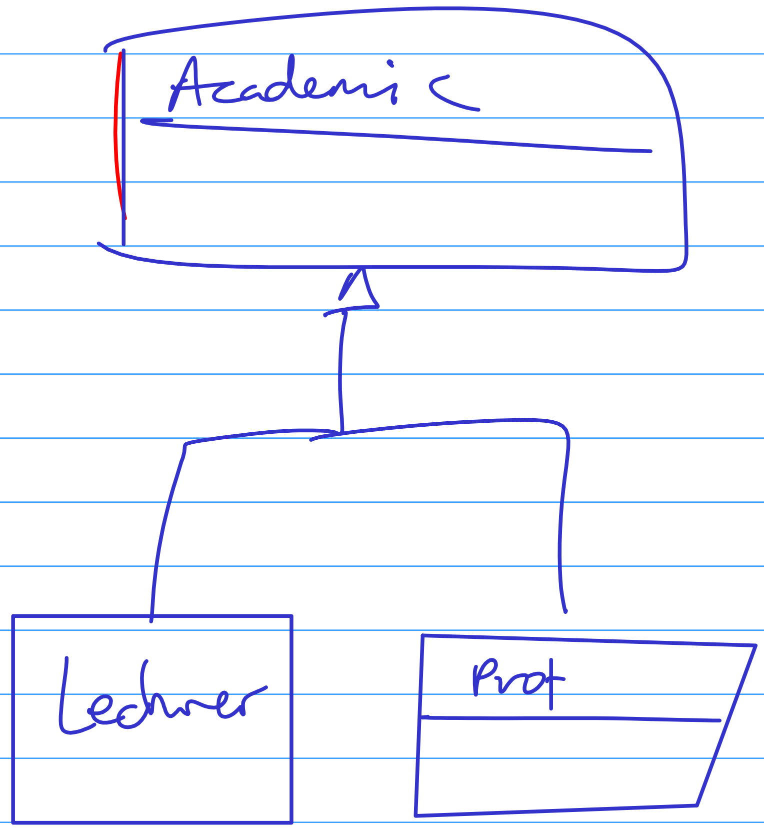
Example problem: Representing academics ~~as~~ if they progress through the rank

Examples Sheet Problem



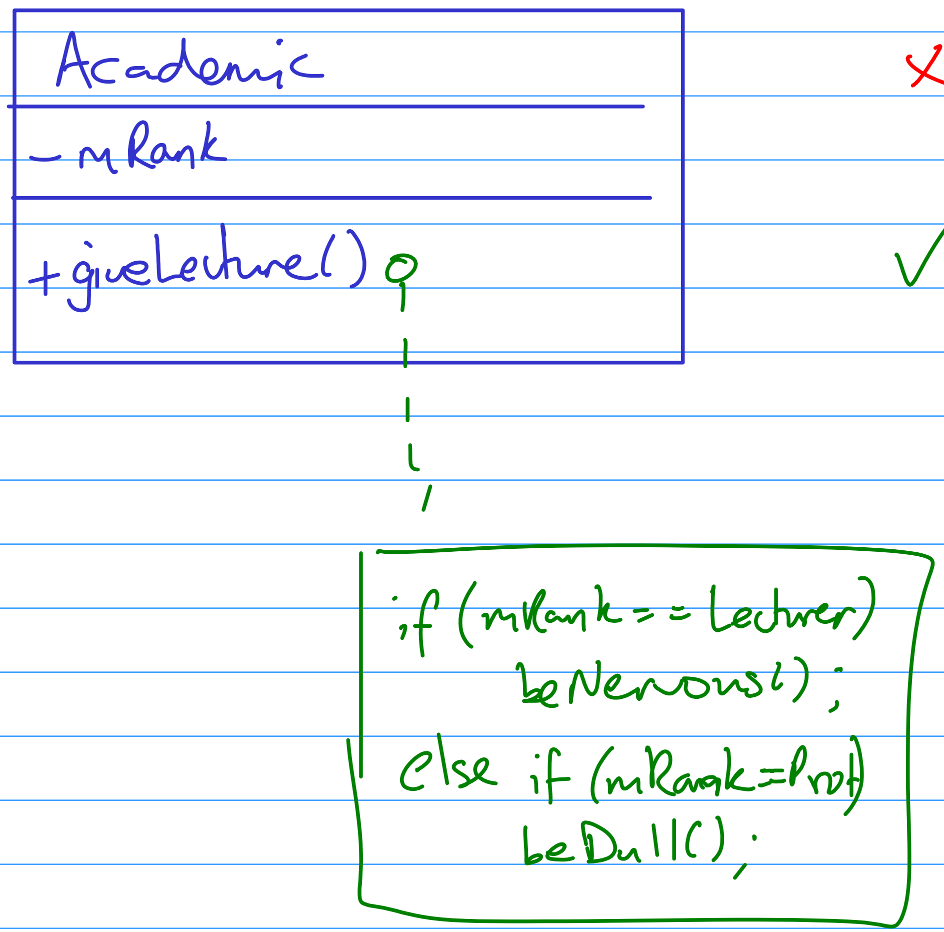
X cannot convert objects

Solution 1: Have an abstract Academic class which acts as a base class for Lecturer, Professor, etc.



