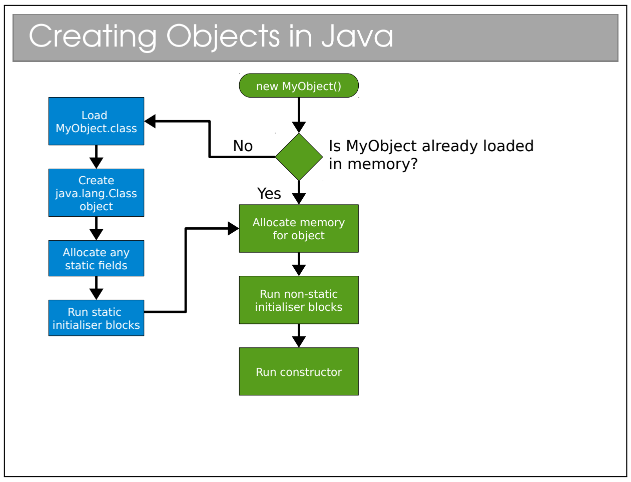
Lecture 6

Lifecycle of an Object

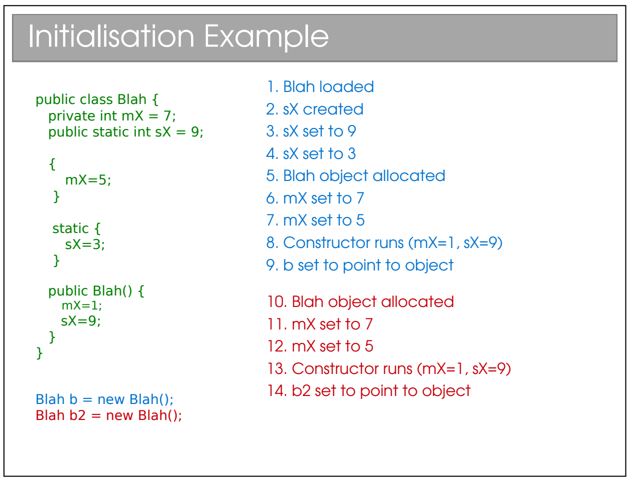
We met constructors earlier in the course as methods that initialise objects. We can now add a bit more detail. When you request a new object, Java will do quite a lot of work:



Creating Objects in Java

new MyObject()
Is MyObject already loaded in memory?
No
Yes
Load MyObject.class
Create java.lang.Class object
Allocate any static fields
Run static initialiser blocks
Allocate memory for object
Run non-static initialiser blocks
Run constructor

Note that Java maintains a java.lang.Class object for every class it loads into memory from a .class file. This object actually allows you query things about the class, such as its name or to list all the methods it has. The ability to do inspect (and possibly modify!) a program's structure is a feature called *reflection*. It's quite a powerful feature that exists in some (but certainly not all) languages. It's out of scope here but worth exploring if you're interested.



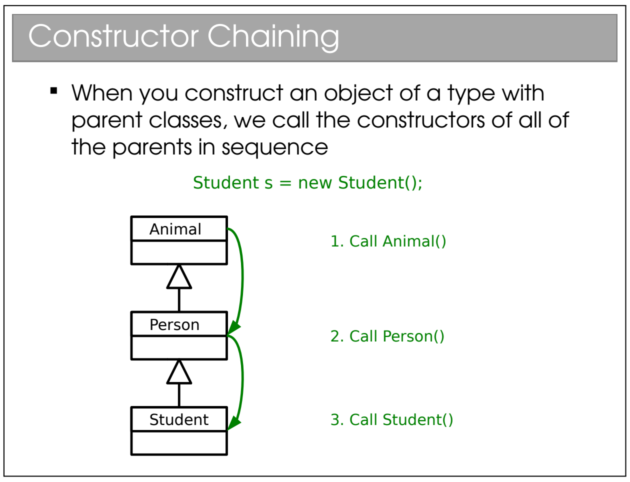
Initialisation Example

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

  {
    mX=5;
  }

  static {
    sX=3;
  }

  public Blah() {
    mX=1;
    sX=9;
  }
}

Blah b = new Blah();
Blah b2 = new Blah();
```

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

10. Blah object allocated
11. mX set to 7
12. mX set to 5
13. Constructor runs (mX=1, sX=9)
14. b2 set to point to object

Things get even more complex when we throw in some inheritance:



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();

In reality, Java asserts that the first line of a constructor *always* starts with super(), which is a call to the parent constructor (which itself starts with super(), etc.). If it does not, the compiler adds one for you:

```
public class Person {
  public Person() {
    }
  }
```

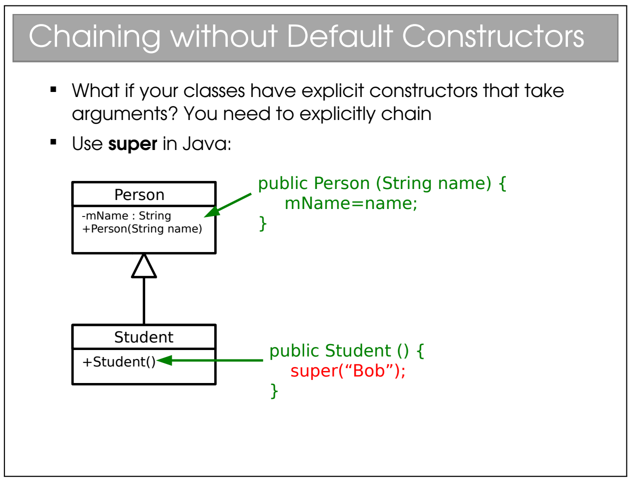
becomes:

```
  public class Person {
    public Person() {
      super();
    }
  }
```

C++

In other languages that support multiple inheritance, this becomes more complex since there may be more than one parent and a simple keyword like super isn't enough. Instead they support manually specifying the constructor parameters for the parents. E.g. for C++:

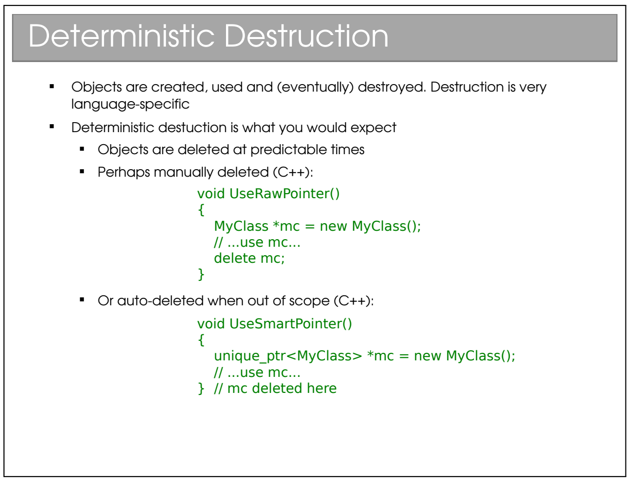
```
  class Child : public Parent1, Parent2 {
    public:
    Child() : Parent1("Alice"),
              Parent2("Bob") {...}
  }
```



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");
}
```



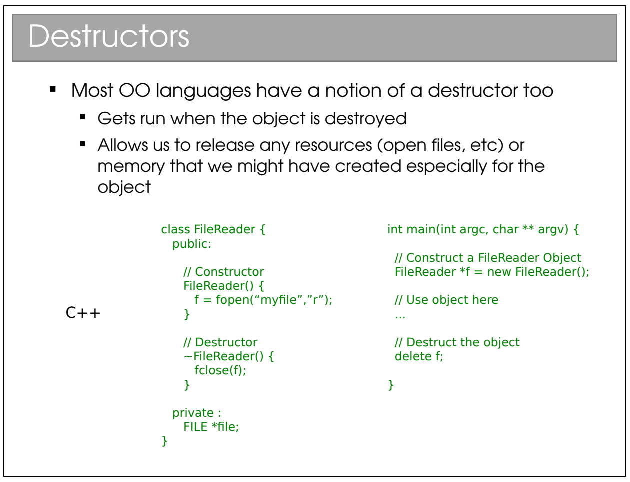
Deterministic Destruction

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

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

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

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



Destructors

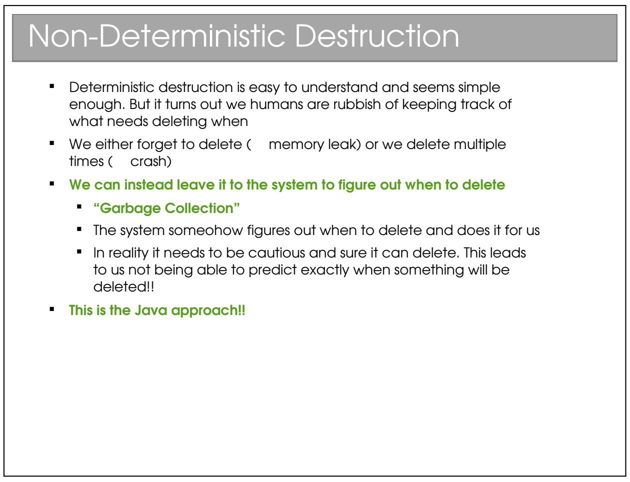
- Most OO languages have a notion of a destructor too
  - Gets run when the object is destroyed
  - Allows us to release any resources (open files, etc) or memory that we might have created especially for the object

