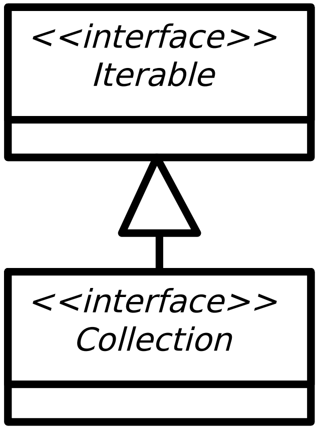
Lecture 7:
Java Collections and Object Comparison

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...

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

Java's Collections Framework



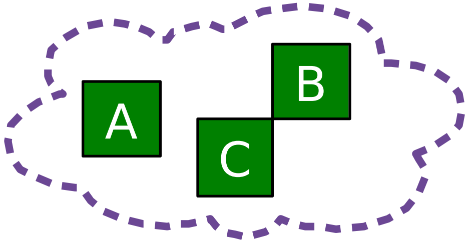
- Important chunk of the class library
- A collection is some sort of grouping of things (objects)
- Usually when we have some grouping we want to go through it ("***iterate*** over it")

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

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)

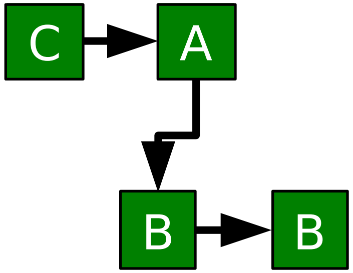


```
TreeSet<Integer> ts = new TreeSet<Integer>();
ts.add(15);
ts.add(12);
ts.contains(7);  // false
ts.contains(12); // true
ts.first(); // 12 (sorted)
```

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

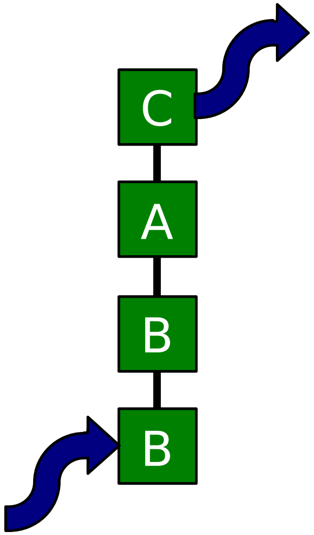


```
LinkedList<Double> ll = new LinkedList<Double>();
ll.add(1.0);
ll.add(0.5);
ll.add(3.7);
ll.add(0.5);
ll.get(1);  // get element 2 (==3.7)
```

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

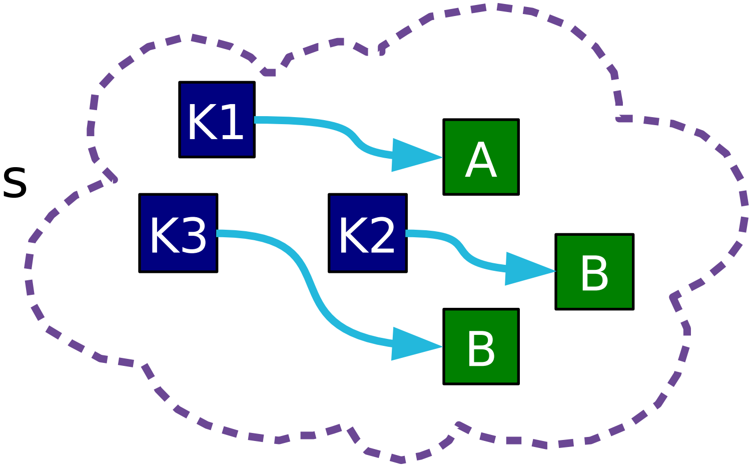


```
LinkedList<Double> ll = new LinkedList<Double>();
ll.offer(1.0);
ll.offer(0.5);
ll.poll(); // 1.0
ll.poll(); // 0.5
```

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
```

Iteration

- for loop

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

- foreach loop (Java 5.0+)

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

Iterators

- What if our loop changes the structure?

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

- Java introduced the Iterator class

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

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

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

- Safe to modify structure

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

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));
    }
}
```

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 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!!”);
```

- 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
}
```

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();
}
```

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)

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

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) {
}
```

- ...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

Assertions

- 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!

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

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

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

```
> java -ea SomeClass

> java -da SomeClass
```

As Oracle Puts It

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

Great for Postconditions

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

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

Sometimes for Preconditions

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

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

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

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

- (you should use <u>exceptions</u> for this)

Sqrt Example

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

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

   // Postcondition
   assert (result>=0);