Same as last slide

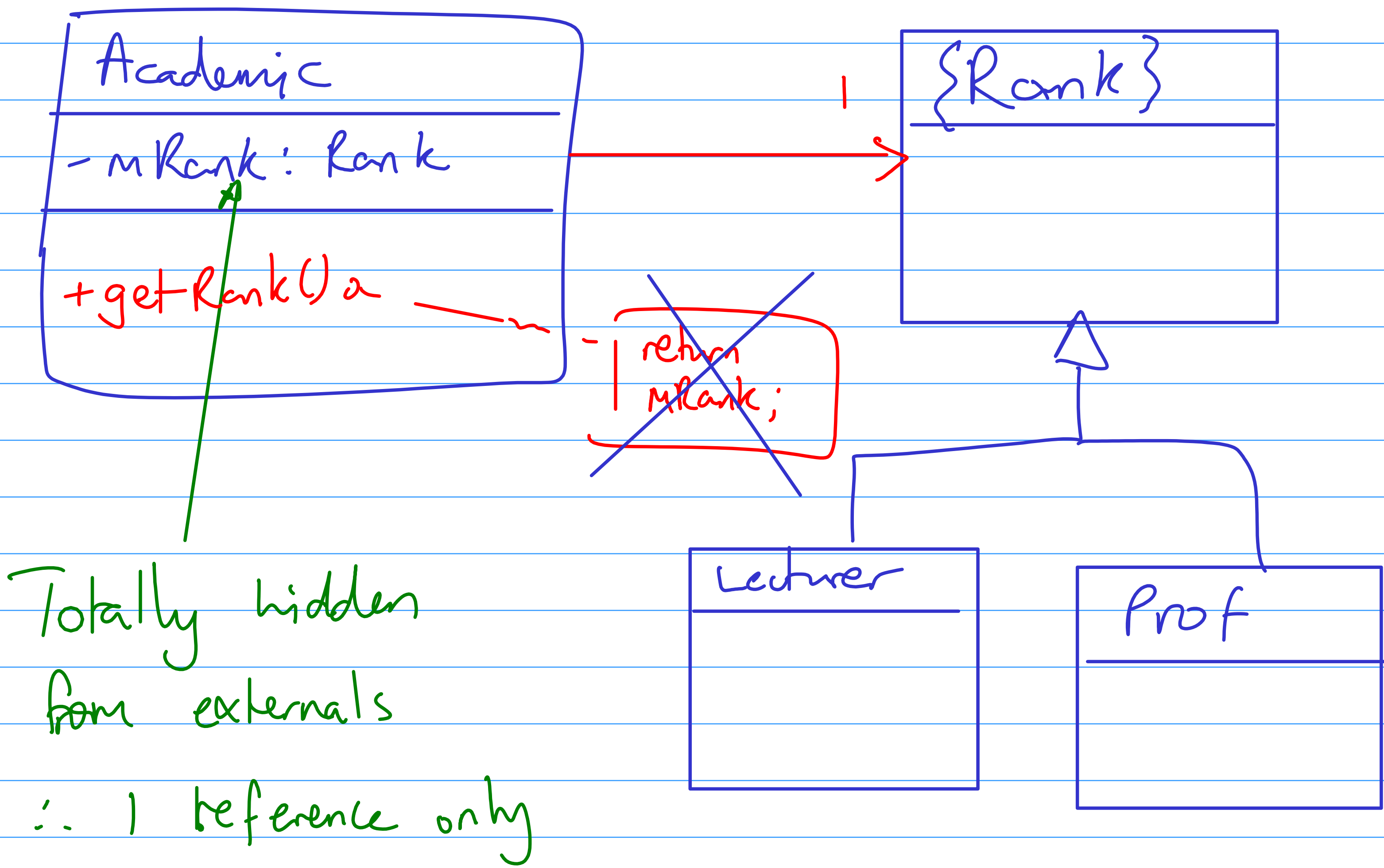
Solution 2: Make Academic a concrete class with a member variable that indicates rank. To get rank-specific behaviour, check this variable within the relevant methods.