C++

```
class FileReader {
  public:

    // Constructor
    FileReader() {
      f = fopen("myfile","r");
    }

    // Destructor
    ~FileReader() {
      fclose(f);
    }

  private :
    FILE *file;
}
```

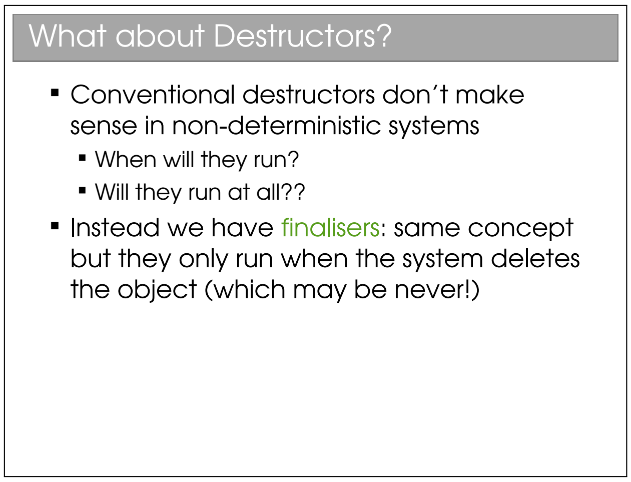
```
int main(int argc, char ** argv) {

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

  // Use object here
  ...

  // Destruct the object
  delete f;

}
```

It will shortly become apparent why I used C++ and not Java for this example.



Non-Deterministic Destruction

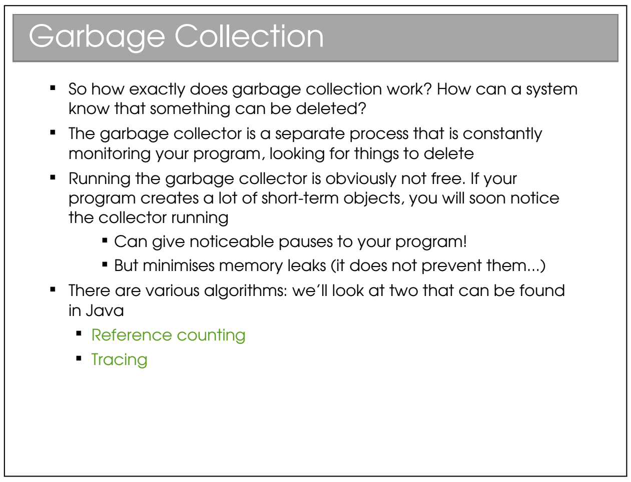
- Deterministic destruction is easy to understand and seems simple enough. But it turns out we humans are rubbish of keeping track of what needs deleting when
- We either forget to delete (memory leak) or we delete multiple times (crash)
- **We can instead leave it to the system to figure out when to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



What about Destructors?

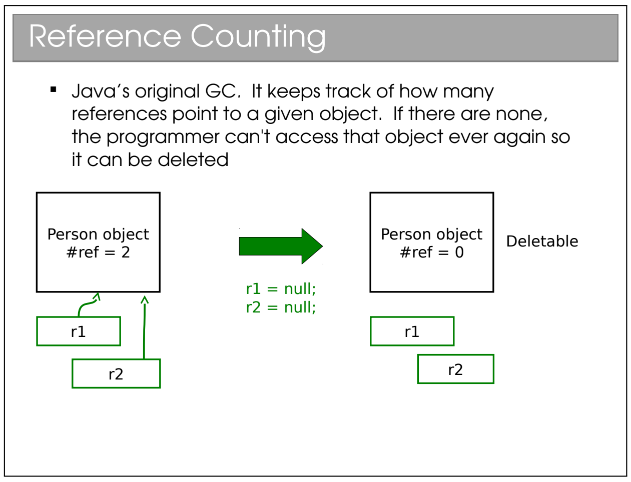
- Conventional destructors don't make sense in non-deterministic systems
  - When will they run?
  - Will they run at all??
- Instead we have finalisers: same concept but they only run when the system deletes the object (which may be never!)

OK, so a finalizer is just a rebadged destructor, but the rebadging is important. It reminds us as programmers that it won't run deterministically. Because you can't tell when finalizer methods will get called in Java, their value is greatly reduced. It's actually quite rare to see them in Java in my experience.



Garbage Collection

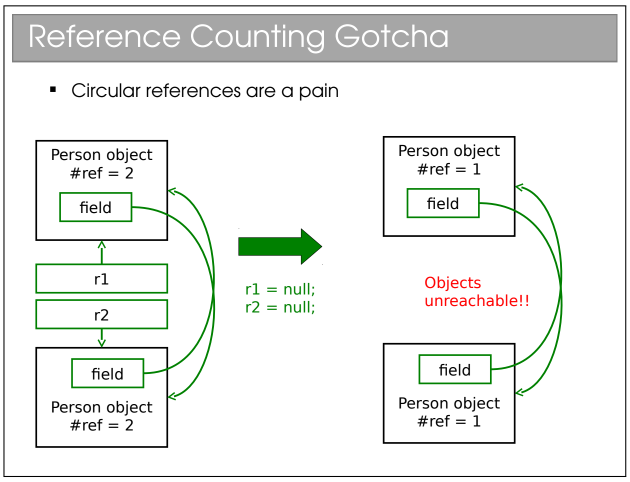
- So how exactly does garbage collection work? How can a system know that something can be deleted?
- The garbage collector is a separate process that is constantly monitoring your program, looking for things to delete
- Running the garbage collector is obviously not free. If your program creates a lot of short-term objects, you will soon notice the collector running
  - Can give noticeable pauses to your program!
  - But minimises memory leaks (it does not prevent them…)
- There are various algorithms: we'll look at two that can be found in Java
  - Reference counting
  - Tracing



Reference Counting

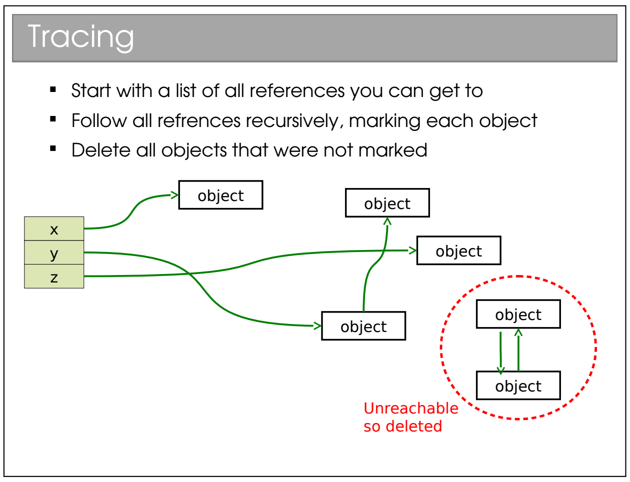
- Java's original GC. It keeps track of how many references point to a given object. If there are none, the programmer can't access that object ever again so it can be deleted

Note that reference counting has an associated cost - every object needs more memory (to store the reference count) and we have to monitor changes to all references to keep the counts up to date.



Reference Counting Gotcha

- Circular references are a pain

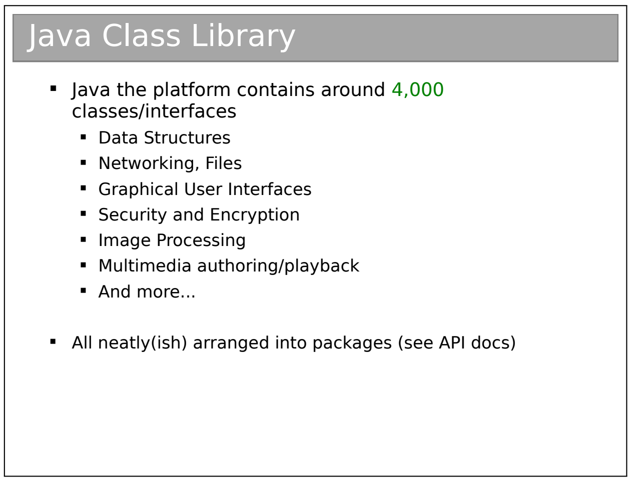


Tracing

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

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)

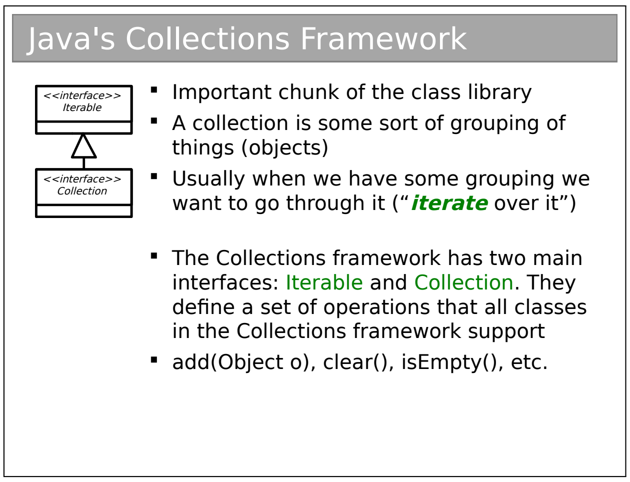
Remember Java is a *platform*, not just a programming language. It ships with a huge *class library*: that is to say that Java itself contains a big set of built-in classes for doing all sorts of useful things like:

- Complex data structures and algorithms
- I/O (input/output: reading and writing files, etc)
- Networking
- Graphical interfaces

Of course, most programming languages have built-in classes, but Java has a big advantage. Because Java code runs on a virtual machine, the underlying platform is abstracted away. For C++, for example, the compiler ships with a fair few data structures, but things like I/O and graphical interfaces are completely different for each platform (Windows, OSX, Linux, whatever). This means you usually end up using lots of third-party libraries to get such extras—not so in Java.

