Exceptions

Error Handling

- You do a lot on this in your practicals, so we'll just touch on it here
- The traditional way of handling errors is to return a value that indicates success/failure/error

```
public int divide(double a, double b) {
   if (b==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 'codes' are for success, etc.
  - The result can't be returned in the usual way

Exceptions

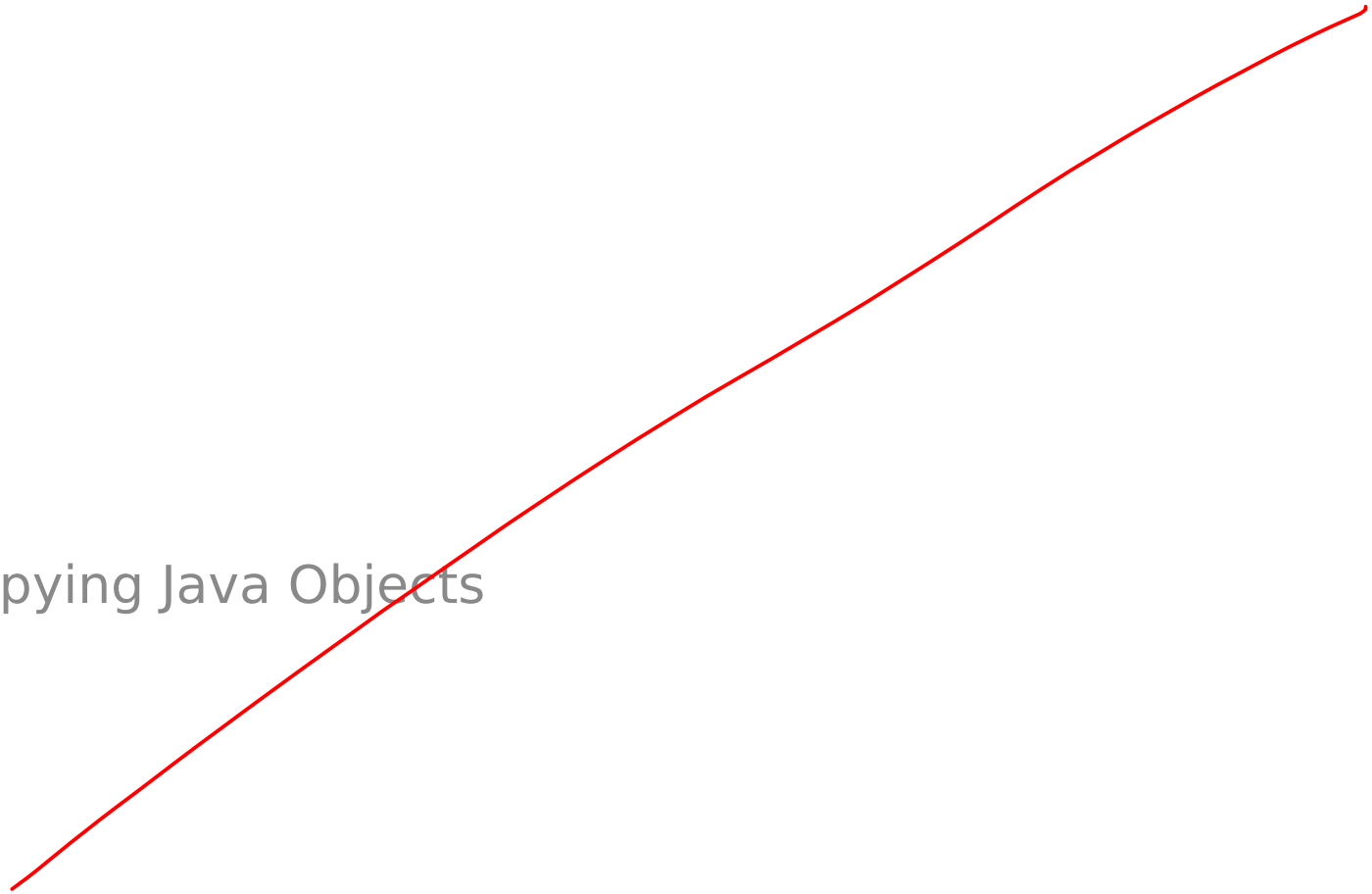
- An exception is an object that can be *thrown* up by a method when an error occurs and *caught* by the calling code

```
public double divide(double a, double b) throws DivideByZeroException {
   if (b==0) throw DivideByZeroException();
   else return a/b;
}

...