   return result;
}
```

Assertions can be Slow if you Like

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

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

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

NOT for Checking your Compiler/Computer

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

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

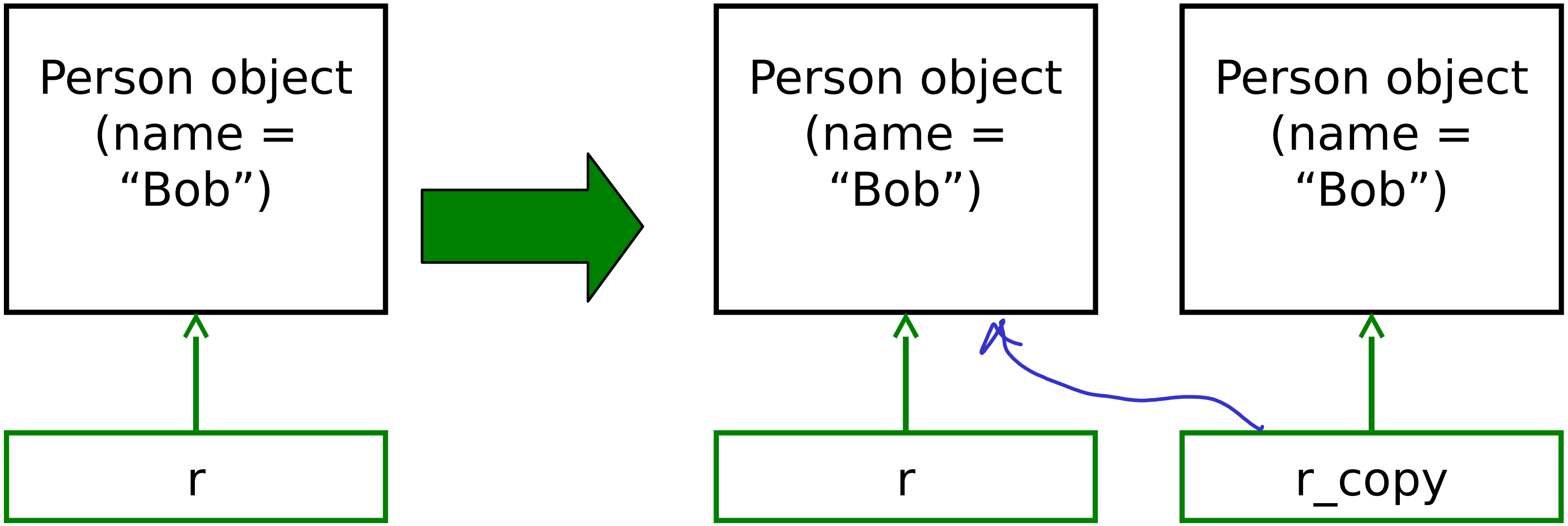
For the Last Word on Assertions...

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

Lecture 9:
Copying Objects

Cloning I

- Sometimes we really do want to copy an object


Person object (name = “Bob”)
r
Person object (name = “Bob”)
r
Person object (name = “Bob”)
r_copy

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

r_copy = r;

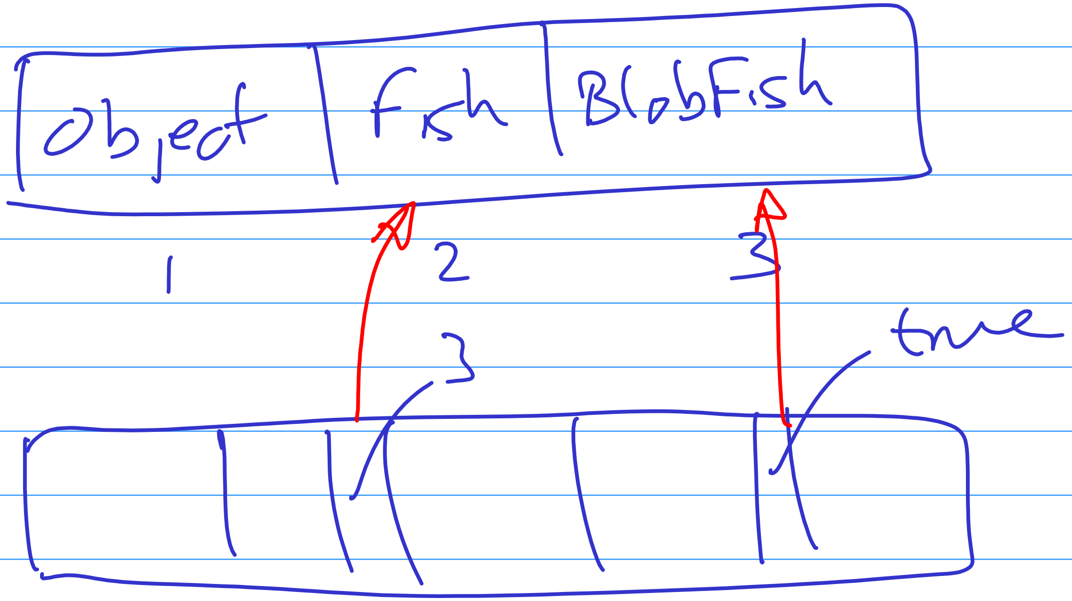
Polymorphic Copy

⇒ Provide copy() method in classes.

⇒ Polymorphic copy (copy when reference is cast to parent)

⇒ Struggled to copy parent state without nasty hack

Copy Constructor



⇒ Copied all parent state

⇒ Failed to cope with parent cast (i.e. cast up the tree)

Clone() Recipe

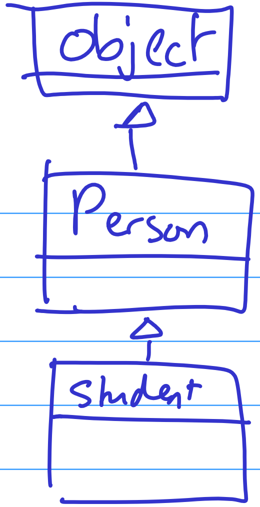
1. implements Cloneable

2. Make a public clone() method

3 Call super.clone() to get a bit-for-bit copy

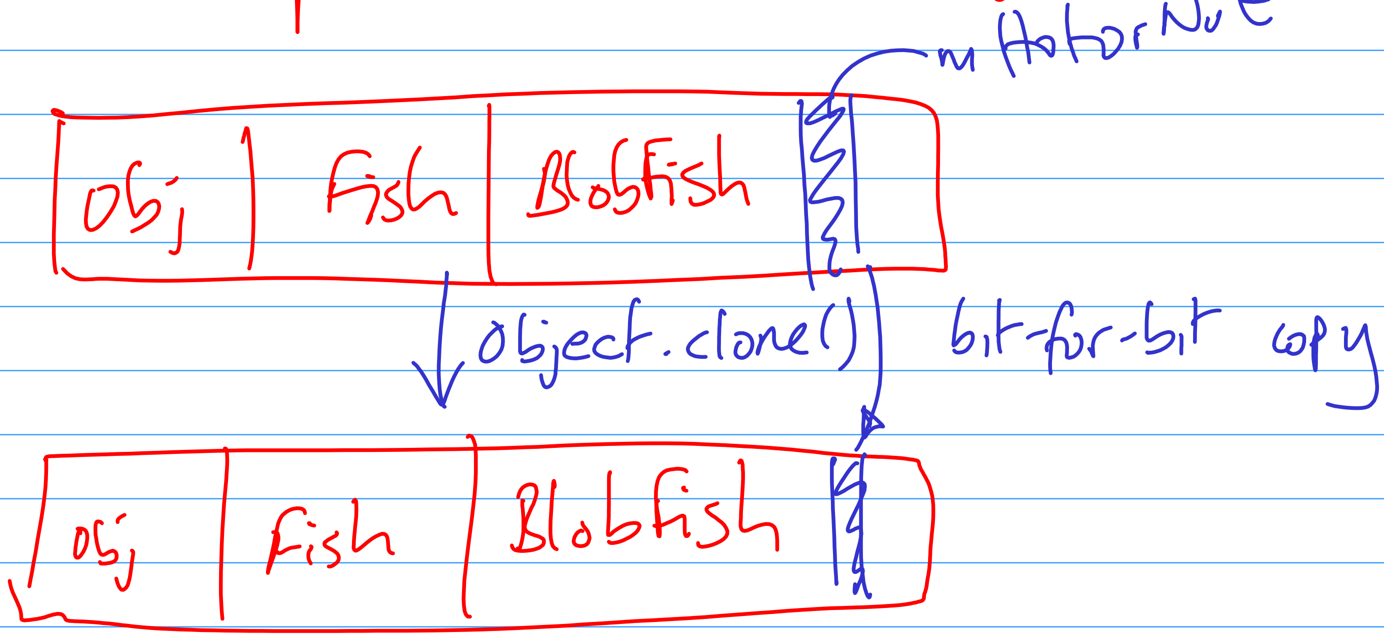
4. set any state in your own class

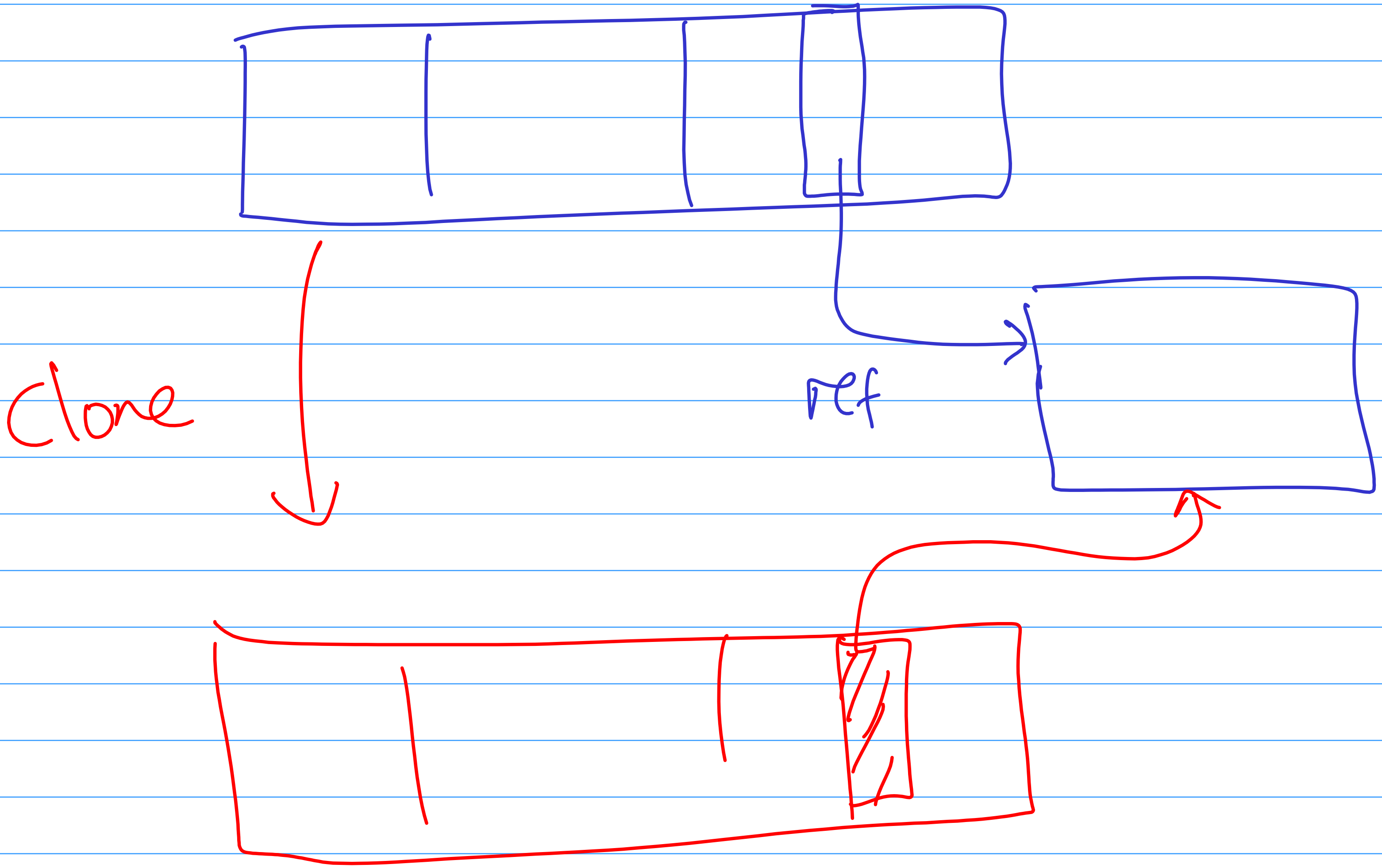
Object.clone()



super.clone() → Object.clone()

special method in object



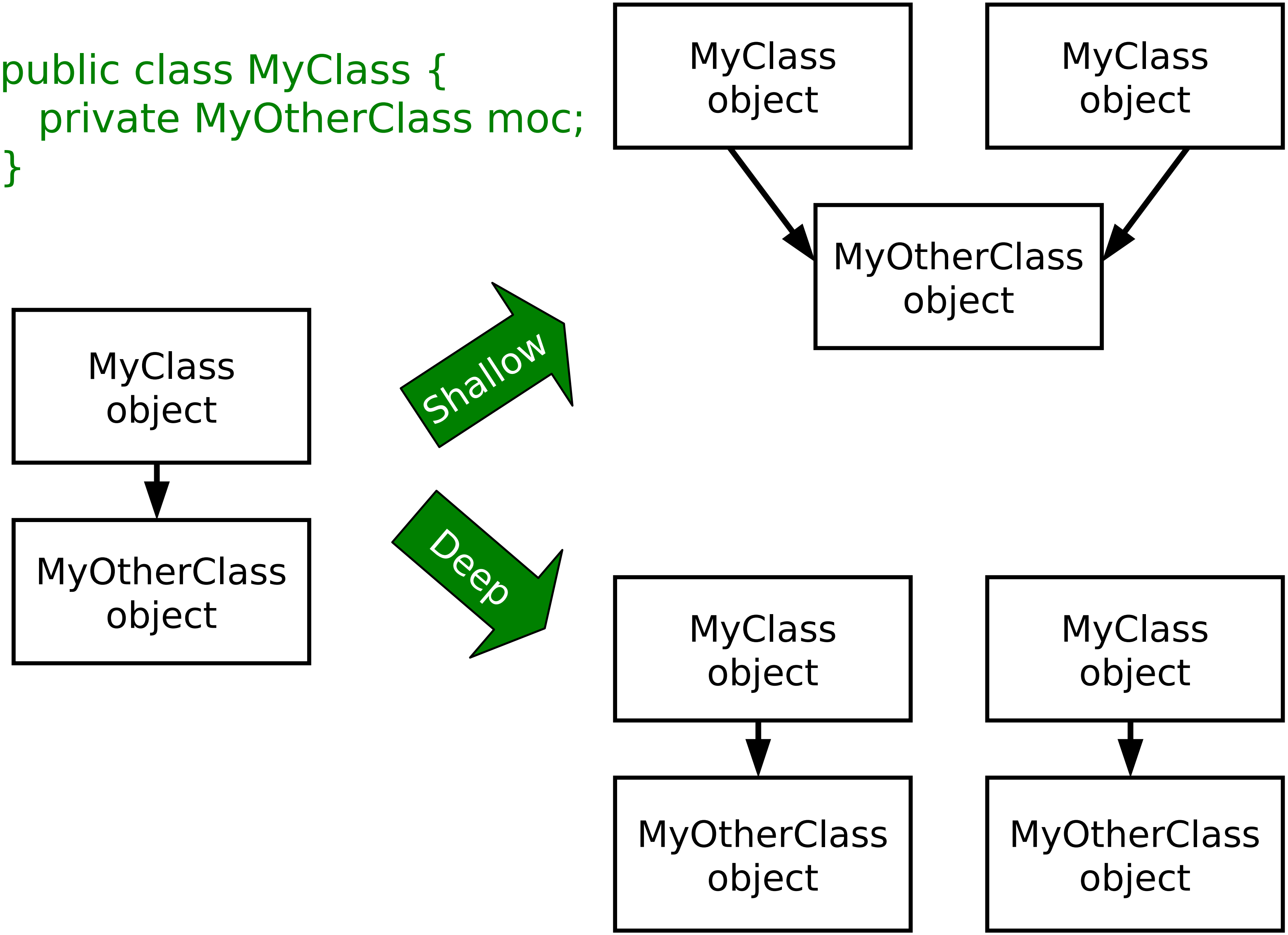

Clone
ref

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;
}
```



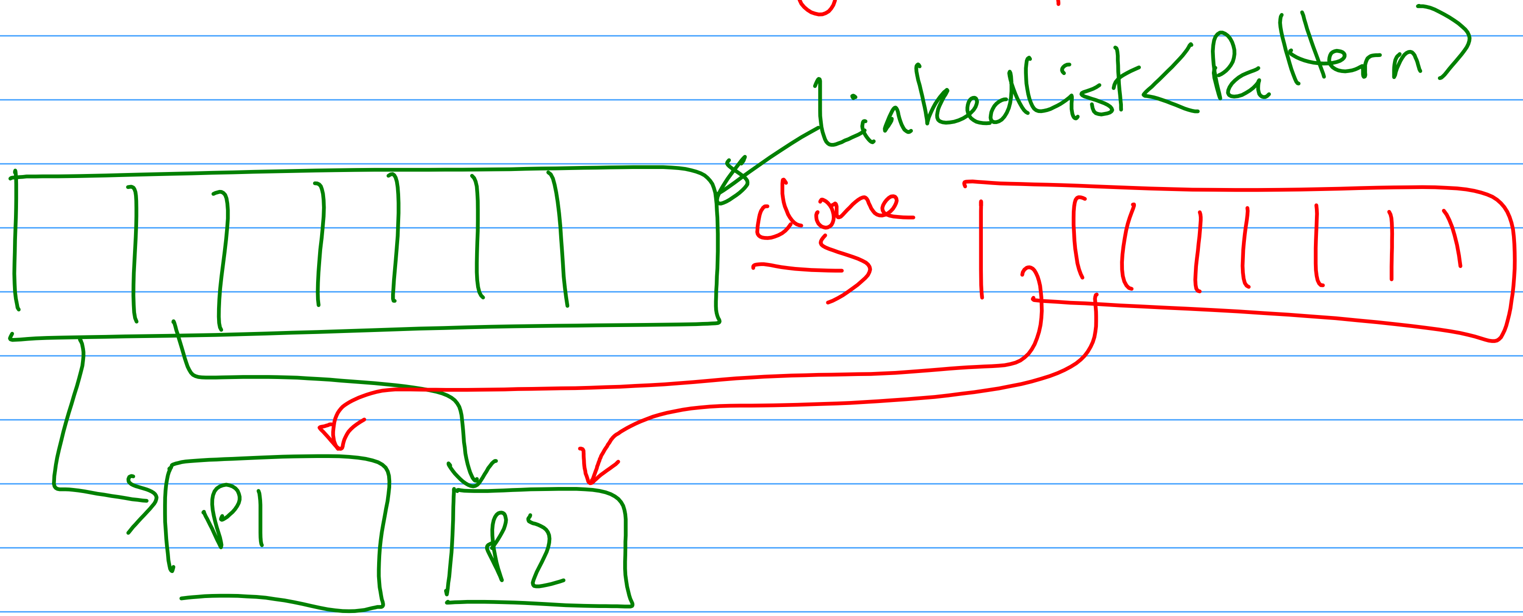
Java Cloning

- 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

Cloning primitives ⇒ clone does this for you

Cloning references (mutable) ⇒ do a deep clone to copy them too

Cloning immutable objects ⇒ Don't go deep



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 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

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

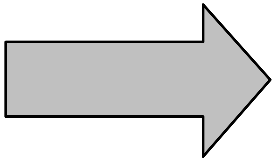
```
// Make a TreeSet object
TreeSet ts = new TreeSet();

// 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);
}
```

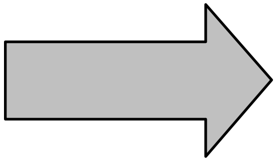


```
LinkedList ll =
    new LinkedList();

...

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

The C++ Templates Solution

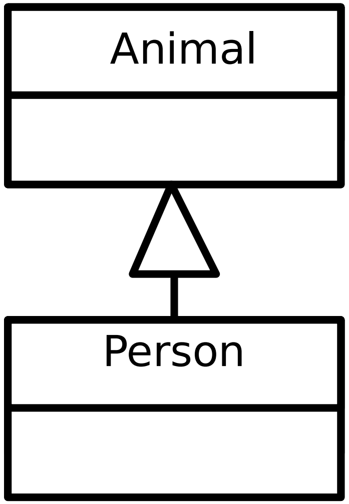
- Compiler first generates the class definitions from the template

```
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?