There is, then, good reason to take a look at the Java class library to see how it is structured.

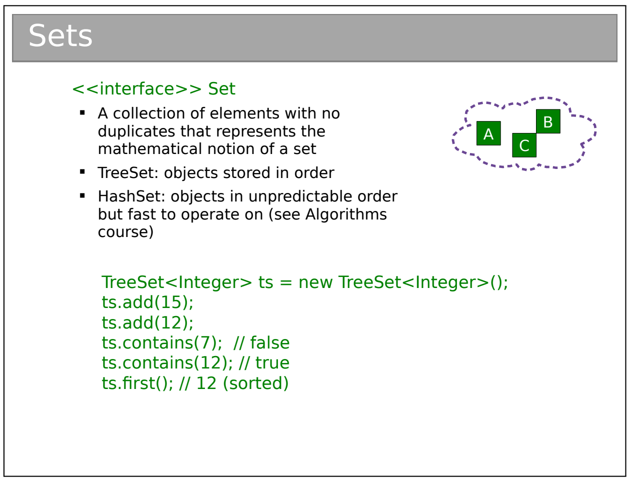
7.1 Collections and Generics


<<interface>> Iterable
<<interface>> Collection

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.

The Java Collections framework is a set of interfaces and classes that handles groupings of objects and allows us to implement various algorithms invisibly to the user (you'll learn about the algorithms themselves next term).

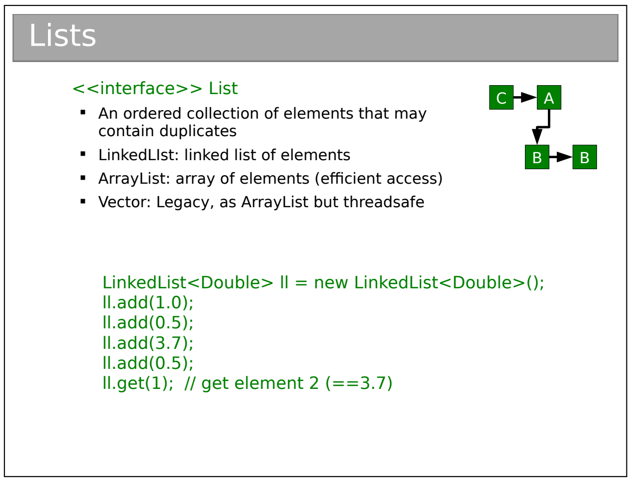


Sets

<<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)
```

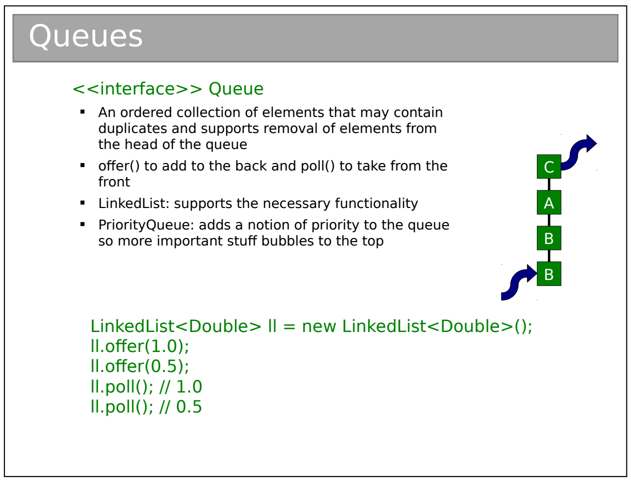


Lists

<<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)
```

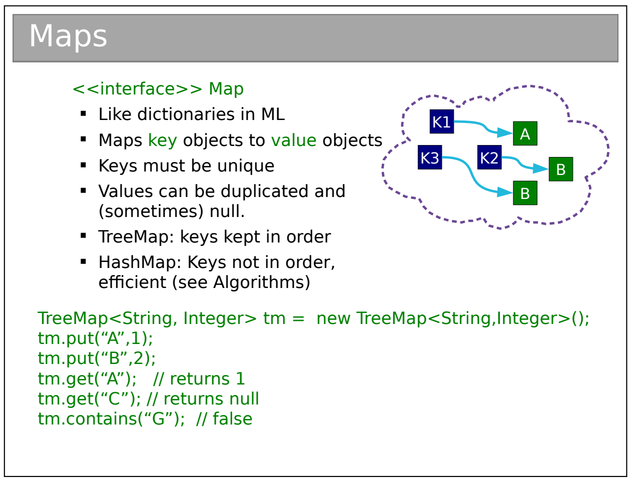


Queues

<<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

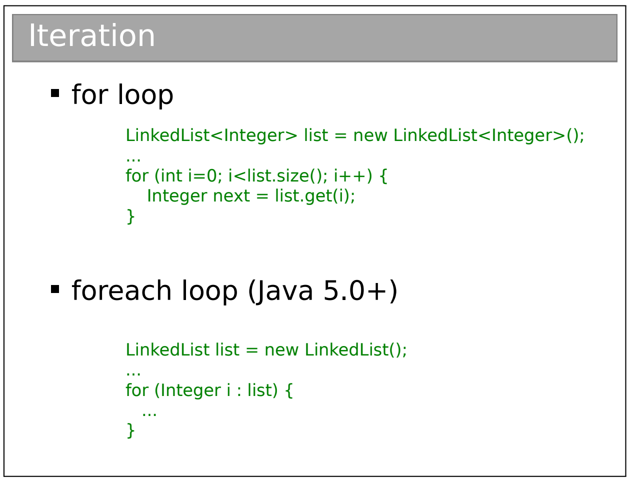
<<interface>> Map

- 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
```

There are other interfaces in the Collections class, and you may want to poke around in the API documentation. In day-to-day programming, however, these are likely to be the interfaces you use.

Now, don't worry about too much what's going on behind the scenes (that comes in the Algorithms course), just recognise that there are a series of implementations in the class library that you can use, and that each has different properties. You should get into the habit of reading the API descriptions to find best choice for your specific problem.



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+)

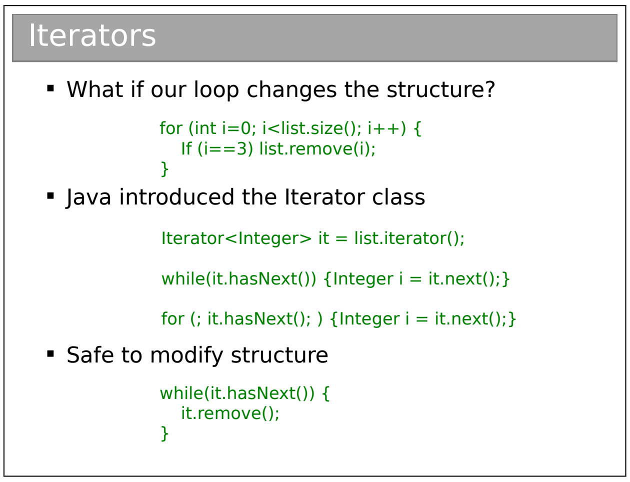
```
LinkedList list = new LinkedList();
…
for (Integer i : list) {
   …
}
```

The foreach notation works for arrays too and it's particularly neat when we have nested iteration. E.g. iteration over all students and their subjects:

```
for (Student stu : studentlist)
  for (Subject sub : subjectlist)
    getMarks(stu, sub);
```

versus:

```
for (int i=0; i<studentlist.size(); i++) {
  Student stu = studentlist.get(i);
  for (int j=0; i<subjectlist.size(); i++) {
    Subject sub = subjectlist.get(j);
    getMarks(stu, sub);
  }
}
```



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

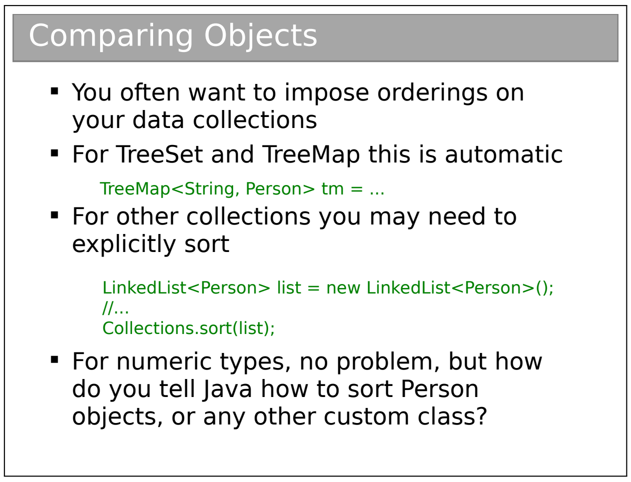
```
Iterator<Integer> it = list.iterator();

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

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

- Safe to modify structure

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

Note that the foreach structure isn't useful with Iterators. So we sacrifice some code readability for the ability to adjust the Collection's structure as we go.