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

Exceptions

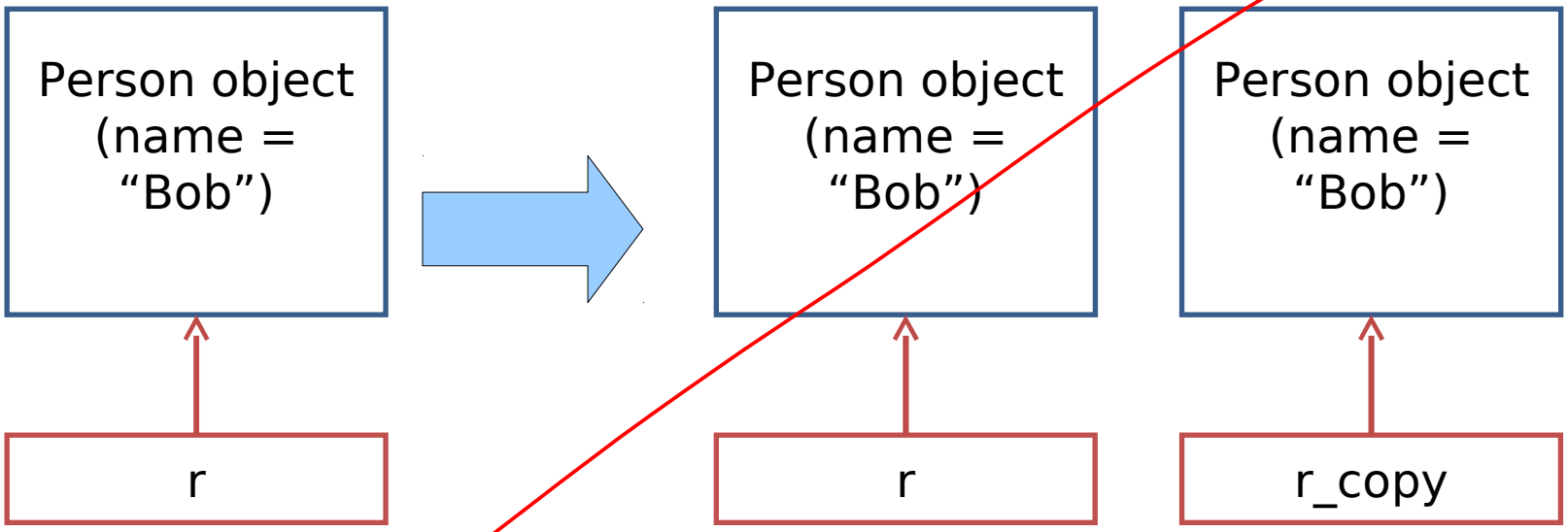
- Advantages:
  - Class name is descriptive (no need to look up codes)
  - Doesn't interrupt the natural flow of the code by requiring constant tests
  - The exception object itself can contain state that gives lots of detail on the error that caused the exception
  - Can't be ignored, only handled