No: it would mean we could write:

```
List<Animal> alist = (List<Animal>) plist;
alist.add(new Blobfish());
```

Fine to add to list<Animal>
but it's really List<Person>!!

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)
}
```

- Enter Java 8...

Lambda Functions

- Supports anonymous functions

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


s->s+”hello”;

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

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

Functions as Values

```
// 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!

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

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

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

Lecture 11:/12
Design Patterns

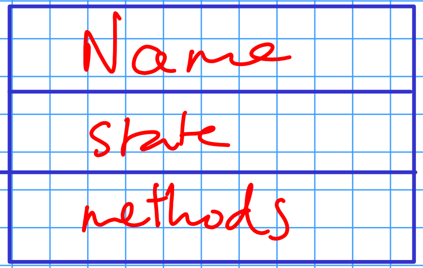
Ticks (Paper 1)

CST	NST/PBST
ML 1,2,3,4 ✓ 5 vacation	ML 1,2,3,4 ✓
Java 1,2,3 ✓ 4,5 vacation	Java 1,2,3 4,5 vacation
Algo 1,2,3	Algo 1

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

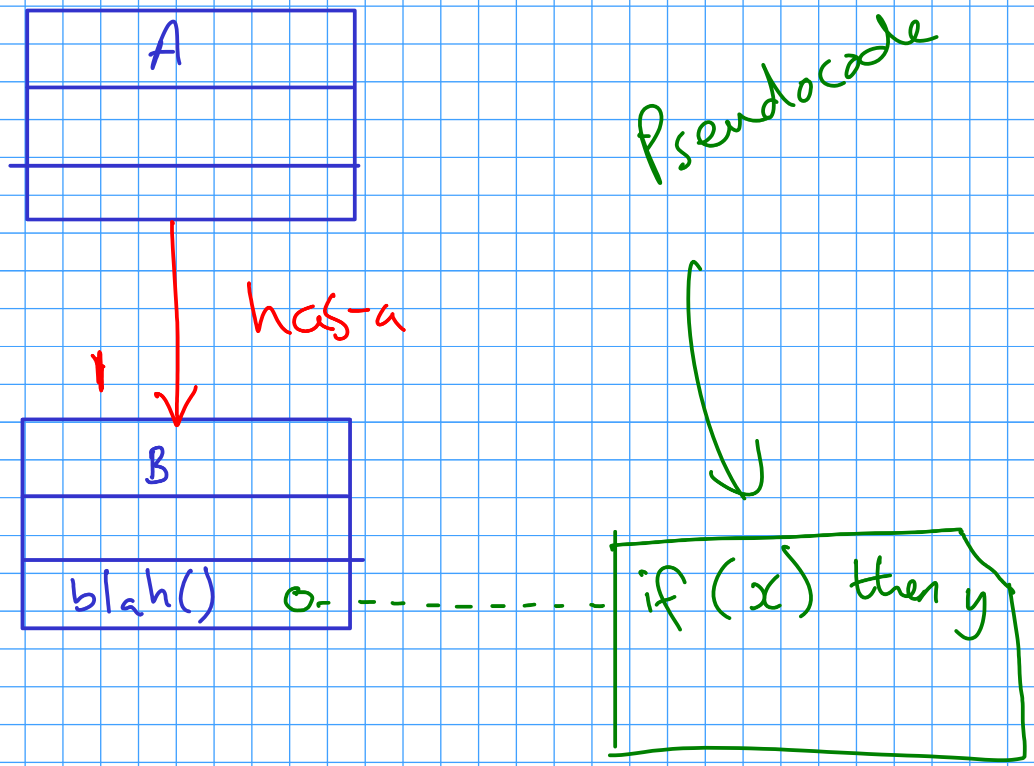
UML Refresher

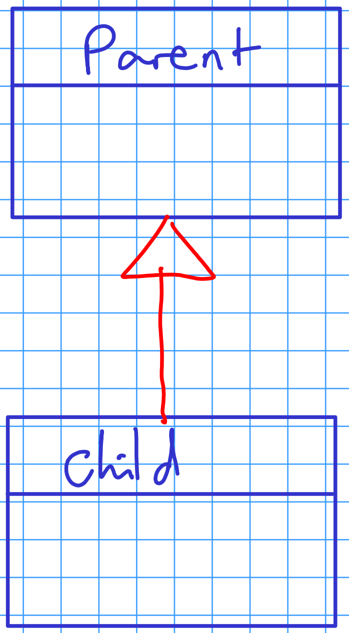


- private

+ public

# protected





The Open-Closed Principle

Classes should be open for extension but closed for modification

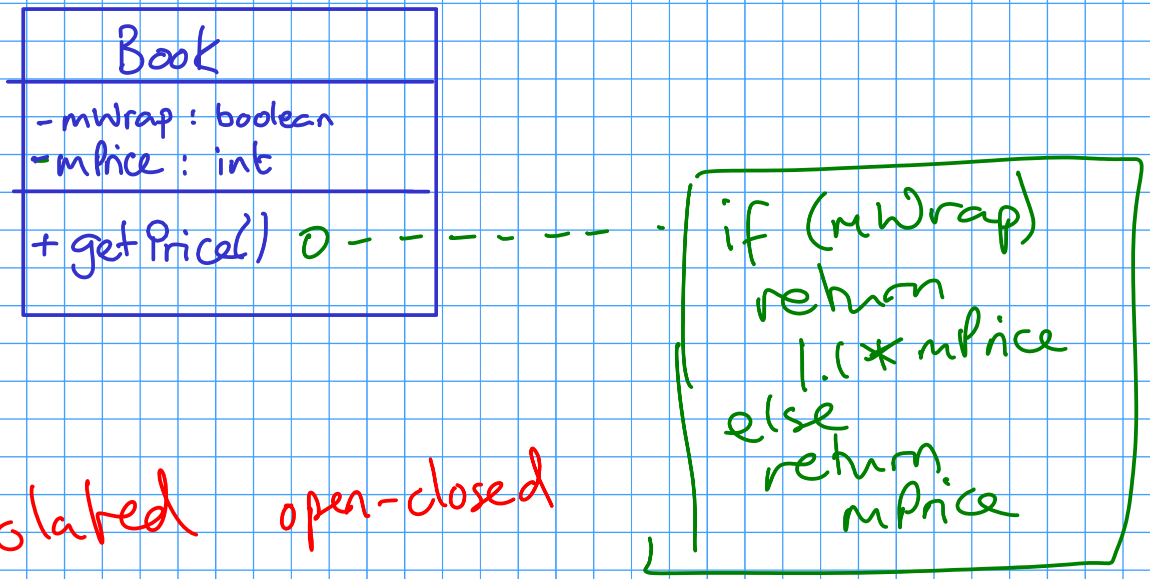
- i.e. we would like to be able to modify the behaviour without touching its source code
- This rule-of-thumb leads to more reliable large software and will help us to evaluate the various design patterns

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.



```
if (mWrap)
  return
    1.1*mPrice
else
  return
    mPrice
```

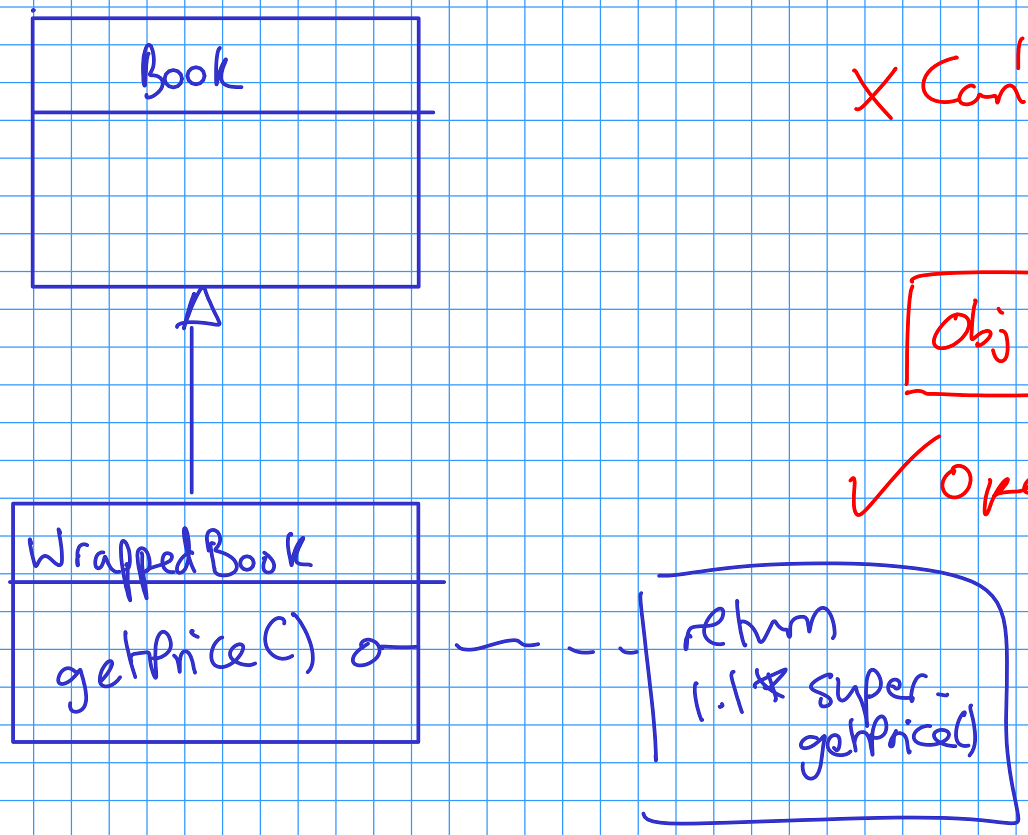
✗ violated open-closed

✗ Wasteful - wasted state

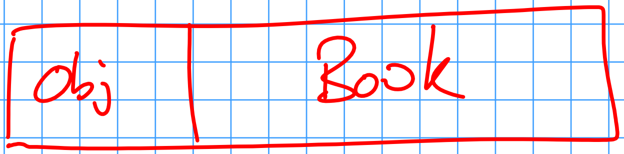
✗ Hard to extend

✓ Can easily unwrap

Solution 2: Extend Book to create WrappedBook.