7.2 Comparing Objects



Comparing Objects

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

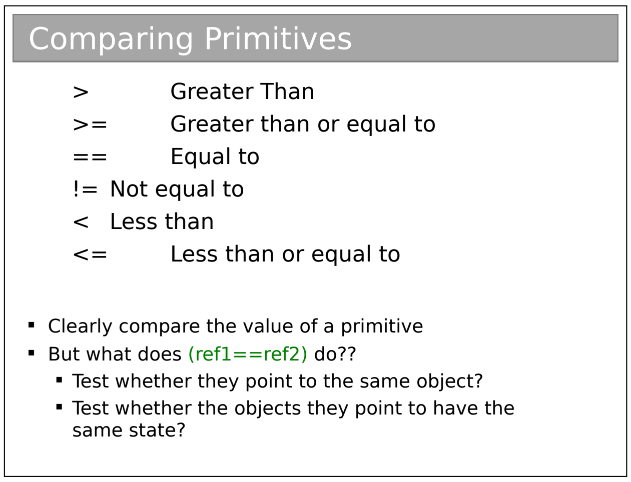
```
TreeMap<String, Person> tm = ...
```

- 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?

Collections are great, but often you end up needing to impose orderings (i.e. sort). Examples include printing users by surname, or computing numerical metrics such as the median.



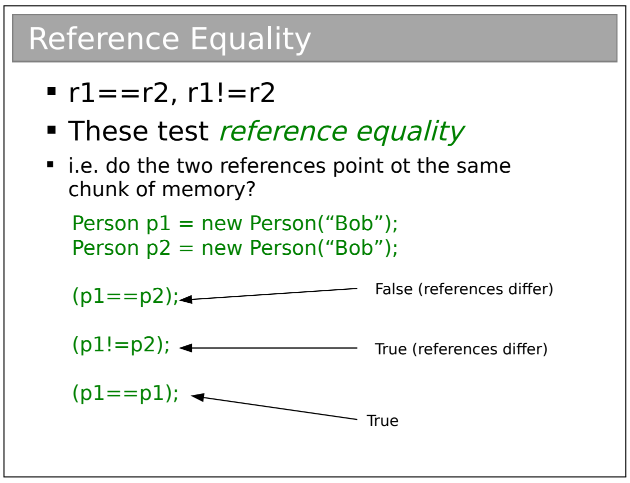
Comparing Primitives

>	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?

The problem is that we deal with references to objects, not objects. So when we compare two things, do we compare the references of the objects they point to? As it turns out, both can be useful so we want to support both.

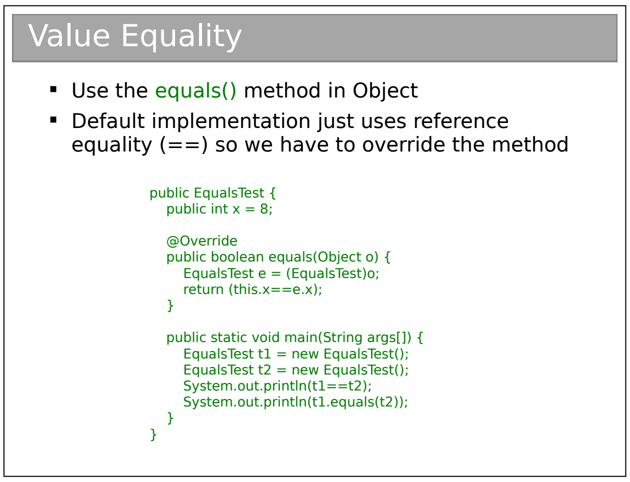
7.2.1 Object Equality



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
```

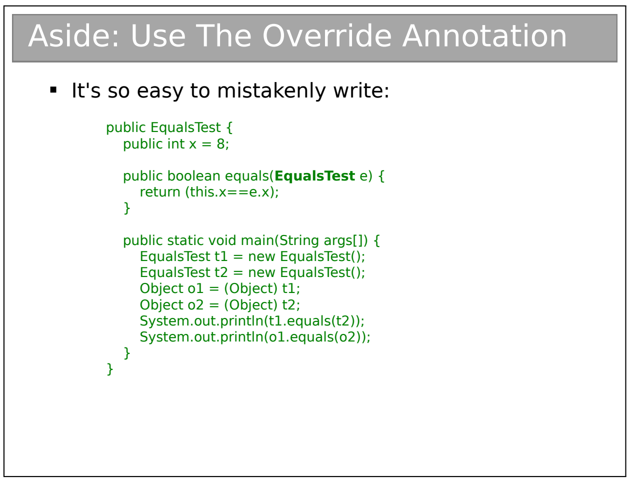


Value Equality

- 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));
   }
}
```

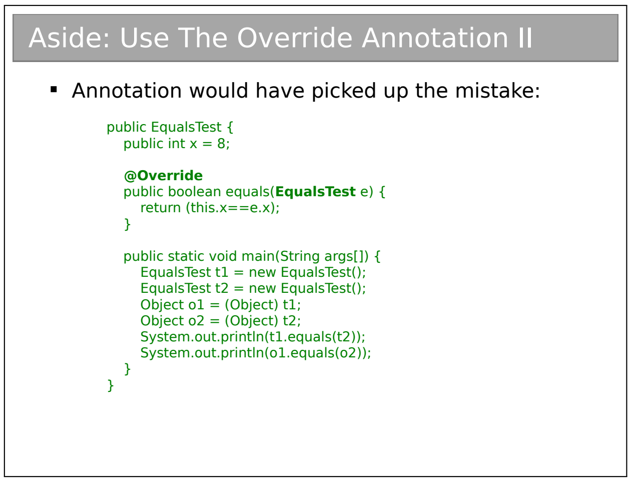
I find this mildly irritating: every class you use will support equals() but you'll have to check whether or not it has been overridden to do something other than ==. Personally, I try to limit my use of equals() to objects from core Java classes, where I trust it to have been done properly.



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));
   }
}
```



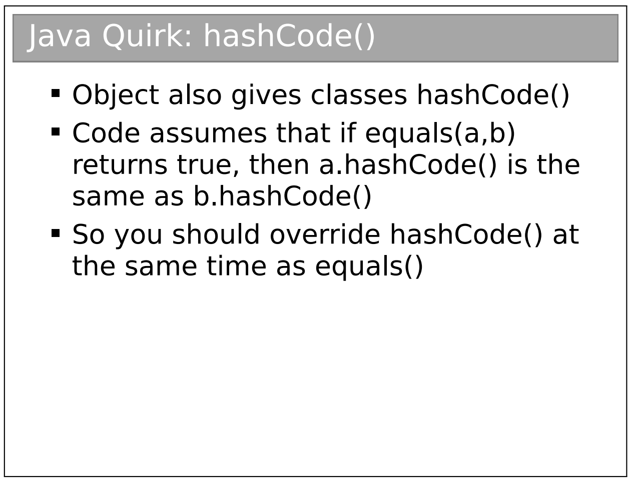
Aside: Use The Override Annotation II

- 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));
   }
}
```

What's happening here is that the signature of our overriding method doesn't match the one in Object. So, Java actually *overloads* it, keeping both methods. By using @Override when we mean to override not overload, the compiler will spot our error.

For the geeks out there (i.e. non-examinable), we could write a compiler that spots that EqualsTest is a subclass of Object and therefore do overriding. This is called *covariant parameter types* and is *not* supported by Java.



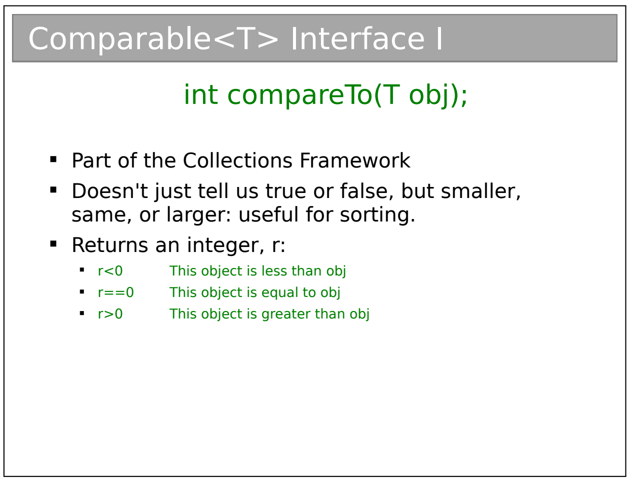
Java Quirk: hashCode()

- 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()

I don't want to go into this in too much detail since you haven't yet met hashes (it's in the Algorithms course next term). For now, just accept that a hash is a function that takes in chunks of information (e.g. all the fields in an object) and spits out a number. Java uses this in its HashMap implementation and other places as a shortcut to having to sequentially compare each field. I mention it here really for completeness so that if any of you override equals() in production code then you know you should also override hashCode(). Details of doing so are easily found on the web and in books (because it's a very common mistake to make!).