Copying Java Objects



Cloning

- Sometimes we really do want to copy an object

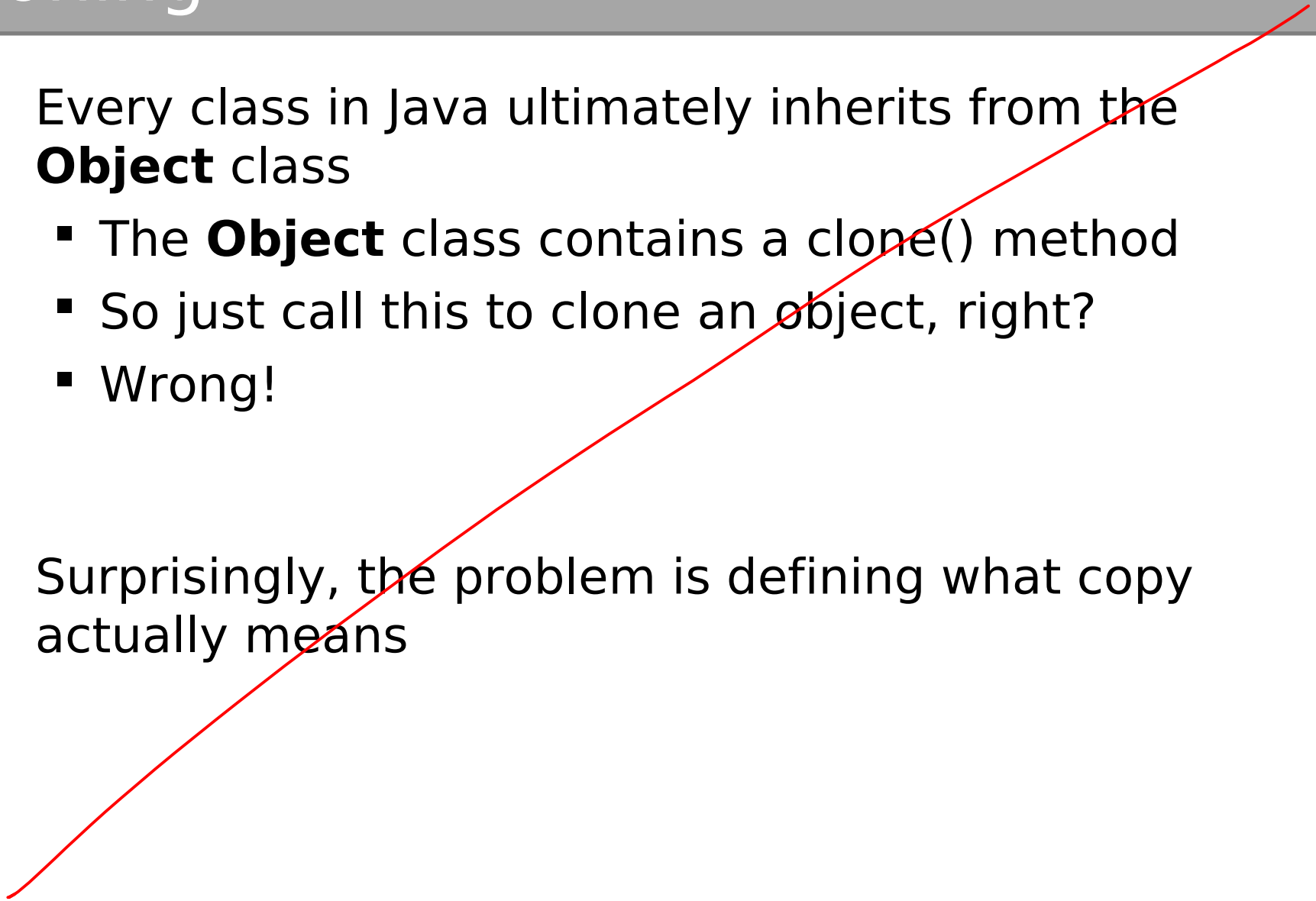


- Java calls this ***cloning***
- We need special support for it

Cloning

- Every class in Java ultimately inherits from the **Object** class
  - The **Object** class contains a clone() method
  - So just call this to clone an object, right?
  - Wrong!

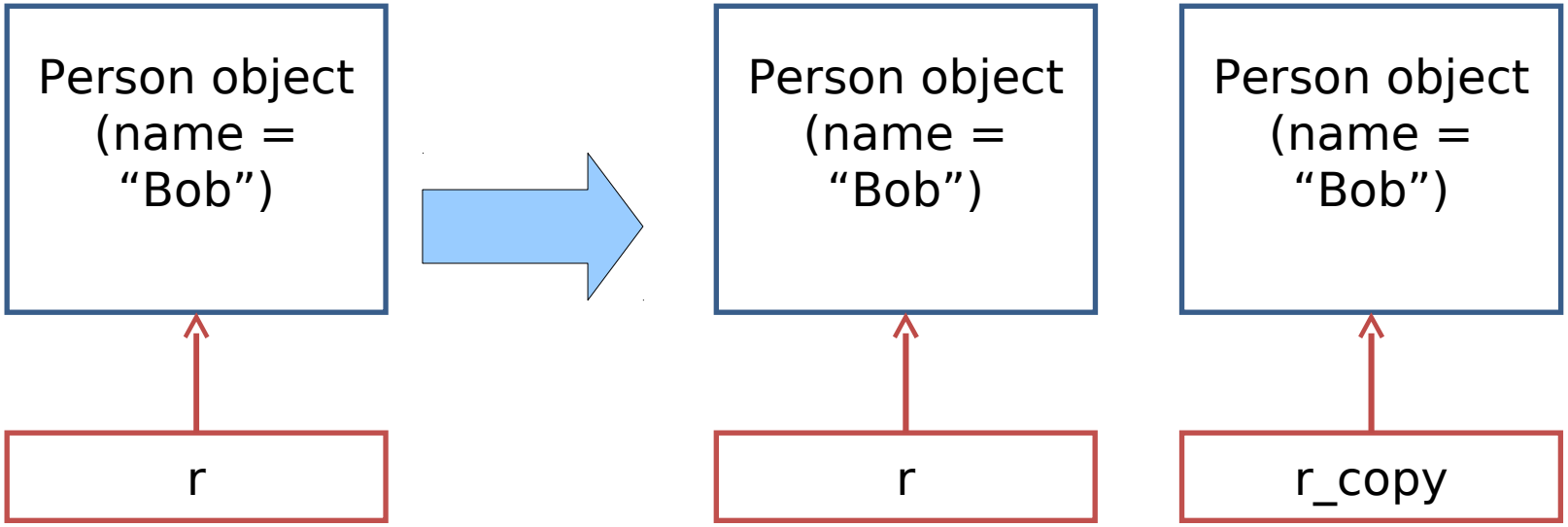
- Surprisingly, the problem is defining what copy actually means