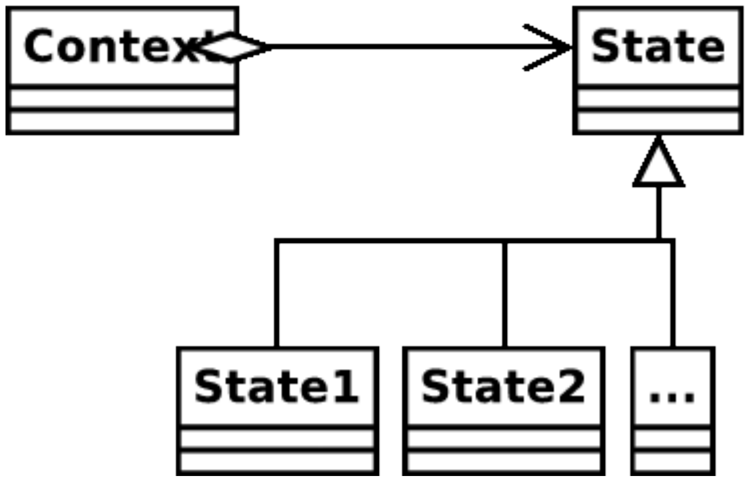
✗ Lose OOP advantages

✓ Does work but ugly!

Solution 3: (State) Make Academic a concrete class that has-a AcademicRank as a member. Use AcademicRank as a base for Lecturer, Professor, etc., implementing the rank-specific behaviour in each..



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?

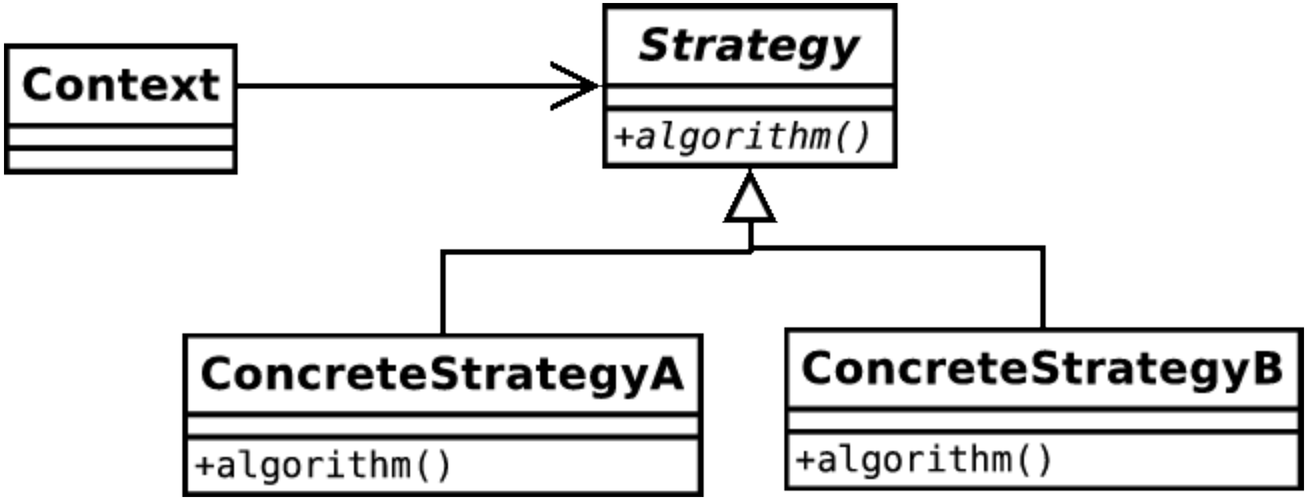
Solution 1: Use a lot of `if...else` statements in the `getChange(...)` method.

Solution 2: (Strategy) Create an abstract ChangeFinder class. Derive a new class for each of our algorithms.