7.3 Less Than and Greater Than

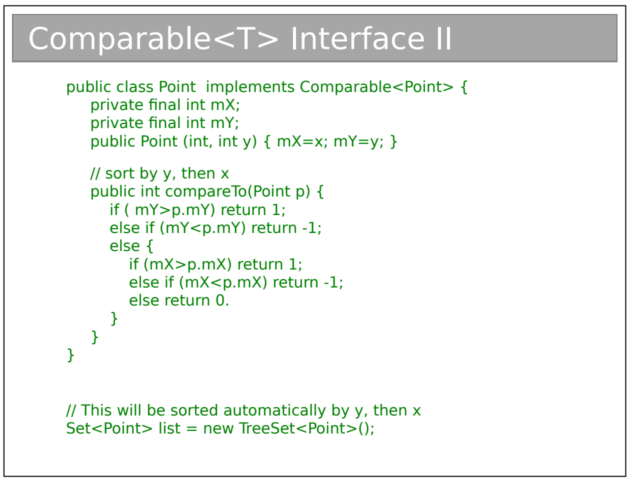
In order to sort your classes using the built in classes, you need to write something that allows two objects to be ordered. Often our classes have a *natural ordering* e.g. people are usually sorted first by surname and then by forename. We can build-in natural ordering to our classes using the Comparable interface:



Comparable<T> Interface I

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

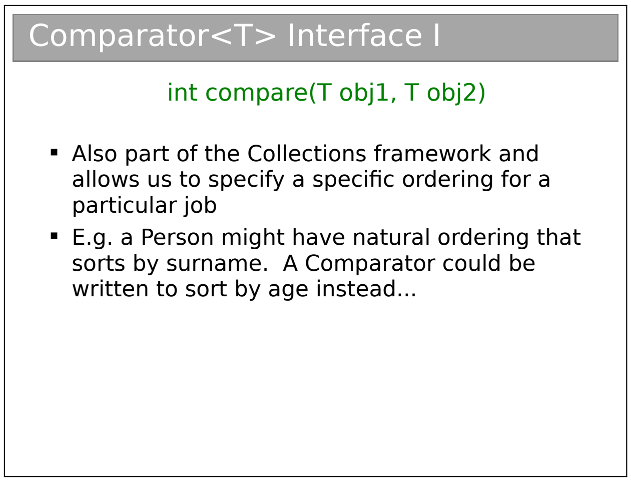
- Part of the Collections Framework
- Doesn't just tell us true or false, but smaller, same, or larger: useful for sorting.
- Returns an integer, r:
  - r<0 This object is less than obj
  - r==0 This object is equal to obj
  - r>0 This object is greater than obj



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>();
```

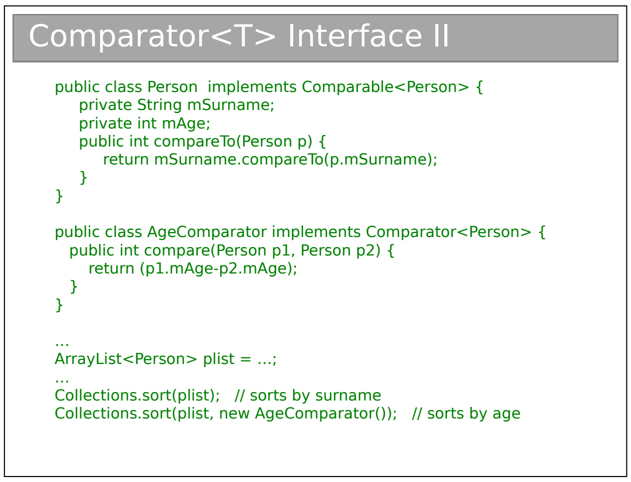
This is all very well, but sometimes we might want to sort with a different ordering (e.g. sort just by forename). Java Collections lets us do this by supplying a custom piece of code for the ordering: a *Comparator*:



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...

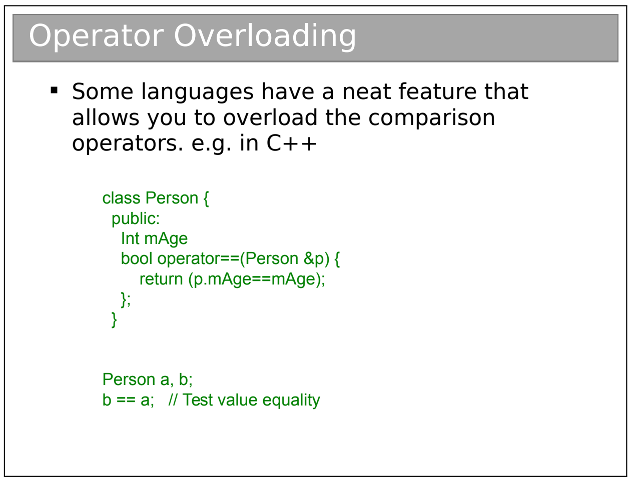


Comparator<T> Interface II

```
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
```

Note that a natural ordering uses compareTo() whilst a comparator uses compare().

7.4 Operator Overloading



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
```

Java doesn't have this, but it's good to know what it is at this stage.

Lecture 8

Error Handling Revisited

As you have almost certainly discovered, errors crop up all over the place when developing software. We see various types:

Syntactic errors (missing brackets or whatever) are usually quite easy to spot because you get a nice explanatory compiler warning (ahem... unless you're using poly/ML...).

Logical errors (i.e. bugs) are more problematic, not least because comprehensive testing (checking the output for every possible input *and* system state) is usually infeasible for anything but toy programs.

External errors occur for processes our code relies on but we don't control. Examples might be a failing hard disk or an overheating CPU causing them to do things that shouldn't be possible.

So what do you do? Firstly you do what you can to minimise the chance of bugs. Secondly, you accept that there will still be problems (if nothing else the external errors will persist) and you use techniques to handle them. You've already seem the latter with ML's exceptions: we'll look at Java's exceptions here too, but first let's consider ways to reduce the bugs in the code you deliver.

8.1 Minimising Bugs

8.1.1 Modular (Unit) Testing

OOP (strongly) pushes you to develop uncoupled chunks of code in the form of classes. Each class should be testable (mostly) independently of the others. It is much easier to comprehensively test lots of small bits of code and then stitch them together than the stitched result!

8.1.2 Using Assertions

When you are debugging an algorithm, it can be useful to use *assertions* at various stages to mark invariants (things that should be true if your algorithm is working). You'll see these next term in the Algorithms course.

8.1.3 Defensive Programming Styles

You can also learn useful habits for each language that can reduce errors. In C for example, `if (exp)` is true whenever `exp` is greater than 0. The problem with this is that you can accidentally do an assignment without realising it:

```
if (x=5) {...}
else {....}
```

Here the programmer presumably wanted to test whether `x` is 5. What actually happens is `x` is *set* to 5 and the expression itself is evaluated as 5, or always true. All because they used = and not == by accident.

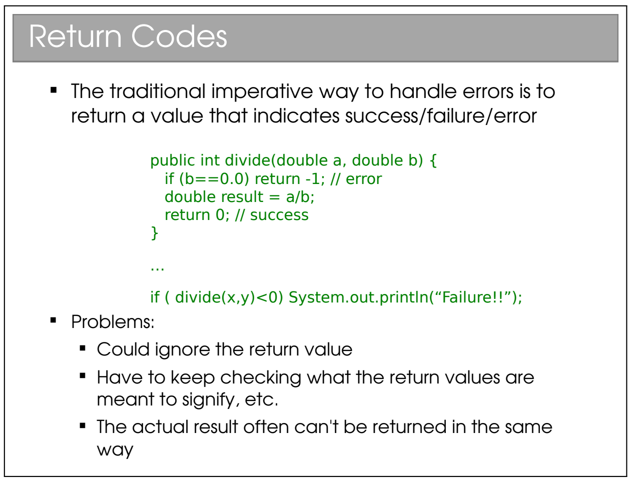
But you can remove this (very common) error altogether by always writing `(5==x)` and not `(x==5)`. Then the error will be caught by the compiler because `(5=x)` is not valid syntax!

8.1.4 Pair Programming etc.

Another quite effective way to spot bugs is via pair programming. Here you program in pairs insofar as one person writes code, while the other watches over their shoulder, looking for errors or bugs. The writer and the watcher switch roles regularly. Various other such agile programming techniques exist.

8.2 Dealing with Errors

8.2.1 Return Codes



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

Many older languages (C included) have no explicit mechanism for error handling, Instead the common approach is to return the error status via the normal return type: a *return code*. If your function isn't proper and its 'result' is a side effect (i.e. it has a void return type in Java) then we can just return the error code:

```
int setValue(LinkedList<int> list,
             int element, int value) {
  if (list.size()>element) {
    list.set(element,value);
    return 0;  // no error to signal
  }
  else return -1; // this element doesn't exist

}
```

Here the function can only return 0 or -1, the latter being a signal that there was an error (the element didn't exist).