Copying Java Objects

Cloning

- Sometimes we really do want to copy an object



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

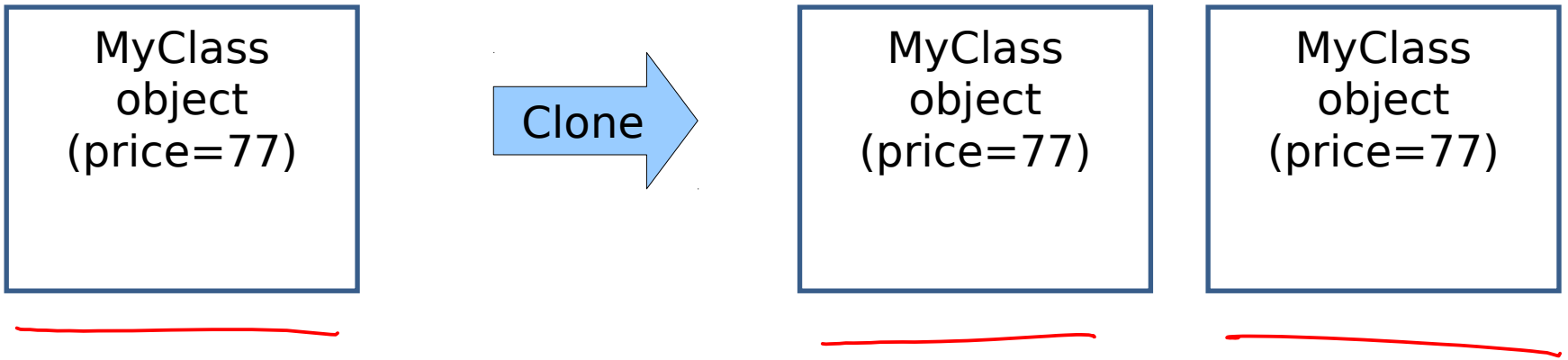
Cloning

- Every class in Java ultimately inherits from the **Object** class
  - The **Object** class contains a clone() method
  - So just call this to clone an object, right?
  - Wrong!

- Surprisingly, the problem is defining what copy actually means

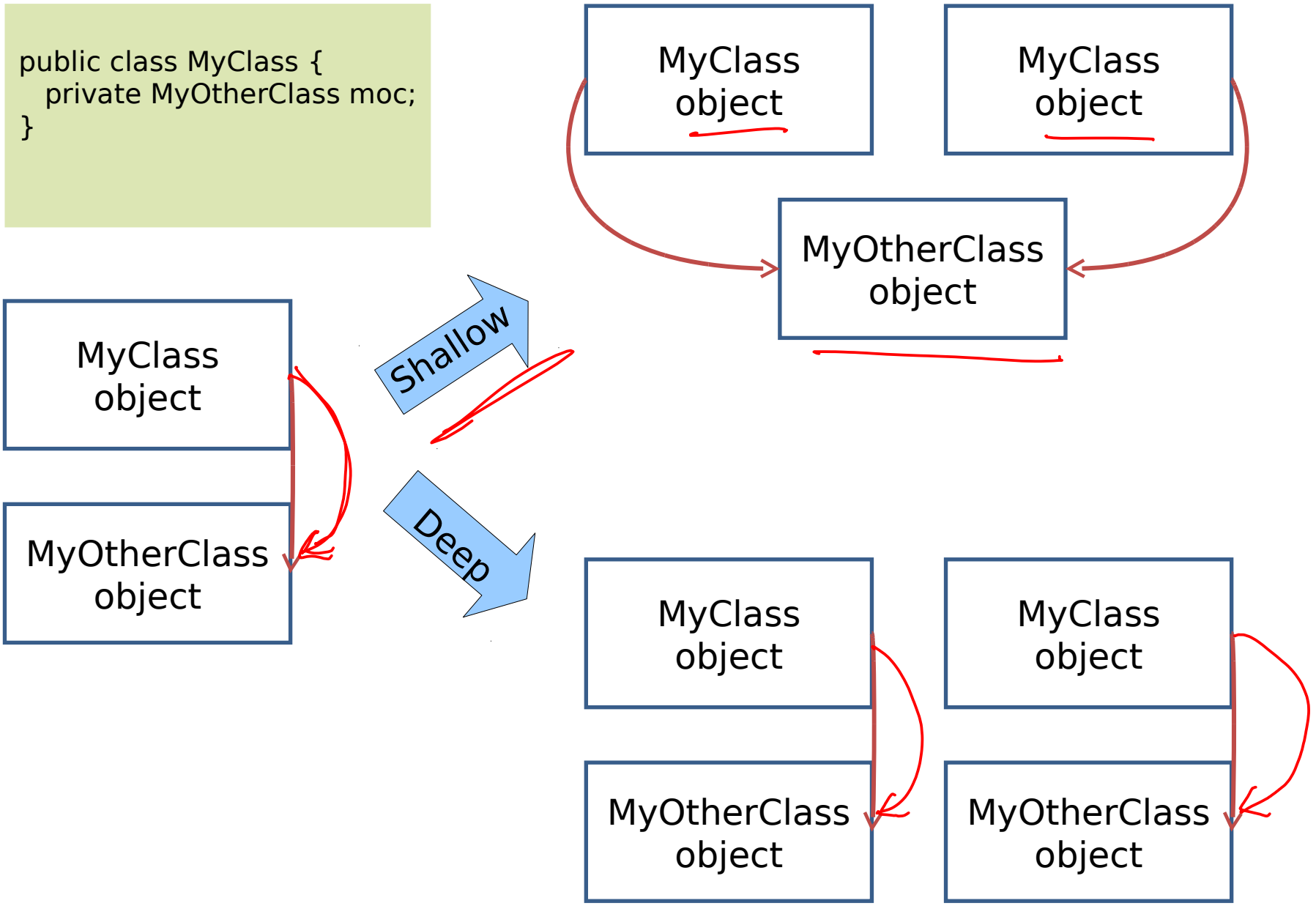
Cloning

```
public class MyClass {
   private float price = 77;
}
```