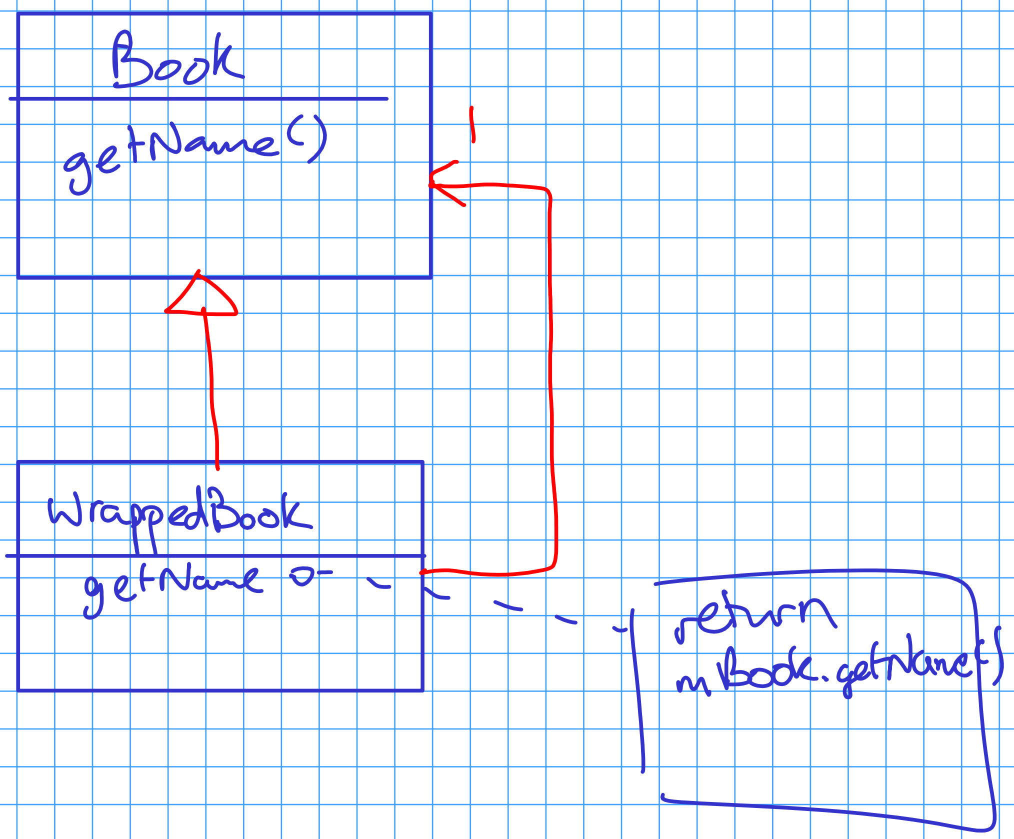


✗ Can't unwrap

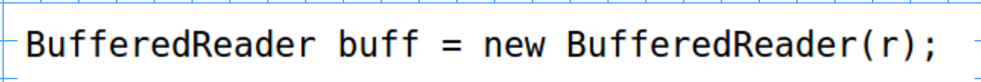


✓ Open-closed principle

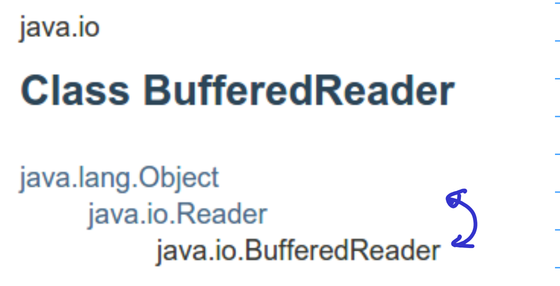
Solution 3: (Decorator) Extend Book to create WrappedBook and also add a member reference *to* a Book object. Just pass through any method calls to the internal reference, intercepting any that are to do with shipping or price to account for the extra wrapping behaviour.



Buffered Reader



```
BufferedReader buff = new BufferedReader(r);
```



java.io

Class BufferedReader

java.lang.Object
    java.io.Reader
        java.io.BufferedReader

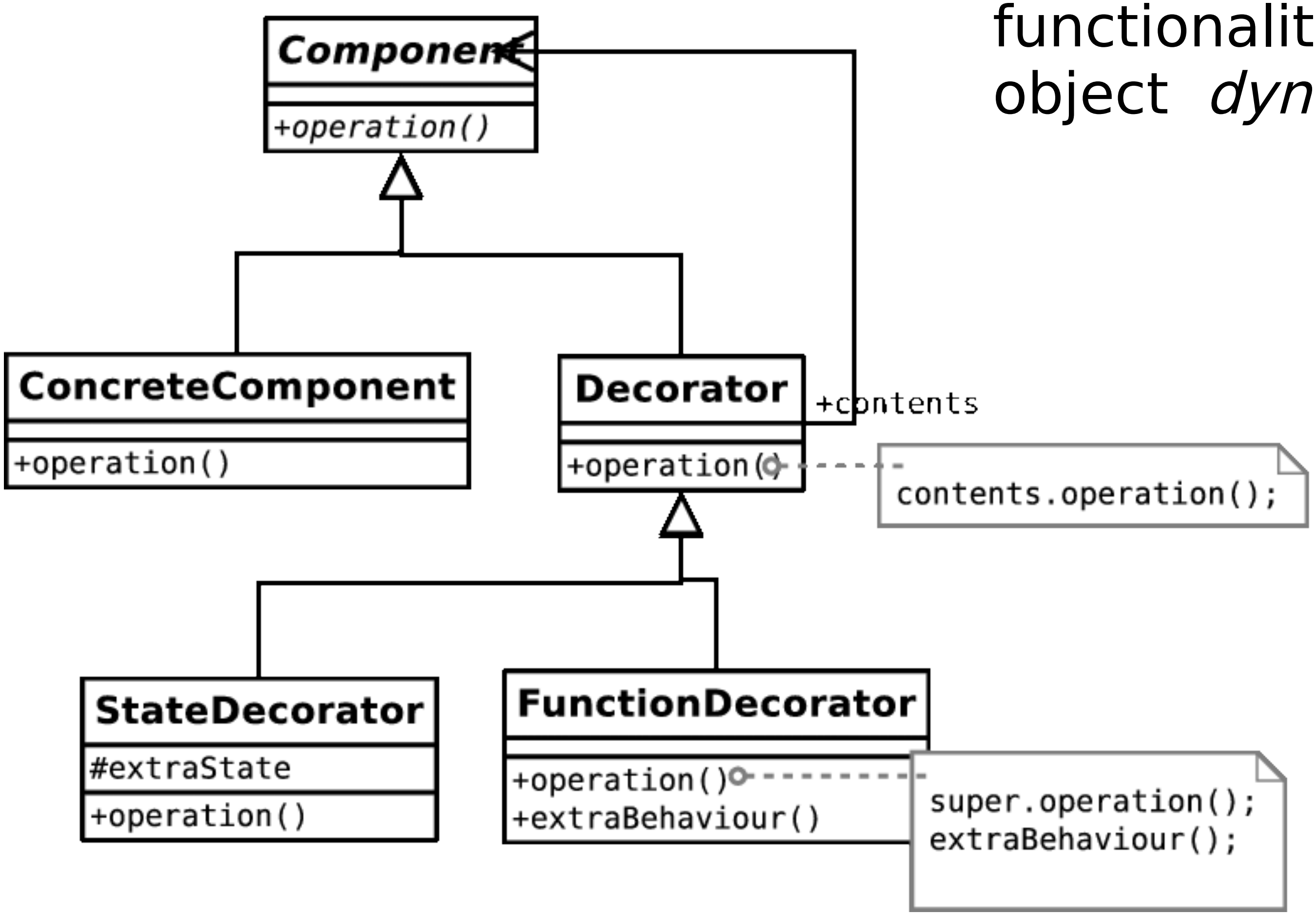
is-a Reader

Reader

```
BufferedReader buff = new BufferedReader(new FileReader("blah"));
```

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 Recipe

1. Make constructor private

2. Create a single static instance

3 Create a static getter for that instance

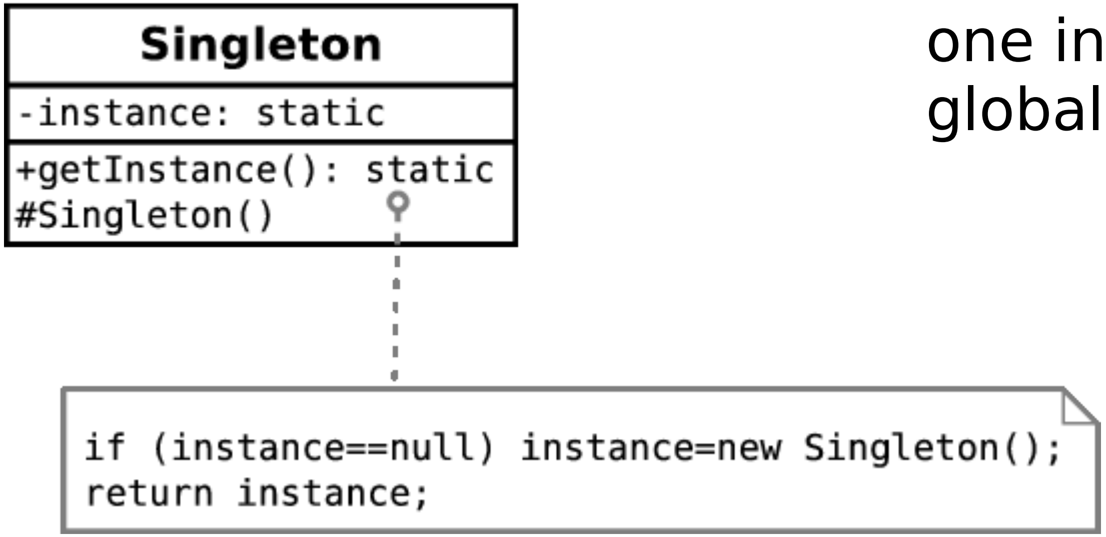
java.util.Runtime

Runtime.getRuntime()

- Singleton representing JVM status

Singleton in General

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



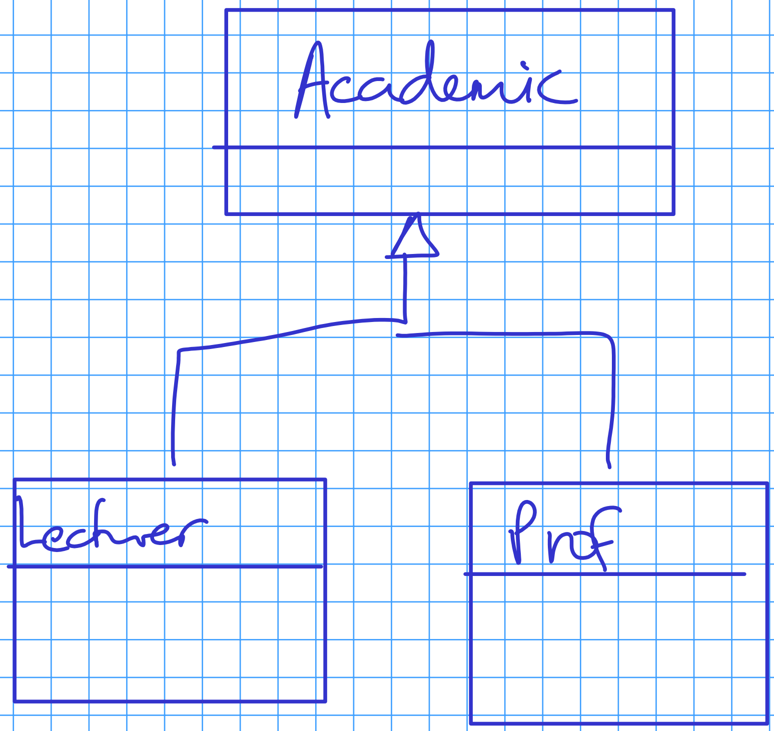
```
if (instance==null) instance=new Singleton();
return instance;
```