If you have a function that naturally returns a result, you can pick some result values that are used (only) to signal errors:

```
float sqrt(float a) {
   if (a<0.0) return -1.0;
   else {
      ...
   }
}
```

Here the sqrt function only returns positive roots so we can repurpose all negative floats to signal errors.

```
float sqrt(float a) {
   if (a<0.0) return -1.0;
   else {
      ...
   }
}
```

If the return type isn't something we can repurpose (e.g. a custom class) then we can instead pass the output by reference and have the function return an integer to indicate the error state. E.g,

```
SomeCustomClass sqrt(float a) {
  return new SomeCustomClass(...);
}
```

becomes

```
int func(float a, SomeCustomClass result ) {
    if (a<0.0) return -1.0;
    else result.set(...);
    return 0;
}
```

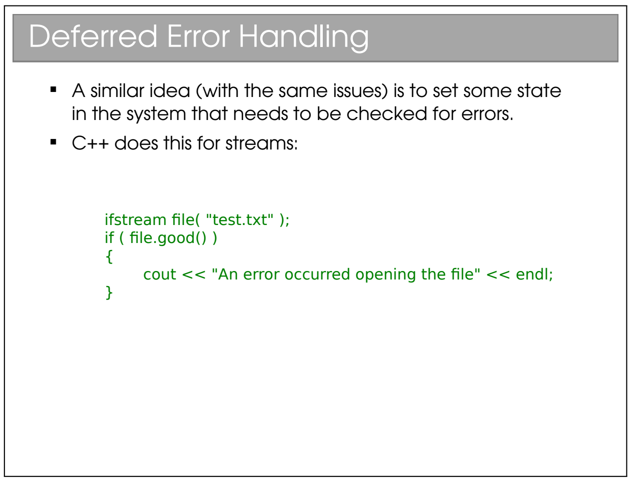
You might see functions that return null if they have an error. This is a very bad practice since it relies on the programmer using the function to check for null. If they don't, they'll likely try to dereference null and their program will die...

In fact, this is a larger problem with the general approach. We are dependent on the programmer testing the return value. Two problems arise: firstly, they could neglect to check (really common); secondly, they end up with really nasty looking code such as:

```
int retval = somefunc();

if (retval==-1) {
   // handle error type 1
}
else if (retval==-2) {
   // handle error type 2
}
else if (retval==-3) {
   // handle error type 3
}
```

Here, just writing one line to call one function results in a screen-worth of error handling code. This constant mixing of code and error handling makes the code all but unreadable.

8.2.2 Deferred Error Handling

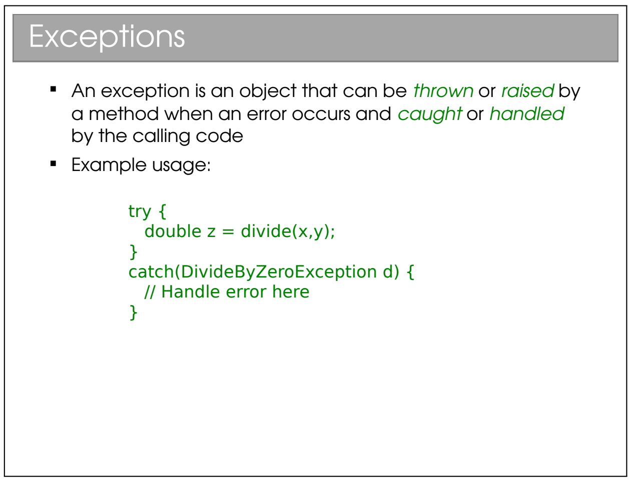


Deferred Error Handling

- 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;
}
```

8.2.3 Exceptions

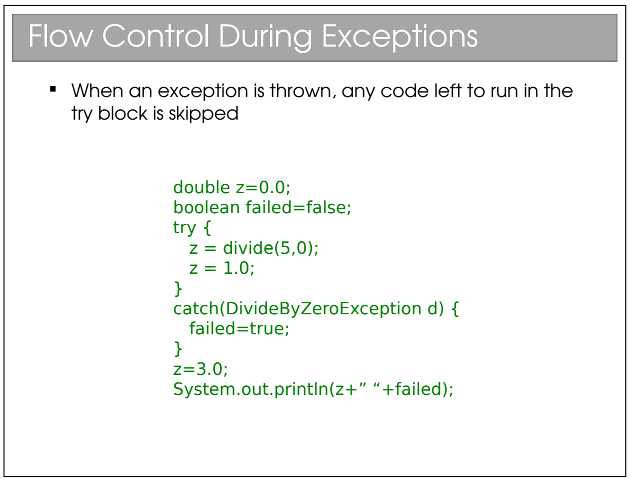


Exceptions

- 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
}
```

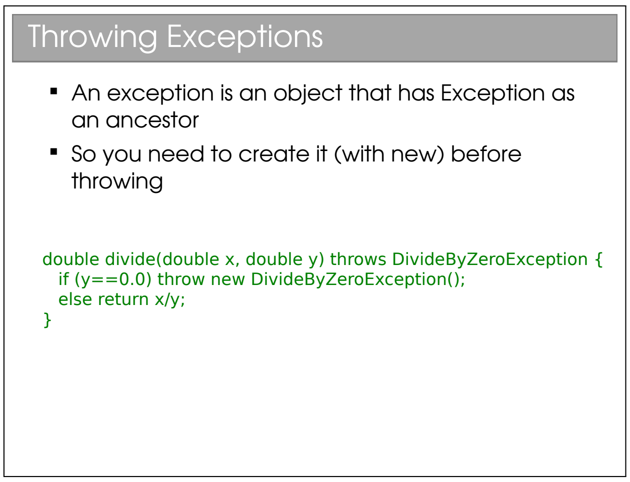
Of course, you already met Java's exceptions in the pre-arrival course, as well as ML's in FoCS. We'll cover the Java concepts in a little more depth here, whilst recapping the content you've done. Note there is a tendency to use the terminology throw/catch rather than raise/handle in OOP languages—I don't know why. First some recap:



Flow Control During Exceptions

- When an exception is thrown, any code left to run in the try block is skipped

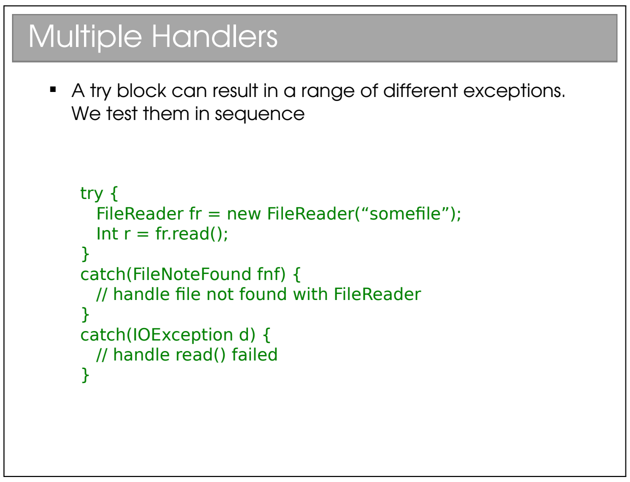
```
double z=0.0;
boolean failed=false;
try {
   z = divide(5,0);
   z = 1.0;
}
catch(DivideByZeroException d) {
   failed=true;
}
z=3.0;
System.out.println(z+” “+failed);
```



Throwing Exceptions

- An exception is an object that has Exception as an ancestor
- So you need to create it (with new) before throwing

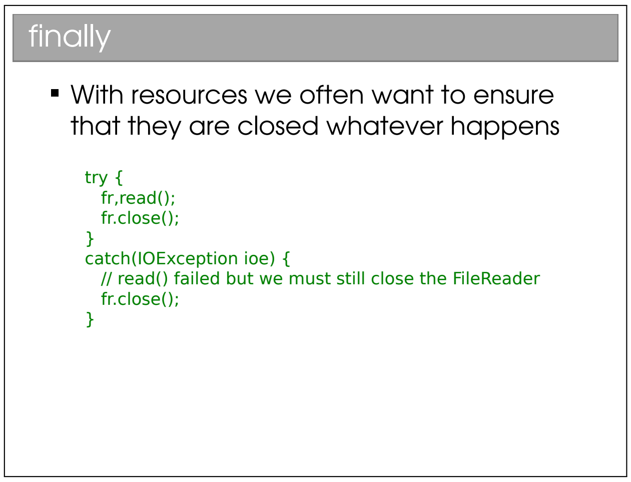
```
double divide(double x, double y) throws DivideByZeroException {
   if (y==0.0) throw new DivideByZeroException();
   else return x/y;
}
```



Multiple Handlers

- A try block can result in a range of different exceptions. We test them in sequence