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

Marker Interfaces

- If you go and 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

Distributing Java Classes

Distributing Classes

- So you've written some great classes that might be useful to others. You release the code. What if you've named your class the same as someone else?
  - E.g. There are probably 100s of “Vector” classes out there..!
- Most languages define some way that you can keep your descriptive class name without getting it confused with others.
- Java uses **packages**. A class belongs to a package
  - A nameless 'default' package unless you specify otherwise
  - You're supposed to choose a package name that is unique.
    - Sun decided you should choose your domain name
    - You do have your own domain name, right? ;)

Distributing Classes

```
package uk.cam.ac.rkh23;

import uk.cam.ac.abc21.*;

Class Whatever {
...
}
```

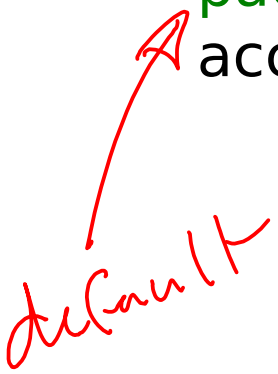
Class Whatever is part of this package

Import all the Classes from some other package

- You get to do lots more about this in your practicals

Access Modifiers Revisited

- Most Languages:
  - public – everyone can access directly
  - protected – only subclasses can access directly
  - private – nothing can access directly
- Java adds:
  - package – anything in the same package can access directly



The Java Class Libraries

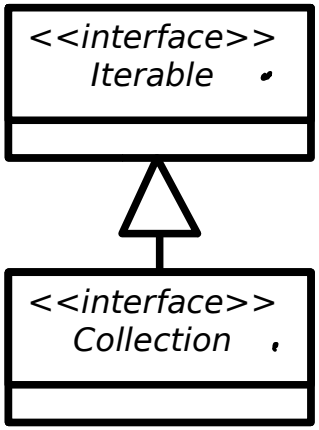
Java Class Library

- 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)

[Arrays]

Java's Collections Framework

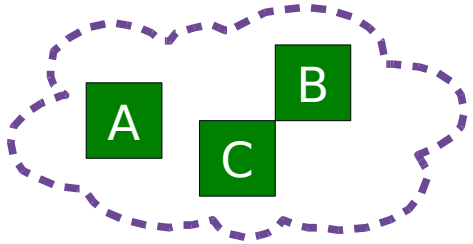


- Important chunk of the class library
- A collection is some sort of grouping of things (objects)
- Usually when we have some grouping we want to go through it ("***iterate*** over it")

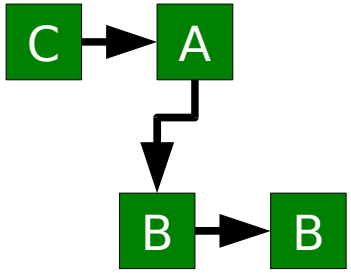
- The Collections framework has two main interfaces: Iterable and Collections. They define a set of operations that all classes in the Collections framework support
- add(Object o), clear(), isEmpty(), etc.

Major Collections Interfaces

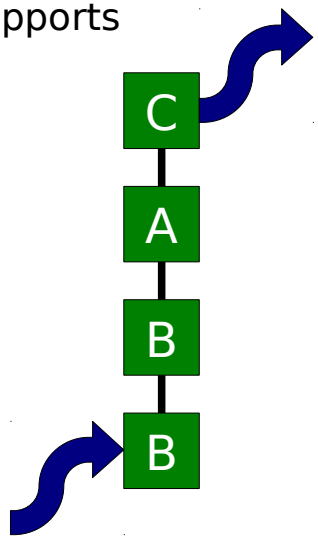
- <<interface>> Set
  - Like a mathematical set in DM 1
  - A collection of elements with no duplicates
  - Various concrete classes like TreeSet (which keeps the set elements sorted)



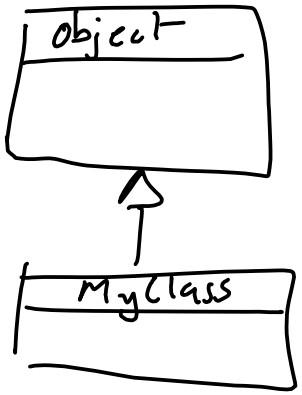
- <<interface>> List
  - An ordered collection of elements that may contain duplicates
  - ArrayList, Vector, LinkedList, etc.



- <<interface>> Queue
  - An ordered collection of elements that may contain duplicates and supports removal of elements from the head of the queue
  - PriorityQueue, LinkedList, etc.