State

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

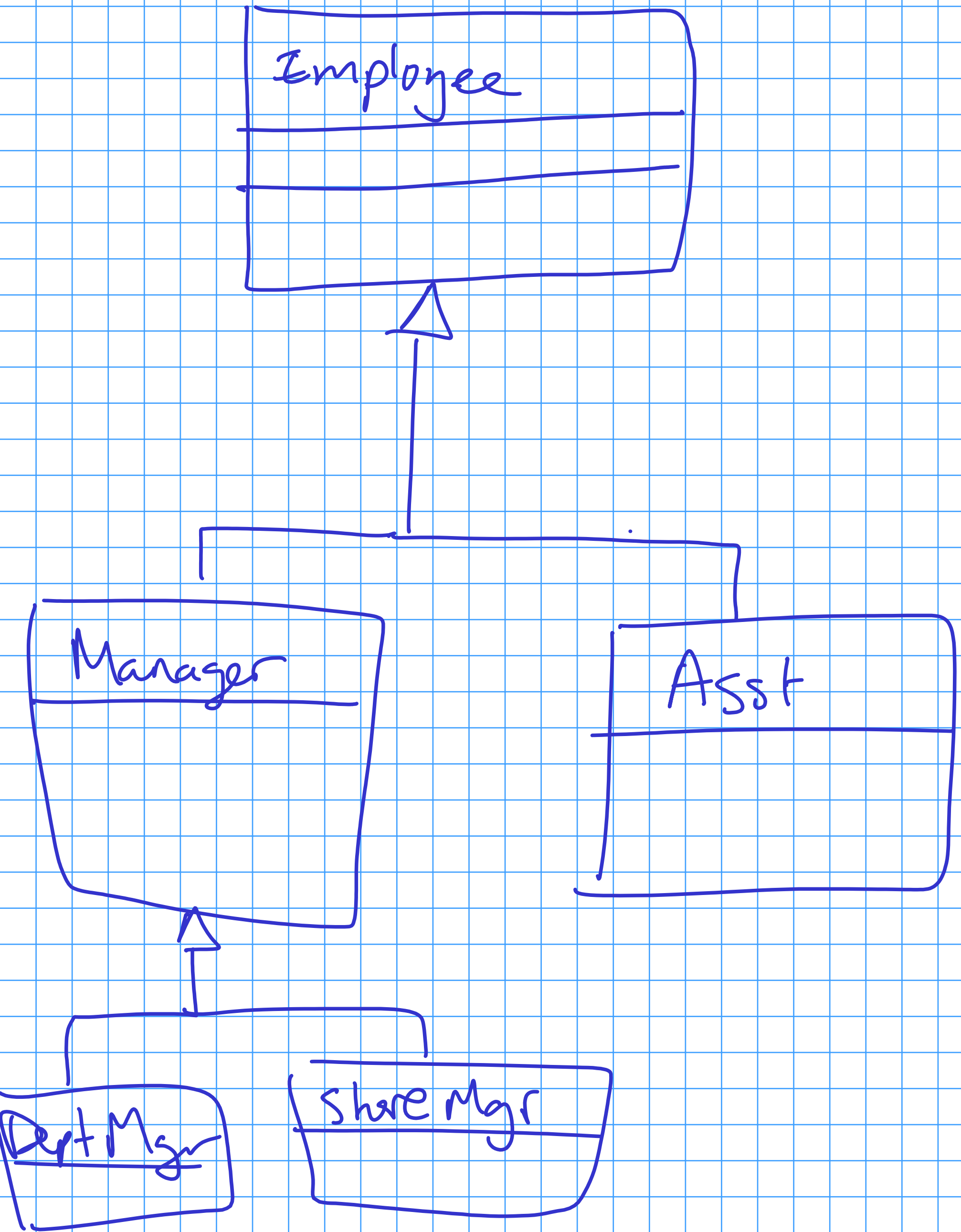
Example problem: Representing academics ~~as~~ if they progress through the rank

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



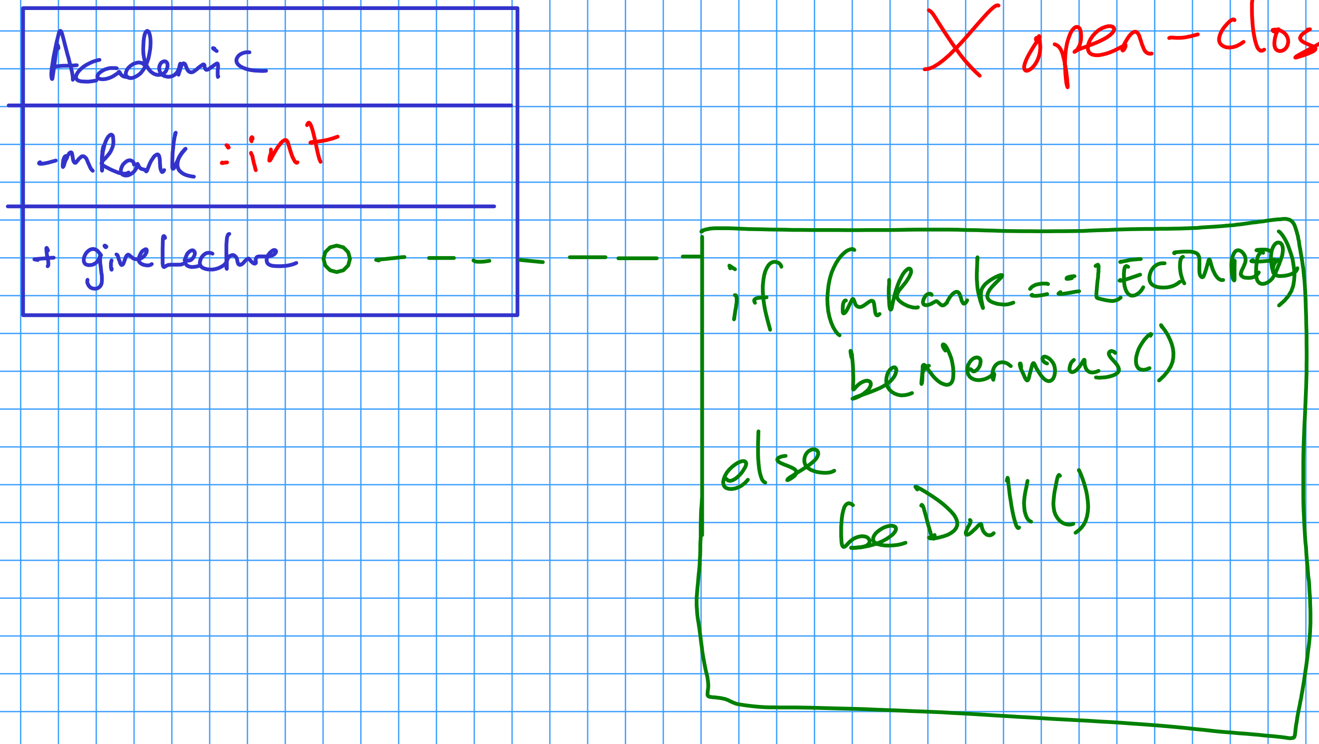
✗ Can't convert objects

✗ Can't guarantee a new object will replace all references to the old one


Employee
Manager
Asst
DeptMgr
ShareMgr

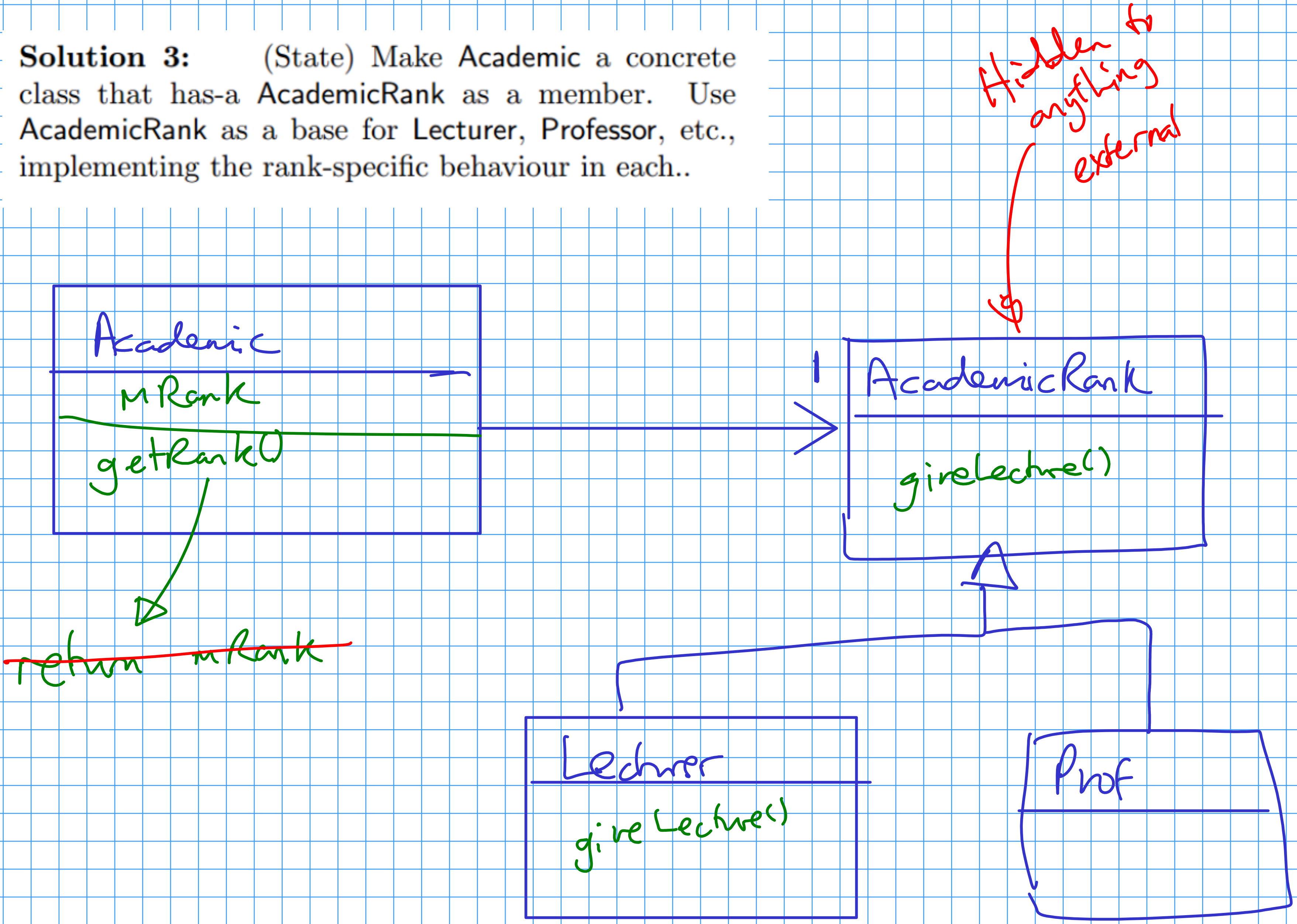
Solution 2: Make Academic a concrete class with a member variable that indicates rank. To get rank-specific behaviour, check this variable within the relevant methods.

X open-closed

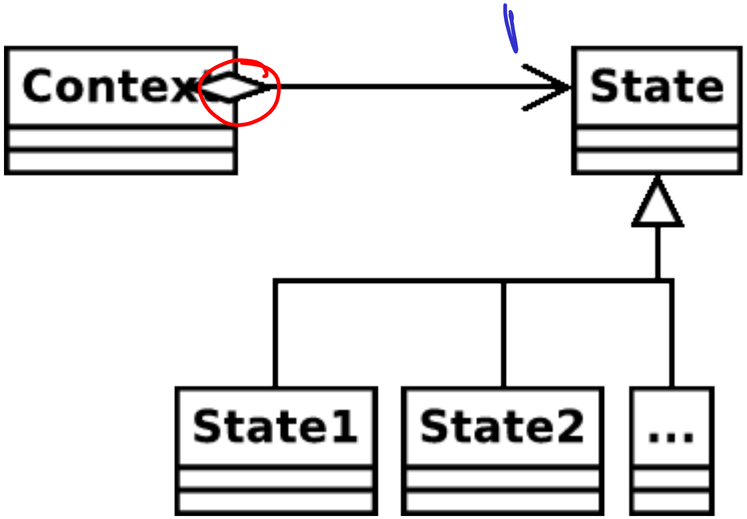


```
if (mRank == LECTURER)
    beNervous()
else
    beDull()
```

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?