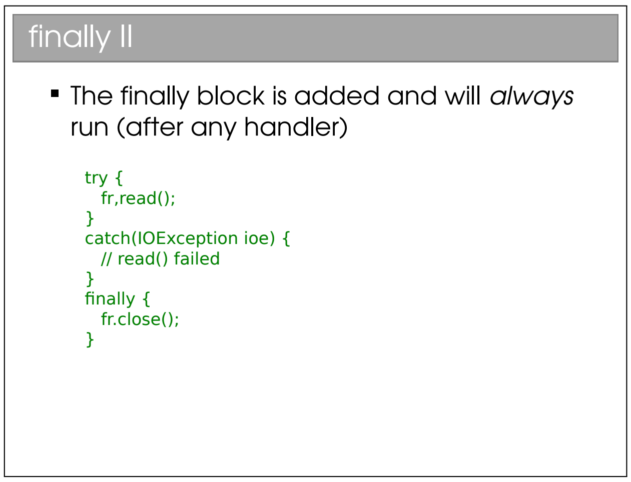
```
try {
   FileReader fr = new FileReader(“somefile”);
   Int r = fr.read();
}
catch(FileNoteFound fnf) {
   // handle file not found with FileReader
}
catch(IOException d) {
   // handle read() failed
}
```



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();
}
```

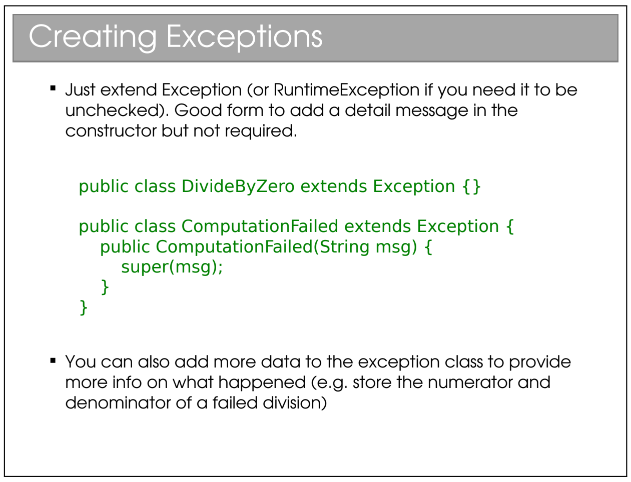


finally II

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

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

Note that once any catch block is matched, the remaining catch blocks are skipped. Whilst you already know about the flow control, you hadn't considered creating your own exceptions:

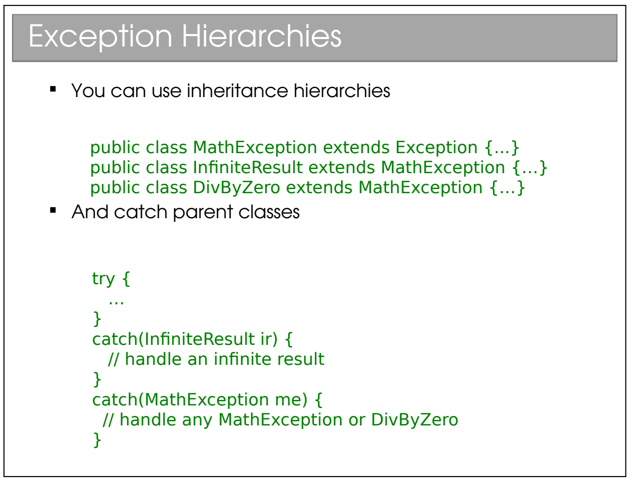


Creating Exceptions

- 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)



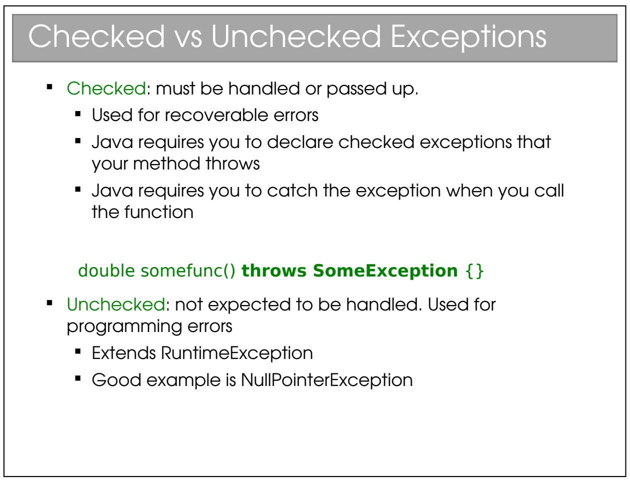
Exception Hierarchies

- 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

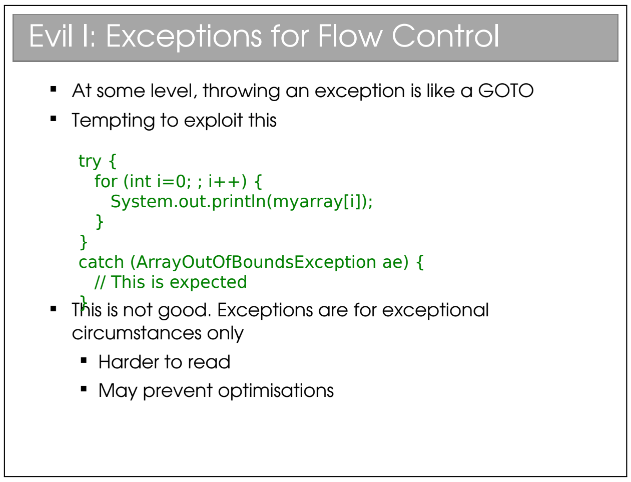
- Checked: must be handled or passed up.
  - Used for recoverable errors
  - Java requires you to declare checked exceptions that your method throws
  - Java requires you to catch the exception when you call the function

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

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

There is an ongoing debate about the value of checked exceptions and they feature in some OOP languages but not others. Most of the time you'll be writing and dealing with checked exceptions in Java. You'll encounter unchecked exceptions only when you mess up in your code.

Aside: It turns out with Java they decided that `RuntimeException` should inherit from `Exception`. This means that if you ever write `catch(Exception e) {...}` then you will also catch the unchecked exceptions. So don't ever write that unless you know what you are doing!



Evil I: Exceptions for Flow Control

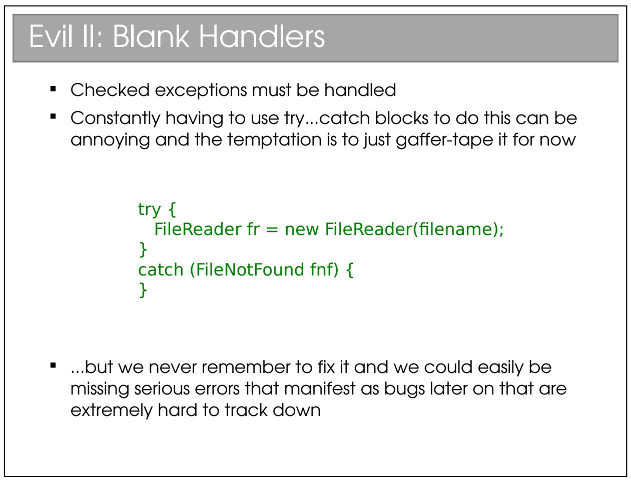
- At some level, throwing an exception is like a GOTO
- Tempting to exploit this

```
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

The code readability argument should be obvious but the second argument warrants more discussion. If you Google the notion of flow control with exceptions, you will probably find many comments that suggest exception throwing is very slow compared to 'normal' code execution. This is attributed variously to the need to create an Exception object; the need to create a stack trace; or even just the need to create a message string. Some people report Exception handling was 50 times slower on the first JVMs!

Now, you *could* write a JVM that handled exception throwing efficiently, such that code like that in the slide would carry little performance penalty. But the crucial point is that there is no guarantee that a JVM will do so (and many still don't). Exceptions are intended to be rare occurrences and it is perfectly reasonable (if not natural) for a JVM creator to assume this and therefore not need to worry about optimising exception handling. Bottom line: this smells bad.



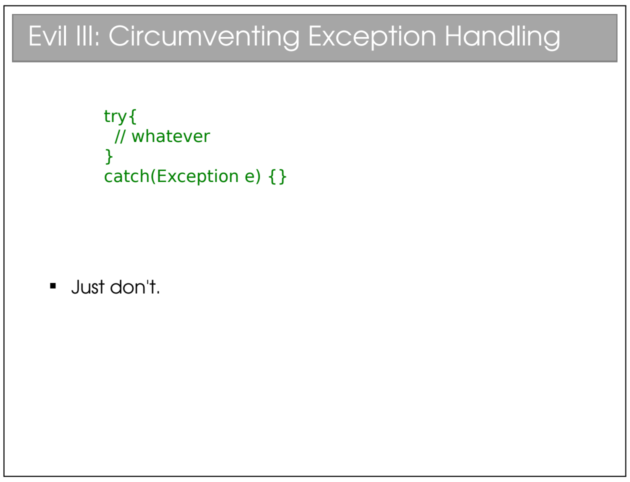
Evil II: Blank Handlers

- 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

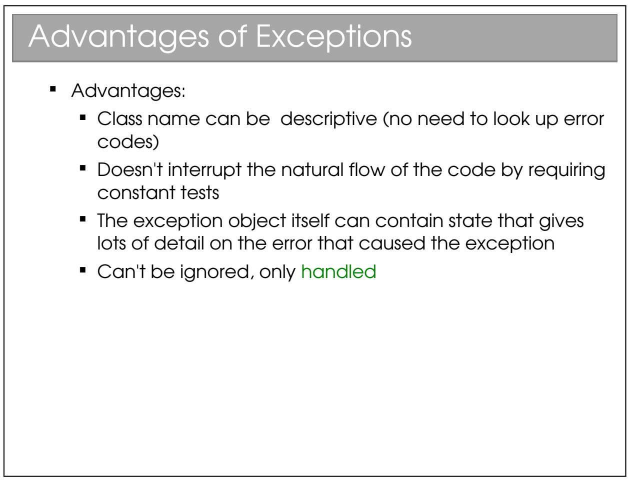
This is a bad habit that novices tend to adopt—try not to develop it yourself. Eclipse at least discourages blank handlers, automatically filling in `e.printStackTrace()` so there's some record of the problem printed to the screen. However, in large programs, where there's often lots of debug output flowing to the console, these messages are easily missed... Better to fill in your handlers!