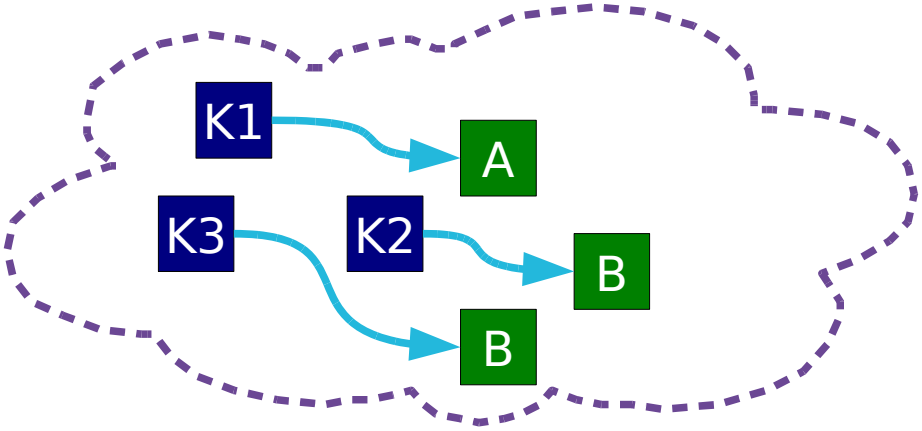


list<int>?



Major Collections Interfaces

- <<interface>> Map
  - Like relations in DM 1
  - Maps key objects to value objects
  - Keys must be unique
  - Values can be duplicated and (sometimes) null.



Generics

- The original Collections framework just dealt with collections of **Object**s
  - Everything in Java “is-a” **Object** so that way our collections framework will apply to any class we like without any special modification.
  - It gets messy when we get something from our collection though: it is returned as an **Object** and we have to do a narrowing conversion to make use of it:

```
// 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;
}
```

Generics

- It gets worse when you realise that the add() method doesn't stop us from throwing in random objects:

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

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

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

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

Generics

- To help solve this sort of problem, Java introduced *Generics* in JDK 1.5
- Basically, this allows us to tell the compiler what is supposed to go in the Collection
- So it can generate an error at compile-time, not run-time

```
// Make a TreeSet of Integers
TreeSet<Integer> ts = new TreeSet<Integer>();

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

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

Won't even compile

No need to cast :-)

Notation in Java API

- Set<E>
- List<E>
- Queue<E>
- Map<K,V>

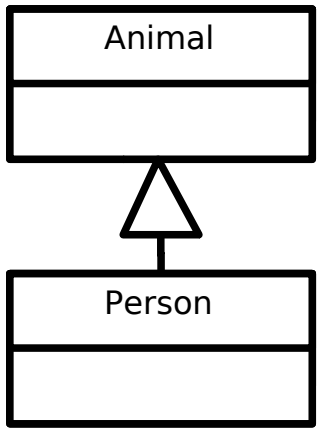
Polymorphism Revisited

- You might recognise Generics as the “polymorphism” you met in FoCS when using ML.
- Both allow you to write code that works for multiple types

  - (Parametric) Polymorphism [FP] or Generics [OOP]
    - The types are determined at compile-time

  - (Sub-type or ad-hoc) Polymorphism [OOP]
    - The types are determined at run-time
    - Needs an inheritance tree

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 **Person**s is a list of **Animal**s, yes?

No

```
BlobFish bf = new BlobFish();
plist.add(bf);
alist.add(bf)
```

Comparing Java Classes

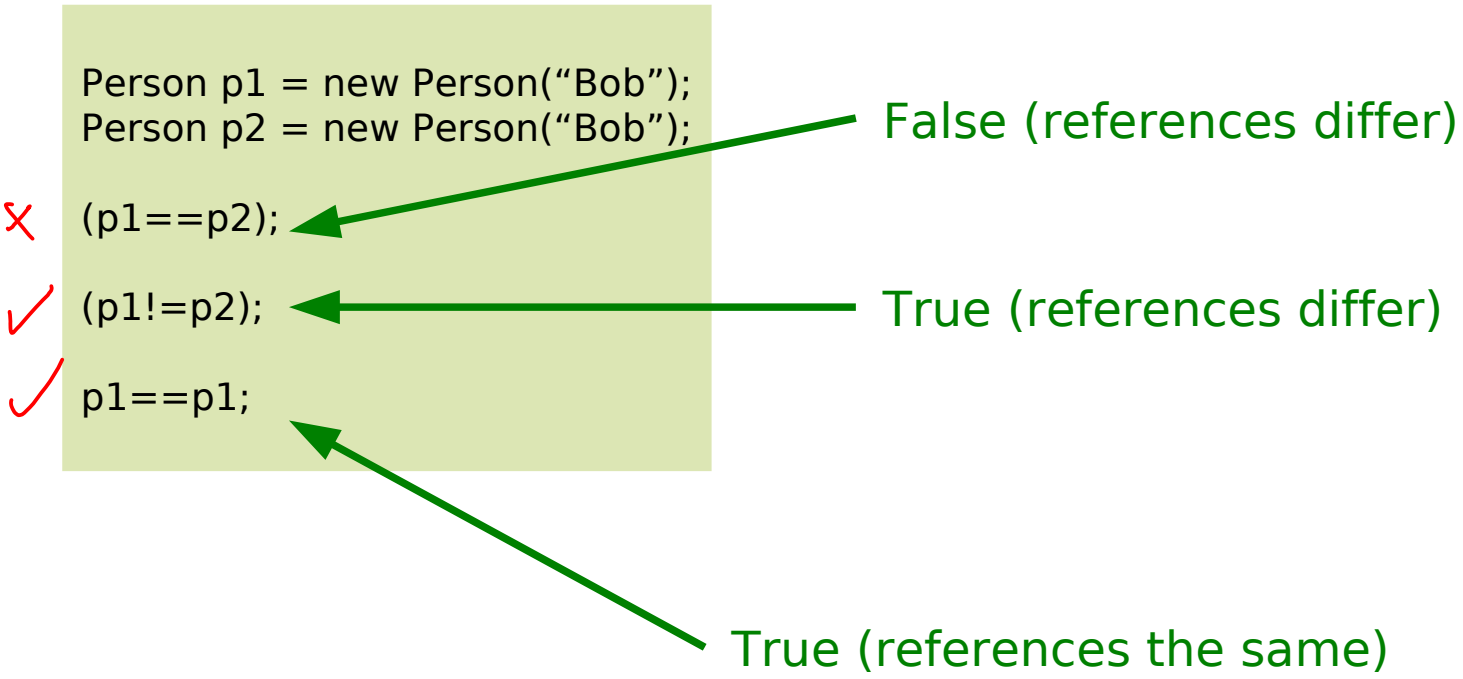
Comparing Primitives

>	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 (object1==object2) mean??
  - Same object?
  - Same state (“value”) but different object?

Option 1: a==b, a!=b

- These compare the *references*