java.util.Comparator

Strategy in General

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

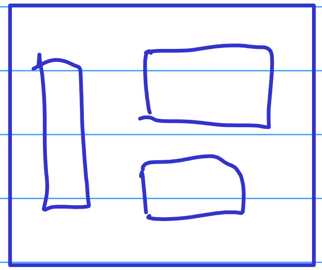


Composite

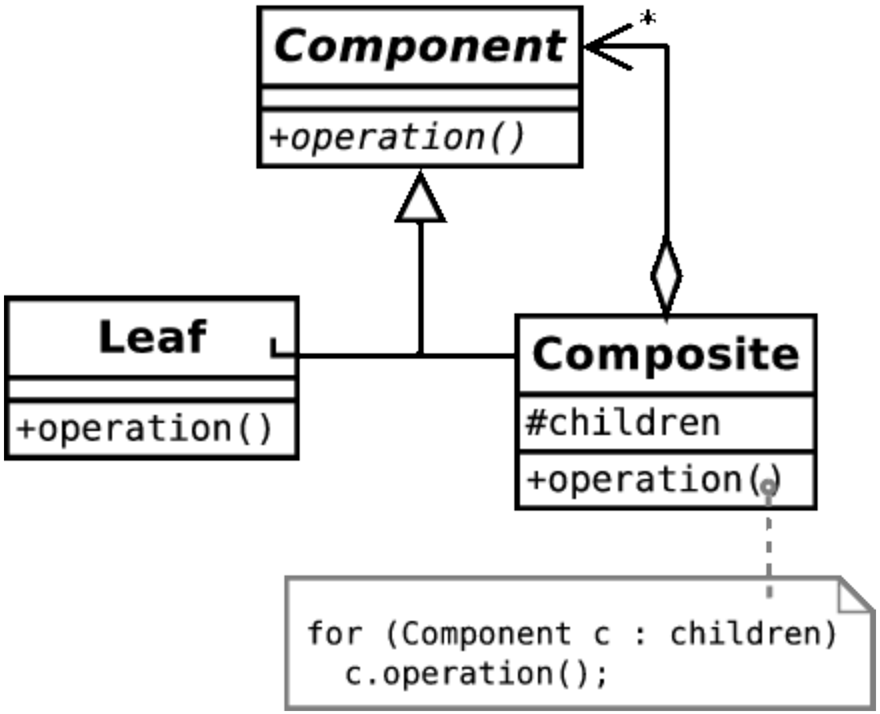
Abstract problem: How can we treat a group of objects as a single object?

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

GUI Layout



Composite in General



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

Observer

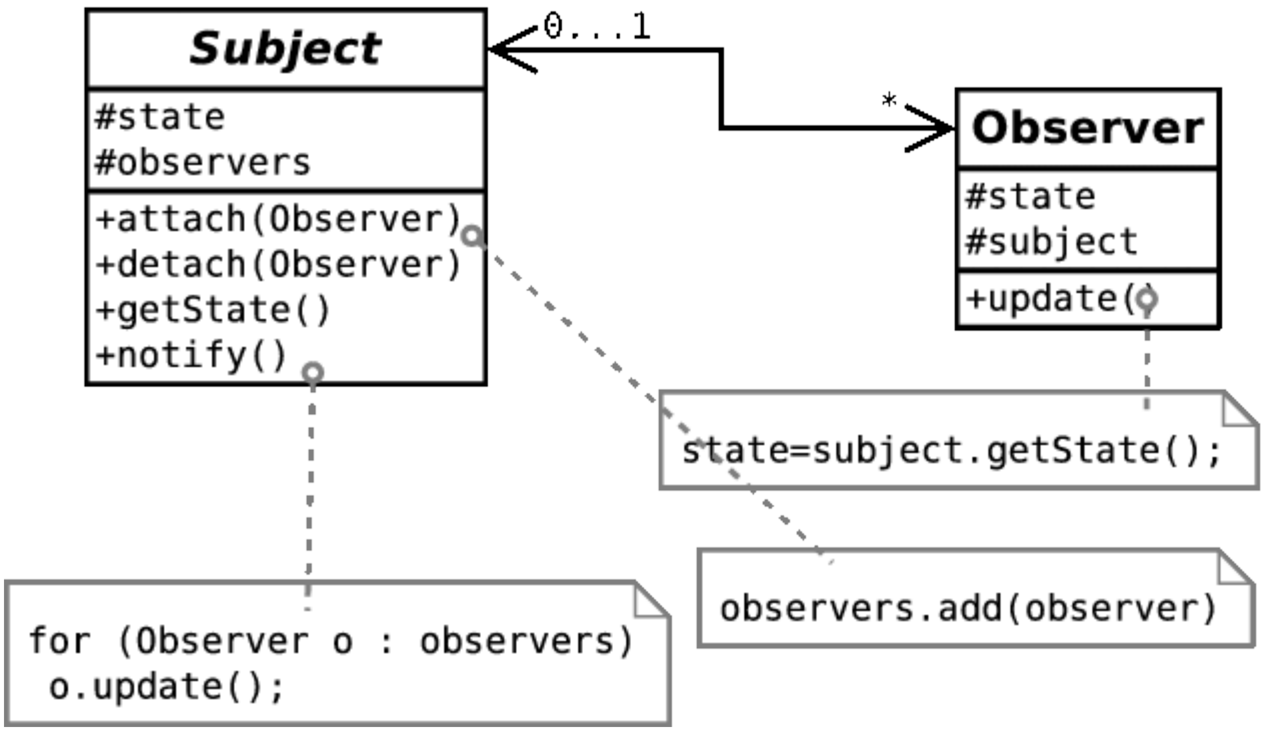
Abstract problem: When an object changes state, how can any interested parties know?