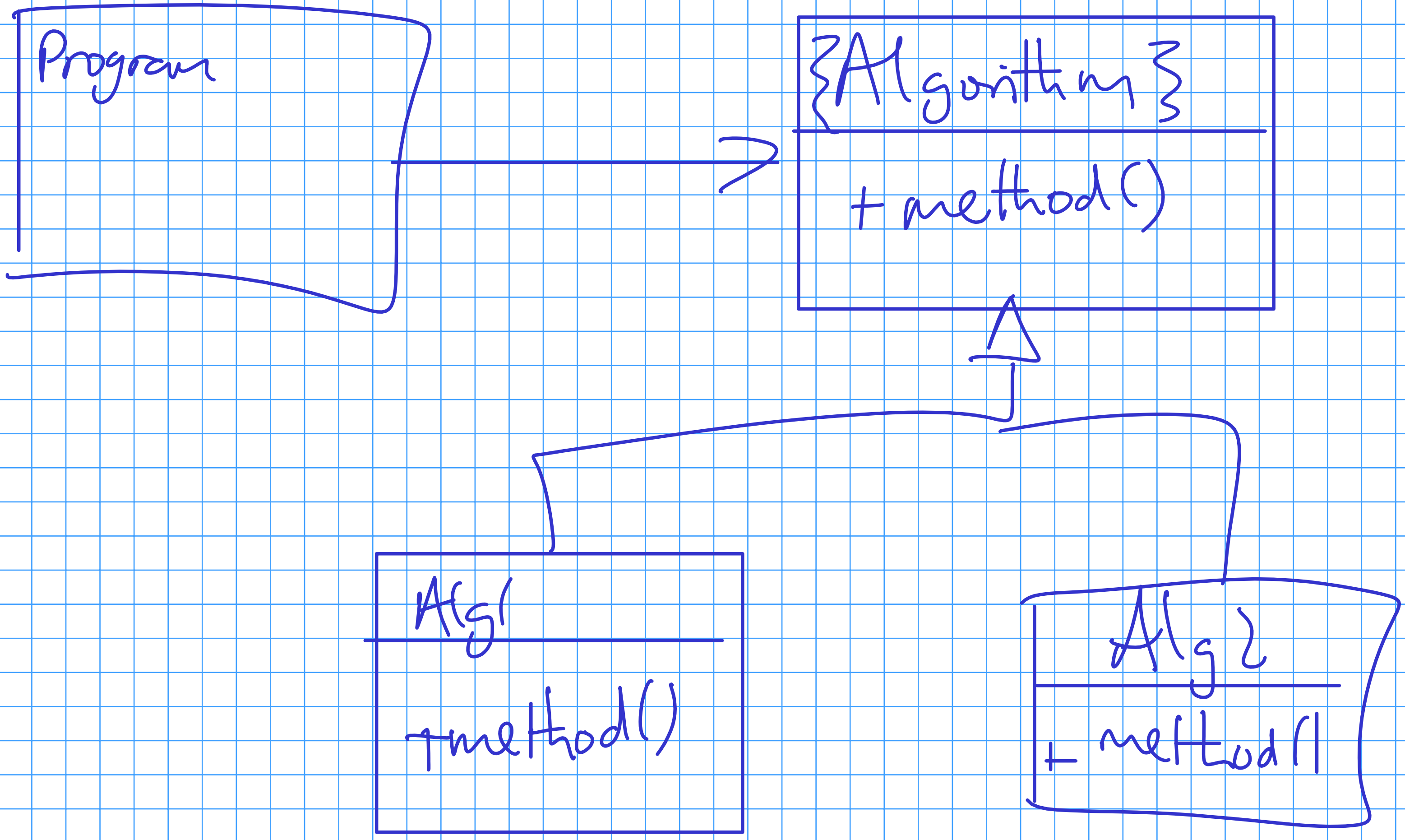
Example problem: We have many possible change-making implementations. How do we cleanly change between them?

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

```
if (method1)
        method1()
else
        method()
```

✗ open-closed

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



State vs Strategy

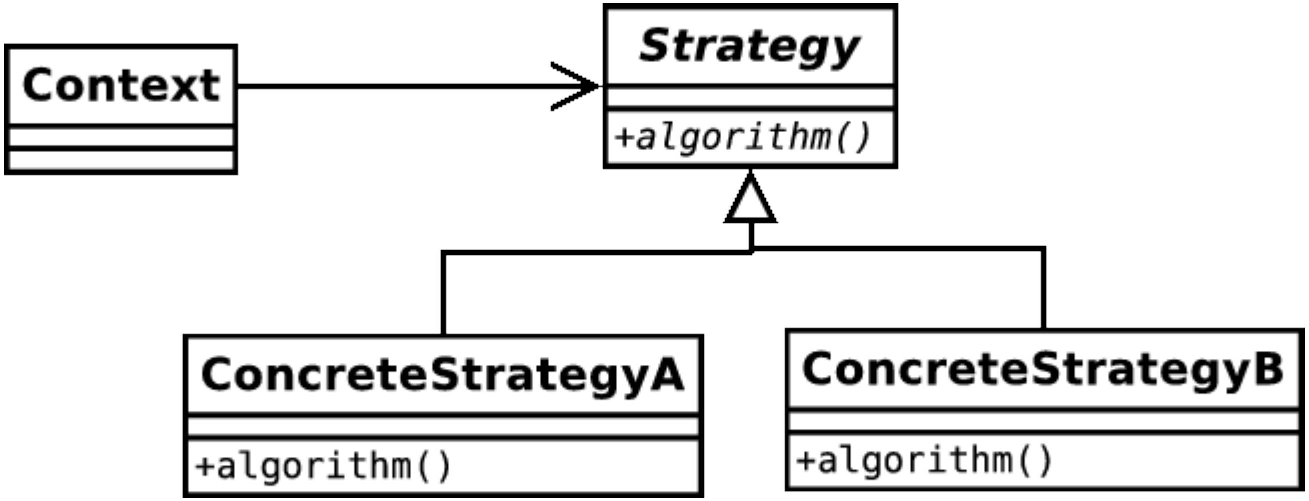
Differ in **intent**

State – Different behaviour depending on current context } Hide the classes

Strategy – Same behaviour but achieved in a different way } Explicit/ exposed

Strategy in General

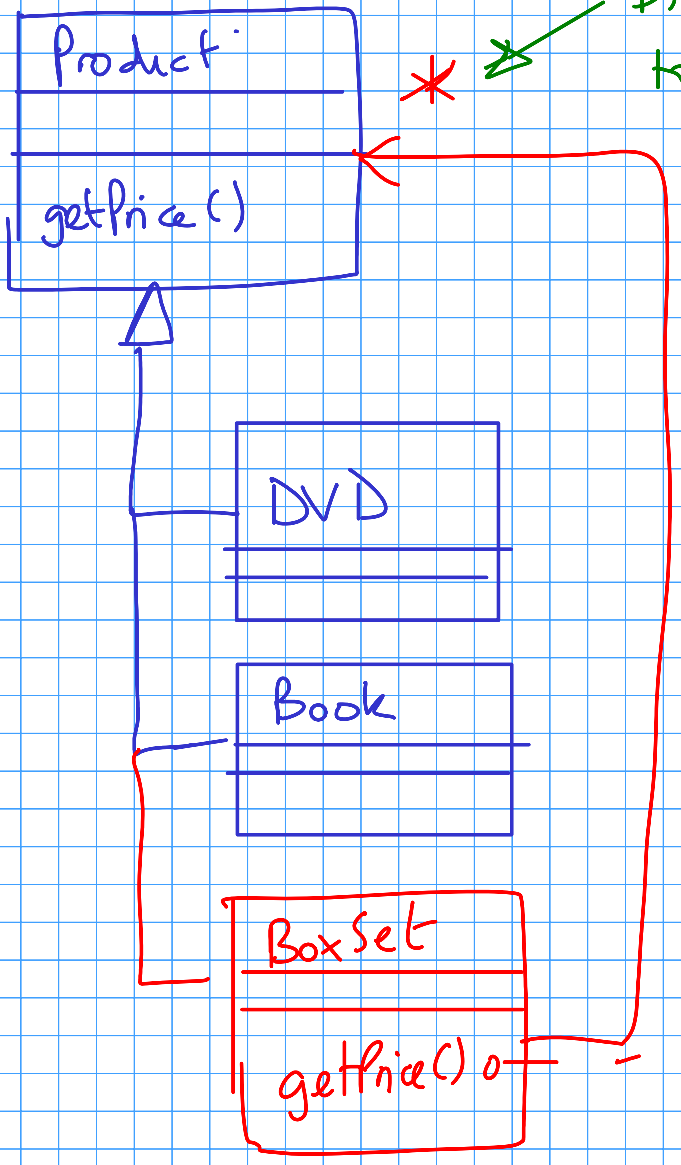
- The strategy pattern allows us to cleanly interchange between algorithm implementations



Composite

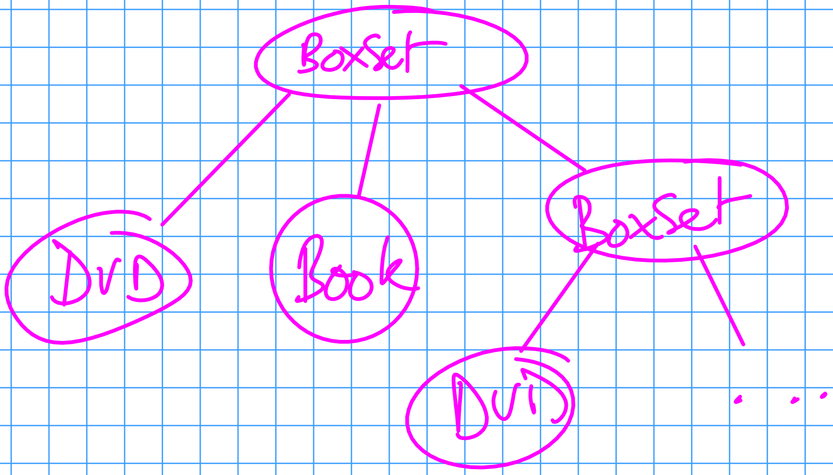
Abstract problem: How can we treat a group of objects as a single object?

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

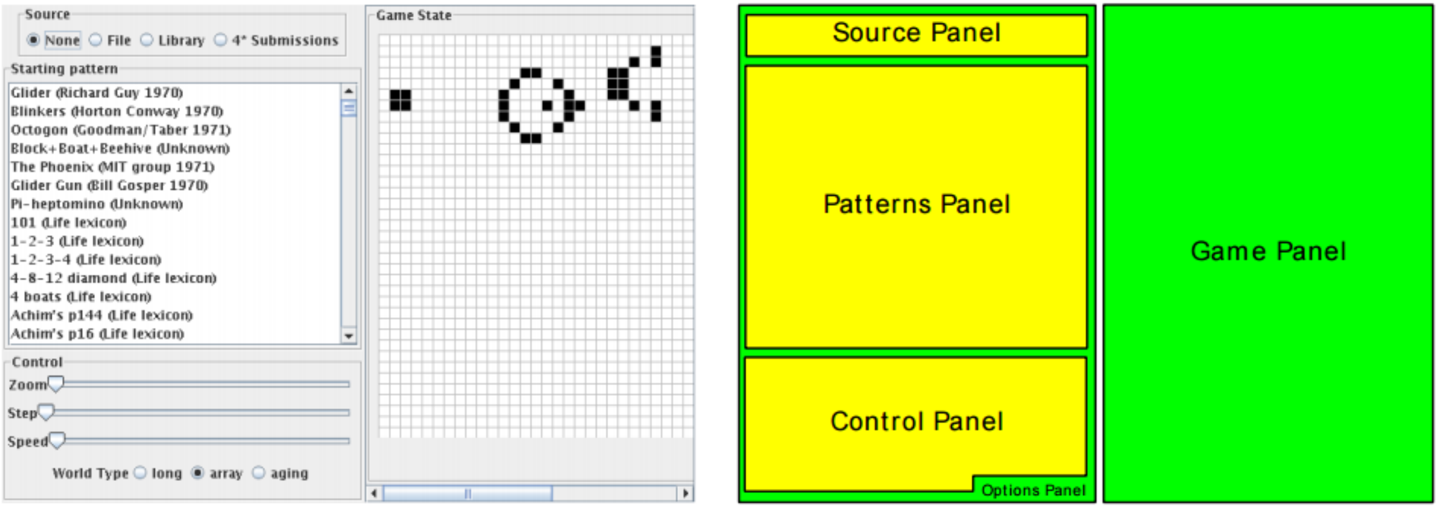


Difference compared to Decorator

⇒ Representing a tree



```
sum = 0
for (Product p : mProds)
    sum += p.getPrice()
return sum * 0.9;
```



Source

● None ○ File ○ Library ○ 4* Submissions

Starting pattern

Glider (Richard Guy 1970)
Blinkers (Horton Conway 1970)
Octogon (Goodman/Taber 1971)
Block+Boat+Beehive (Unknown)
The Phoenix (MIT group 1971)
Glider Gun (Bill Gosper 1970)
Pi-heptomino (Unknown)
101 (Life lexicon)
1-2-3 (Life lexicon)
1-2-3-4 (Life lexicon)
4-8-12 diamond (Life lexicon)
4 boats (Life lexicon)
Achim's p144 (Life lexicon)
Achim's p16 (Life lexicon)