```
String s = "Hello";
if (s=="Hello") System.out.println("Hello");
else System.out.println("Nope");
```

Option 2: The equals() Method

- Object defines an equals() method. By default, this method just does the same as ==.
  - Returns boolean, so can only test equality
  - Override it if you want it to do something different
  - Most (all?) of the core Java classes have properly implemented equals() methods

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

(p1==p2);
p1.equals(p2)
String s1 = "Bob";
String s2 = "Bob";

(s1==s2);
s1.equals(s2);
```

False (we haven't overridden the equals() method so it just compares references

True (String has equals() overridden)

Option 3: Comparable<T> Interface

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

- Part of the Collections Framework
- 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

Option 3: Comparable<T> Interface

```
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>();
```

Option 4: Comparator<T> interface

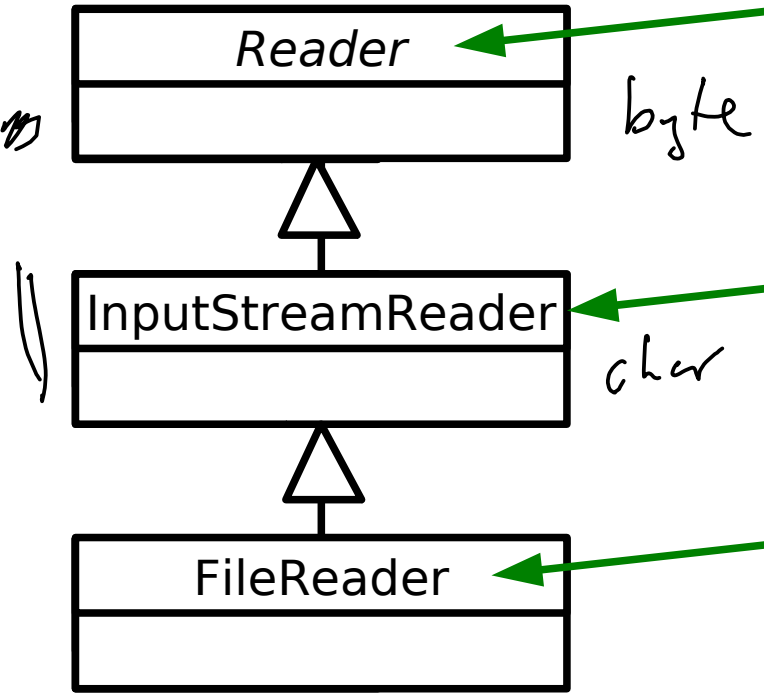
int compareTo(T obj1, T obj2)

- Also part of the Collections framework and allows us to specify a particular comparator for a particular job
- E.g. a Person might have a compareTo() method that sorts by surname. We might wish to create a class AgeComparator that sorts Person objects by age. We could then feed that to a Collections object.

Some Examples...

Java's I/O framework

- Support for system input and output (from/to sources such as network, files, etc).