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

Observer Pattern

Observer in General

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



Factory Pattern

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)?

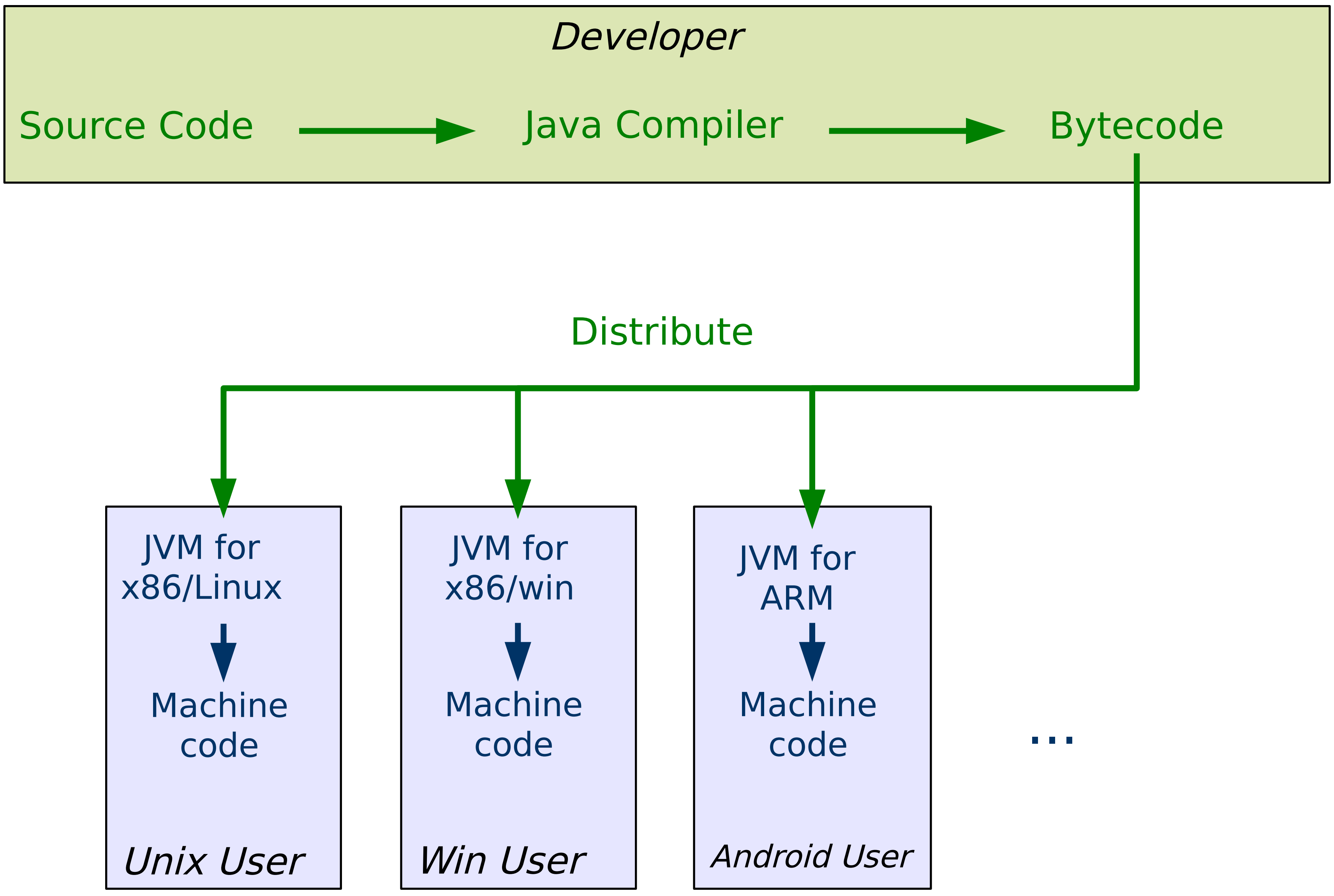
- Could use an interpreter (→ Javascript). But:
  - High level languages not very space-efficient
  - The source code would implicitly be there for anyone to see, which hinders commercial viability.

- 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 ($\rightarrow$ easier job $\rightarrow$ faster job)

- Still a performance hit compared to fully compiled ("native") code