Control

Zoom

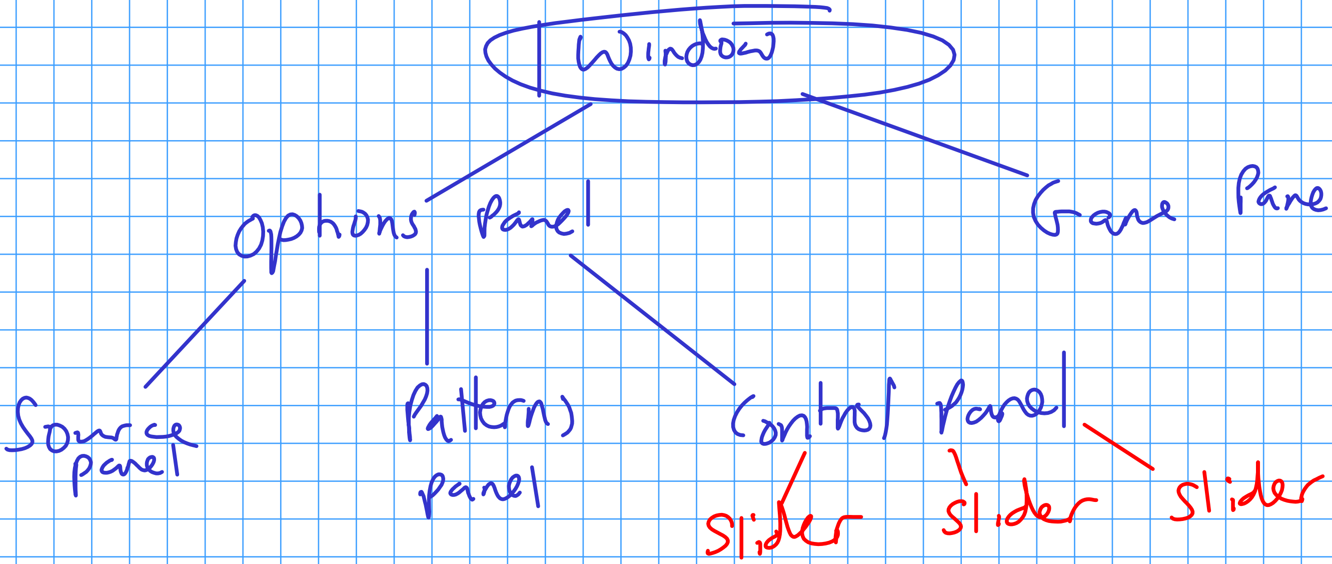
Step

Speed

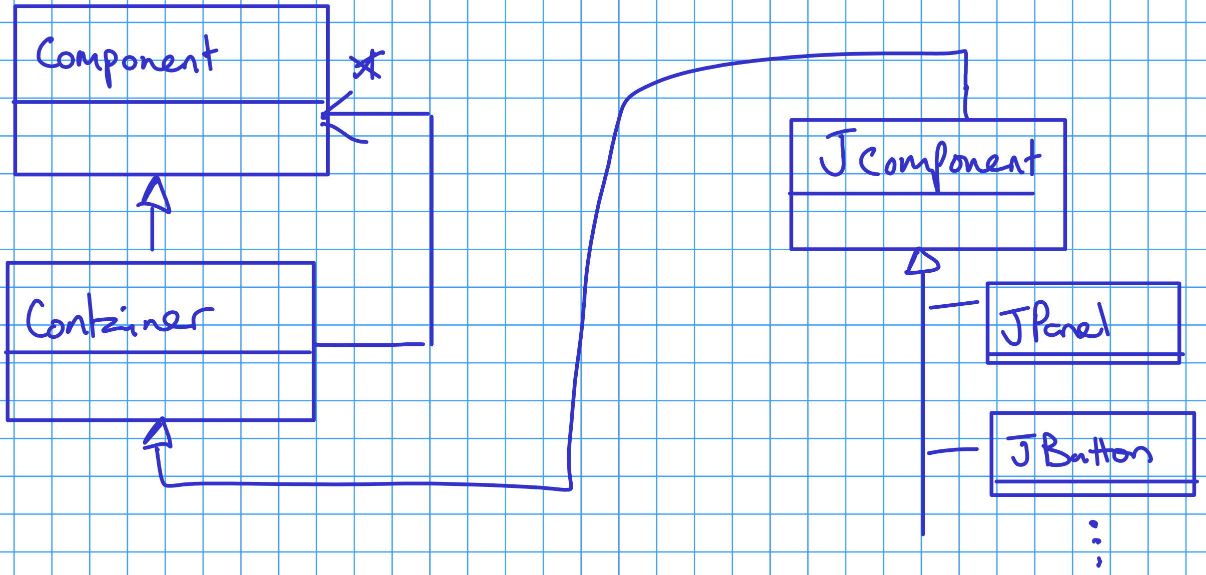
World Type ○ long ● array ○ aging

Game State

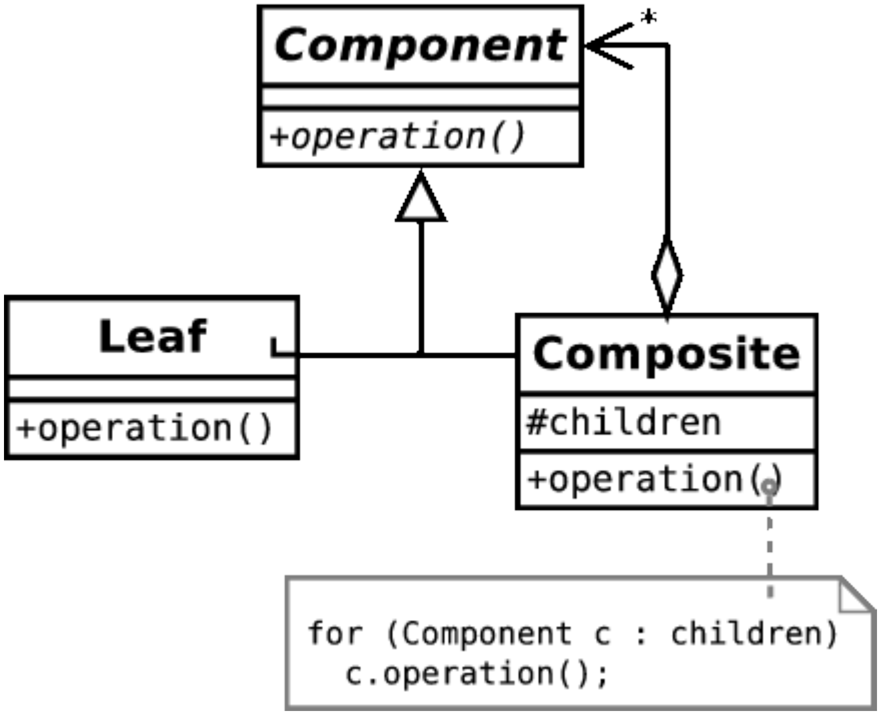
Figure 1. Layout of GUI for the Game of Life; the left shows the GUI, and the right shows the names of the main interface containers



Swing



Composite in General



```
for (Component c : children)
  c.operation();
```

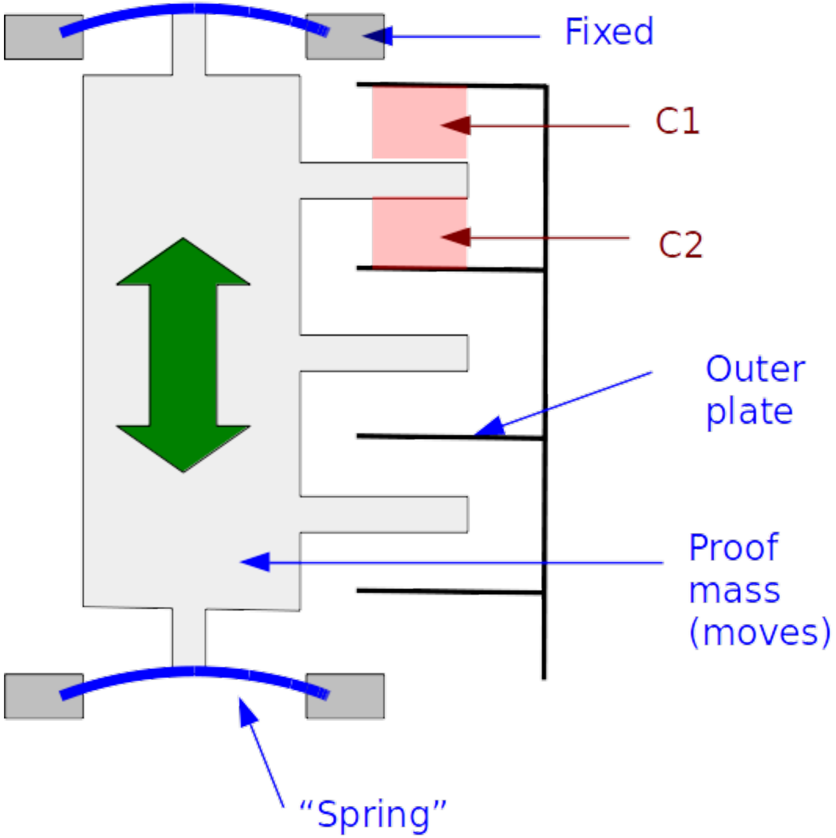
- The composite pattern lets us treat objects and groups of objects uniformly

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?

publish - subscribe



Fixed

C1

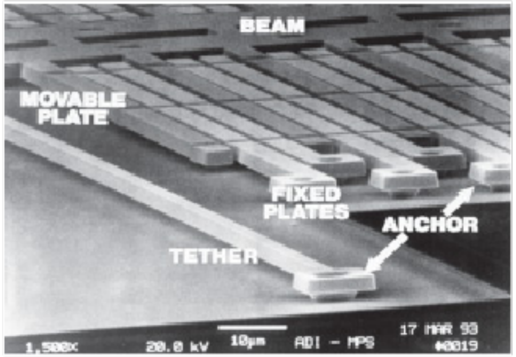
C2

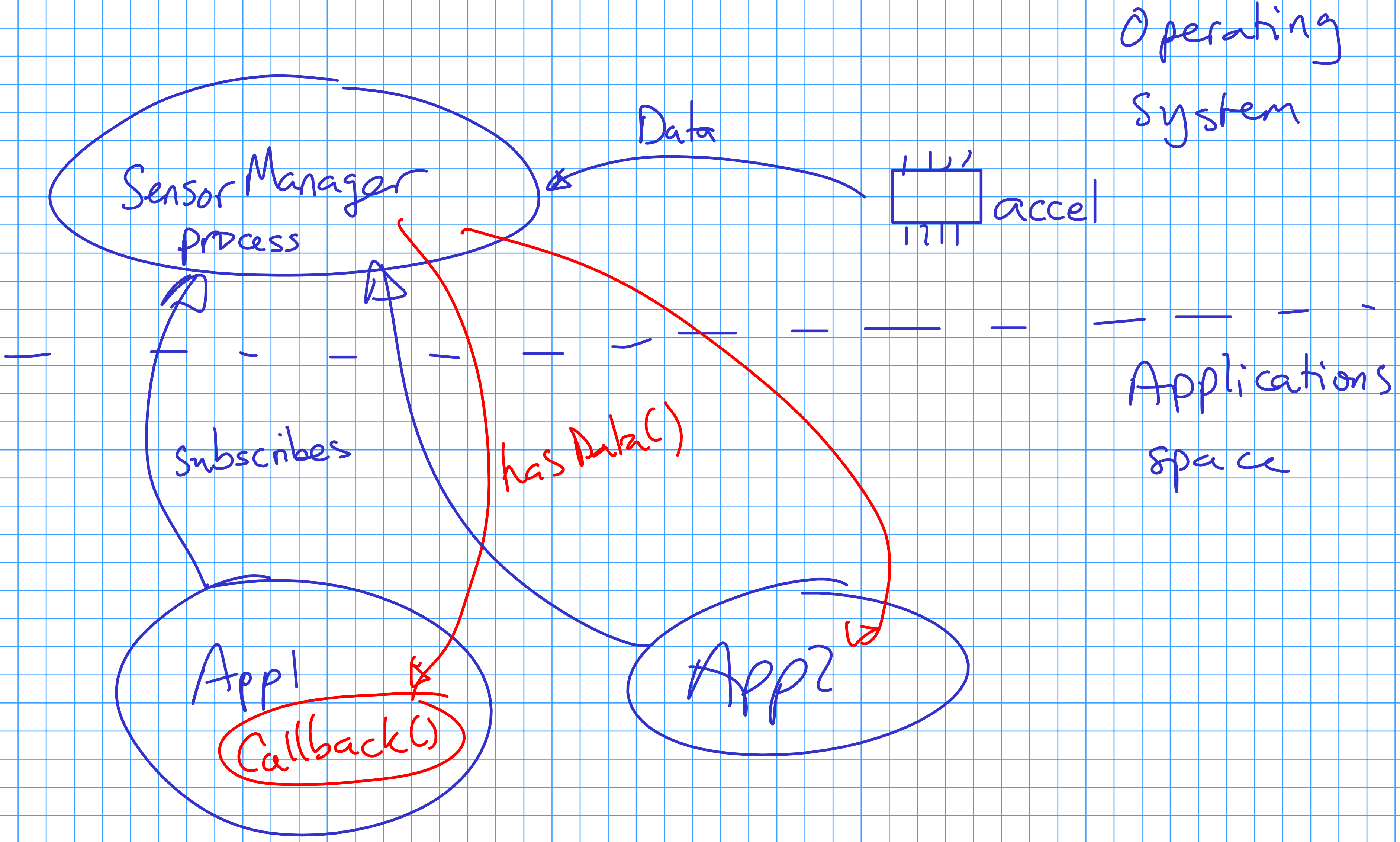
Outer plate

Proof mass (moves)