Abstract class for reading data from some source

Concrete Instance that works on an InputStream object

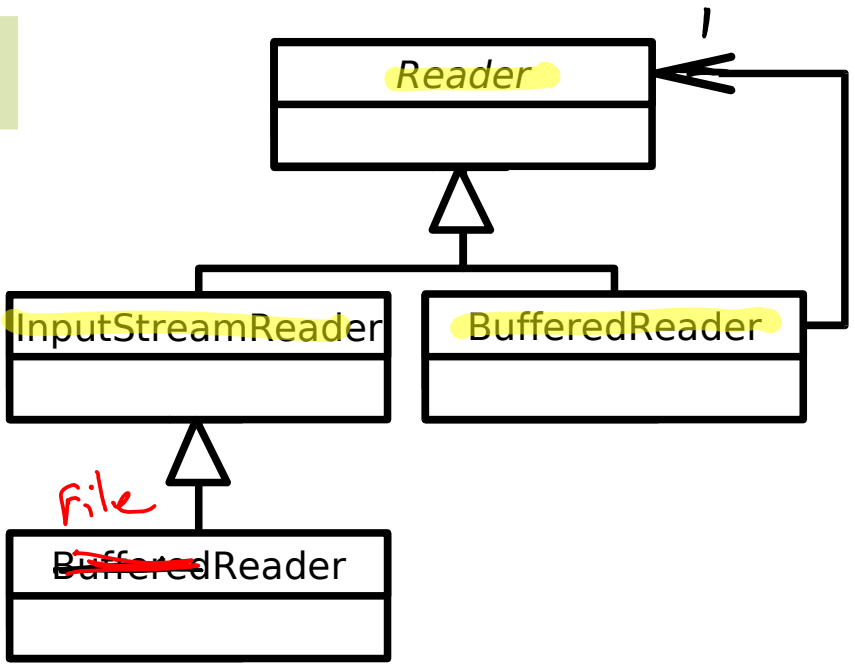
Specialisation that allows us to specify a filename, then creates and InputStream for it

Speeding it up

- In general file I/O is sloowwww
- One trick we can use is that whenever we're asked to read some data in (say one byte) we actually read lots more in (say a kilobyte) and buffer it somewhere on the assumption that it will be wanted eventually and it will just be there in memory, waiting for us. :-)
- Java supports this in the form of a **BufferedReader**

```
FileReader f = new FileReader();
BufferedReader br = new BufferedReader(f);
```

- Whenever we call read() on a BufferedReader it looks in its buffer to see whether it has the data already
- If not it passes the request onto the Reader object
- We'll come back to this...

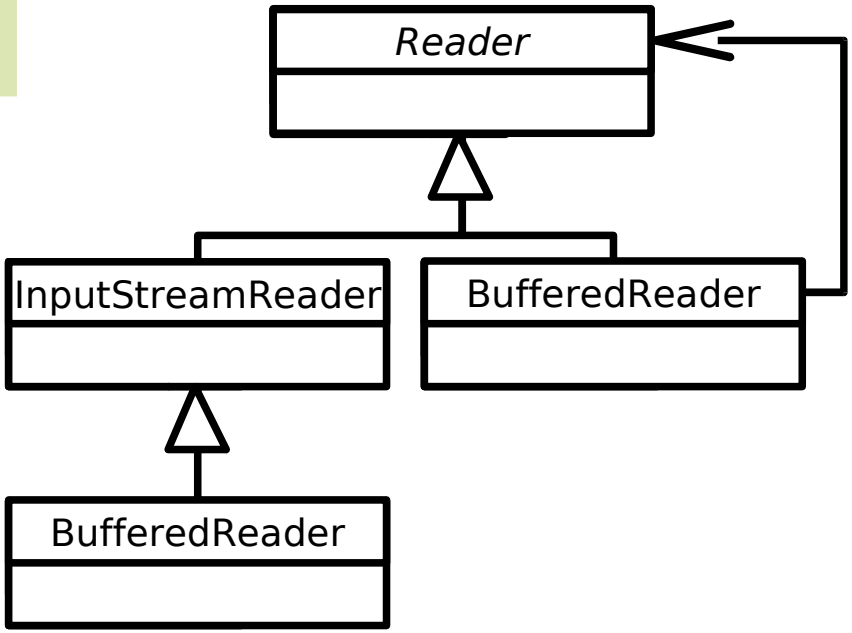


Speeding it up

- In general file I/O is sloowwww
- One trick we can use is that whenever we're asked to read some data in (say one byte) we actually read lots more in (say a kilobyte) and buffer it somewhere on the assumption that it will be wanted eventually and it will just be there in memory, waiting for us. :-)
- Java supports this in the form of a **BufferedReader**

```
FileReader f = new FileReader();
BufferedReader br = new BufferedReader(f);
```

- Whenever we call read() on a BufferedReader it looks in its buffer to see whether it has the data already
- If not it passes the request onto the Reader object
- We'll come back to this...



Speeding it up

- In general file I/O is sloowwww
- One trick we can use is that whenever we're asked to read some data in (say one byte) we actually read lots more in (say a kilobyte) and buffer it somewhere on the assumption that it will be wanted eventually and it will just be there in memory, waiting for us. :-)
- Java supports this in the form of a **BufferedReader**

```
FileReader f = new FileReader();
BufferedReader br = new BufferedReader(f);
```

- Whenever we call read() on a BufferedReader it looks in its buffer to see whether it has the data already
- If not it passes the request onto the Reader object
- We'll come back to this...