“Spring”

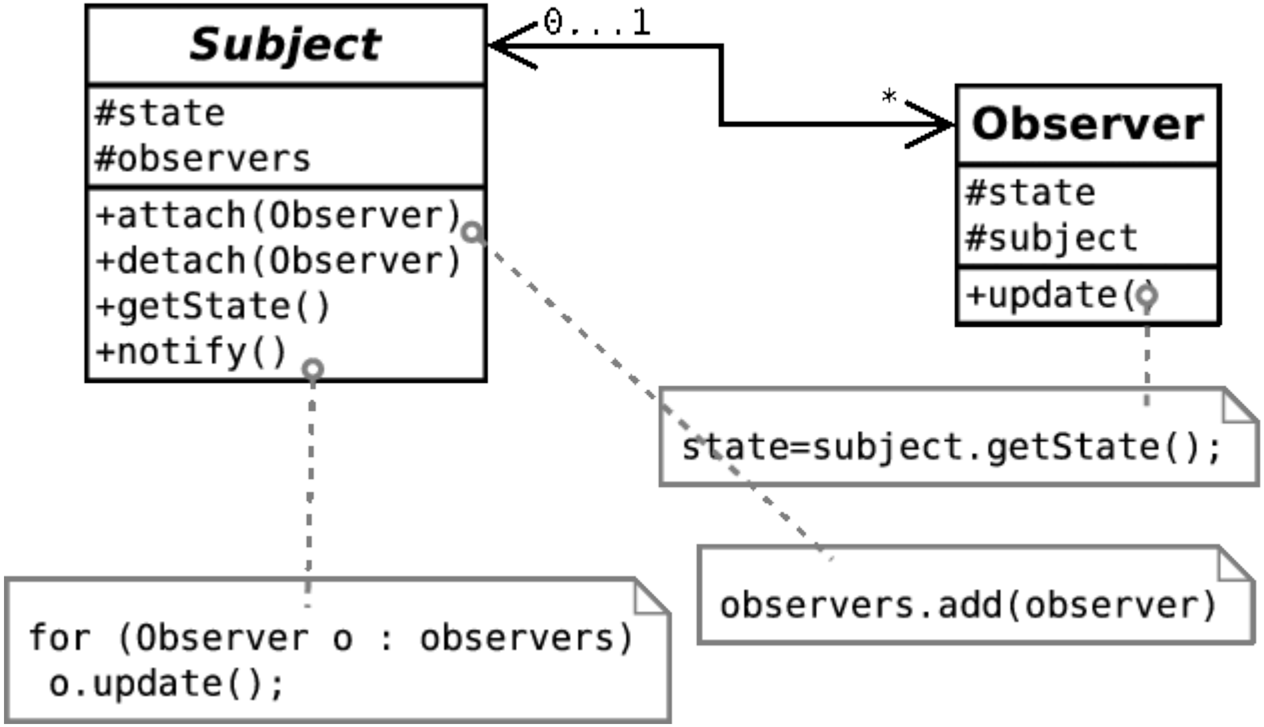
- Have a proof mass between springs and a series of 'plates'
- Measure deflection via capacitance changes
- 1-D only



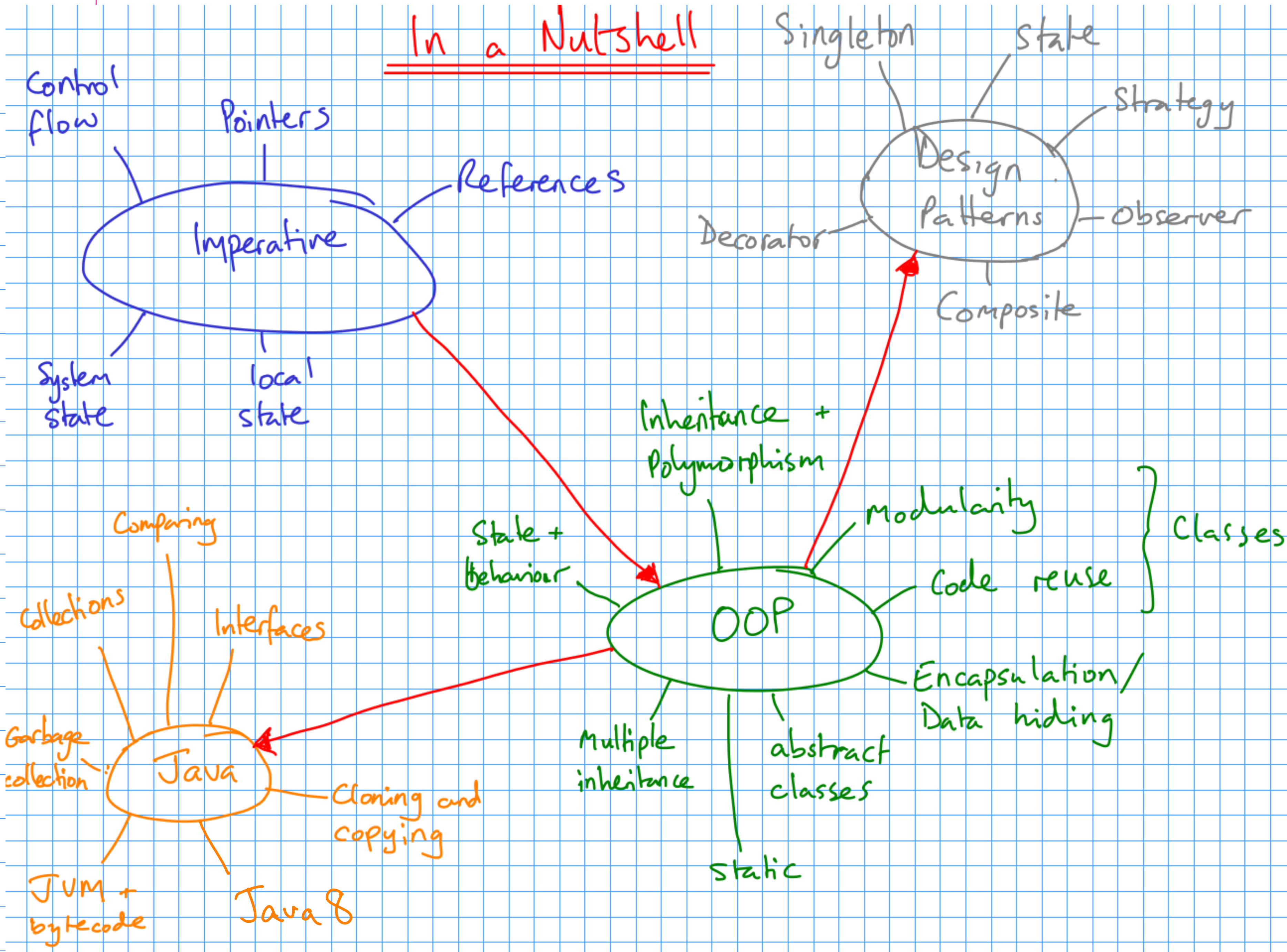

Operating System
Data
accel
Sensor Manager process
subscribes
hasData()
App1
Callback()
App2
Applications space

Observer in General

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



In a Nutshell



MERRY

CHRISTMAS !

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)?

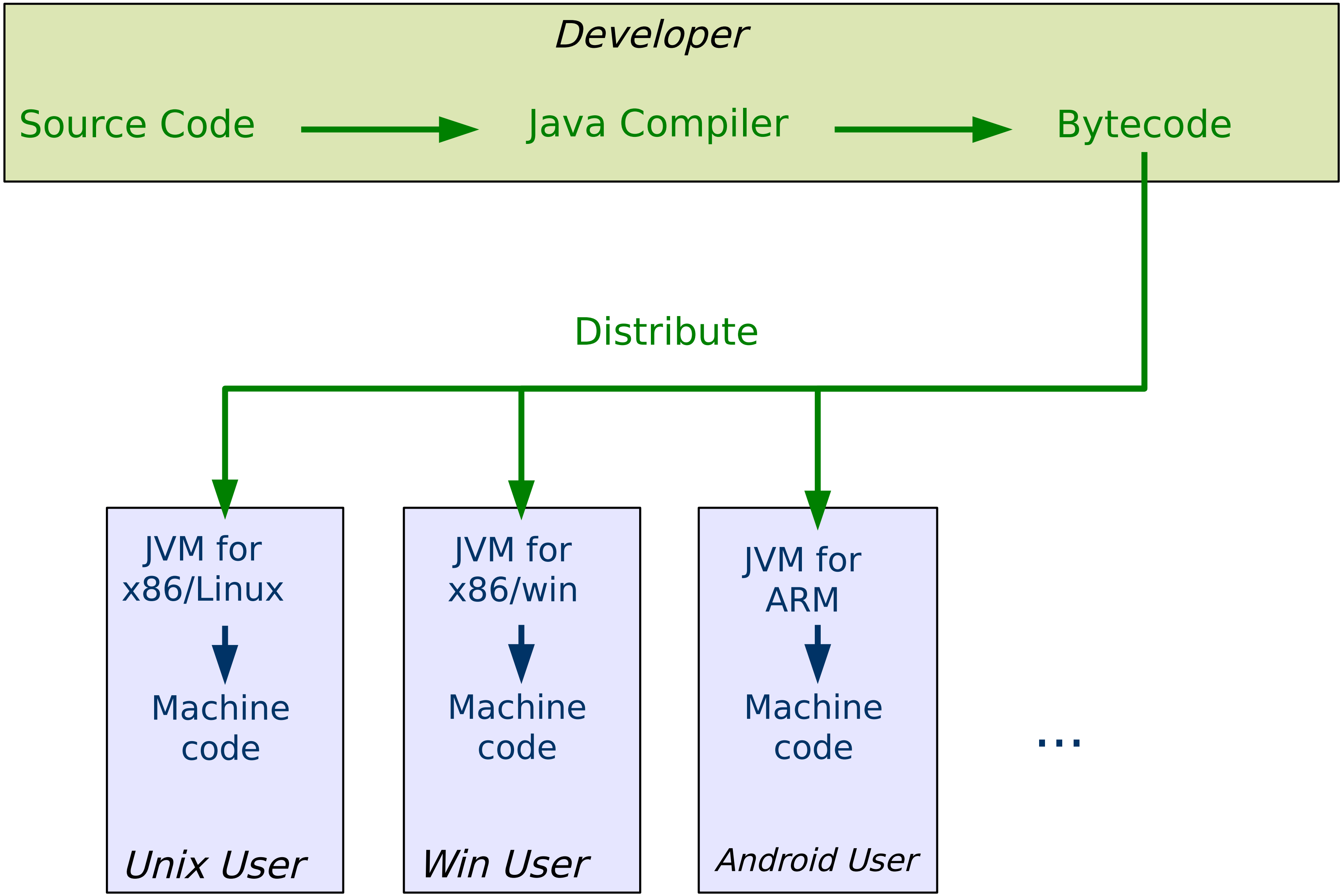
- Could use an interpreter (→ Javascript). But:
  - High level languages not very space-efficient
  - The source code would implicitly be there for anyone to see, which hinders commercial viability.

- 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



Developer

Source Code → Java Compiler → Bytecode

Distribute

JVM for x86/Linux → Machine code

Unix User

JVM for x86/win → Machine code

Win User

JVM for ARM → Machine code

Android User

...

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