Evil III: Circumventing Exception Handling

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

- Just don't.

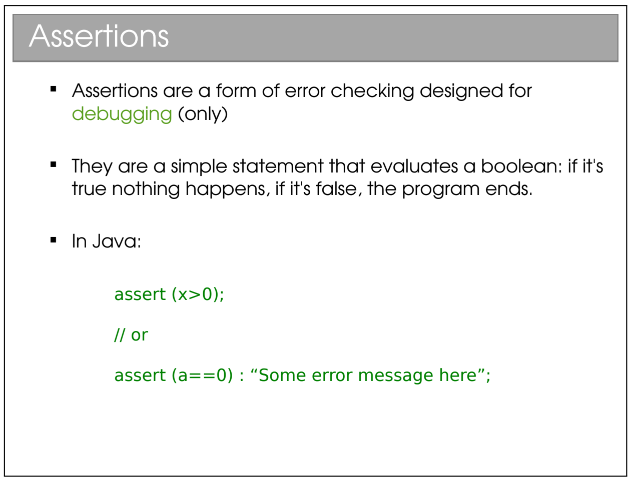


Advantages of Exceptions

- 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

`http://java.sun.com/docs/books/tutorial/essential/exceptions/`

8.2.4 Assertions

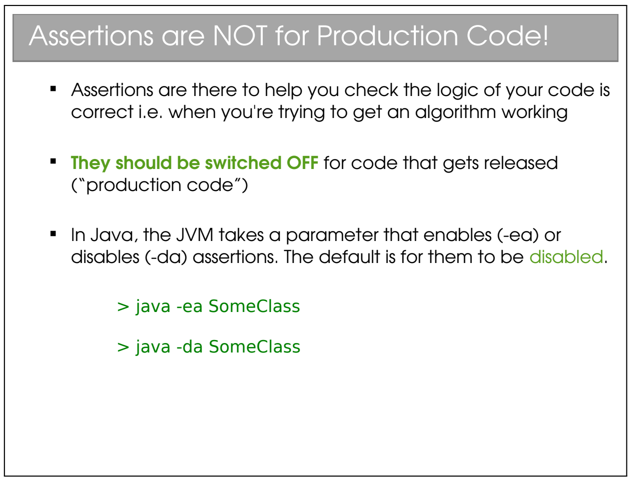


Assertions

- 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 a simple addition to many languages that can really help development, but they complement exceptions (or other error handling techniques) rather than replace them.

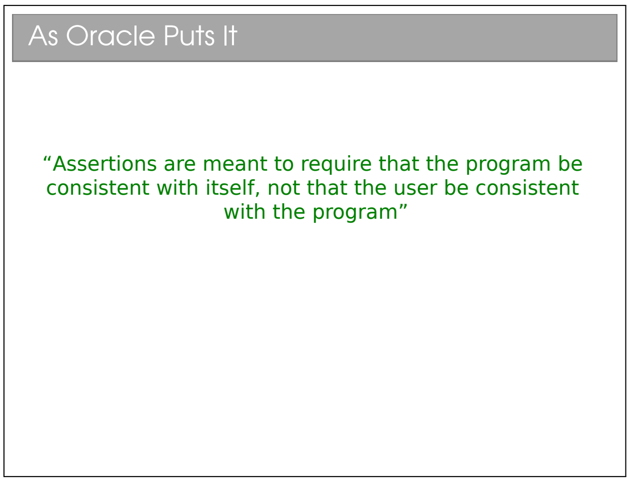


Assertions are NOT for Production Code!

- 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.

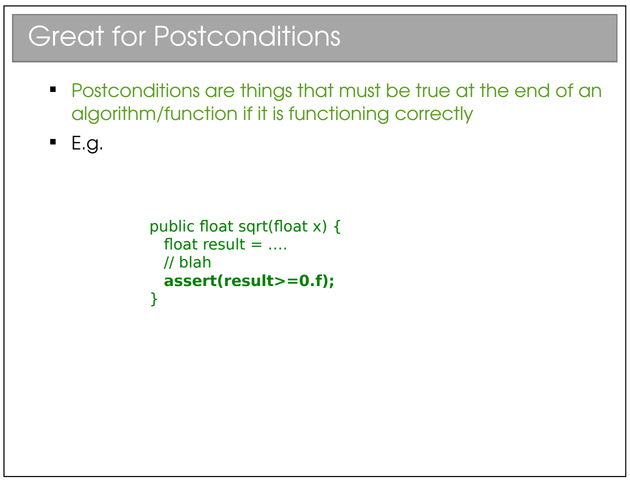
```
> java -ea SomeClass

> java -da SomeClass
```

This is important: assertions will kill your program if they detect an error. There's no opportunity to handle the error so they're just for development, not production.



As Oracle Puts It

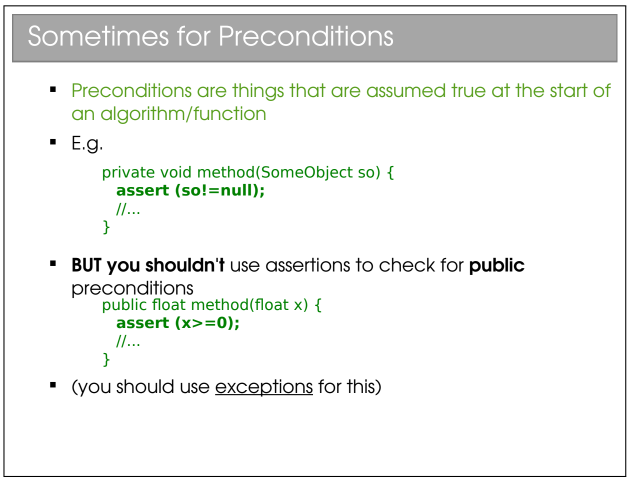
"Assertions are meant to require that the program be consistent with itself, not that the user be consistent with the program"



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.

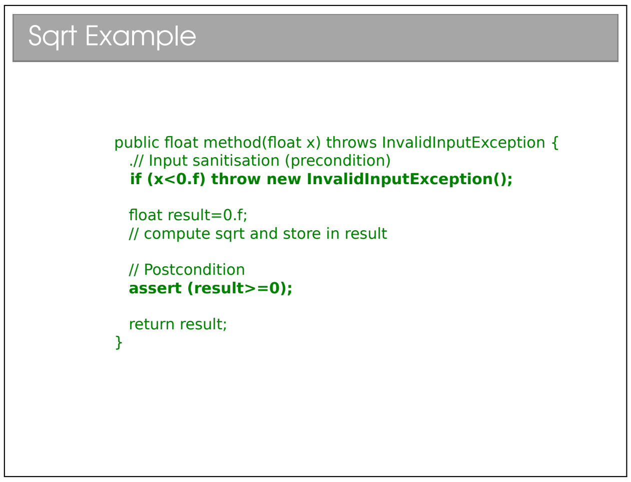
```
private void method(SomeObject so) {
   assert (so!=null);
   //...
}
```

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

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

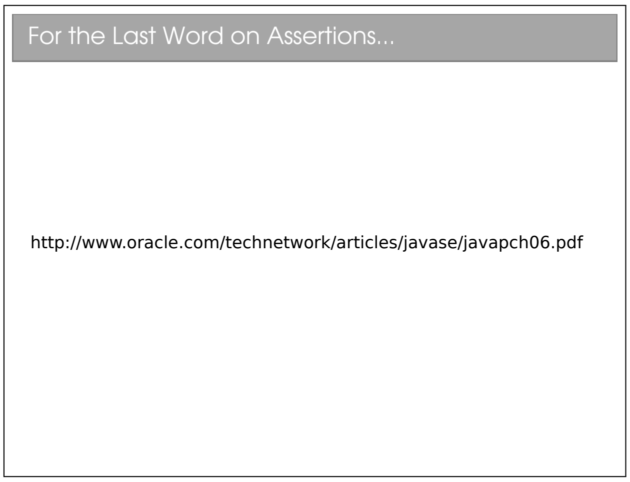
- (you should use exceptions for this)

If a use of your method provides bad (nonsensical) inputs, you should offer them the chance to remedy the mistake by throwing an exception. Assertions would just kill the program (if enabled for release , which they shouldn't be), or not catch the error because they are disabled!



Sqrt Example

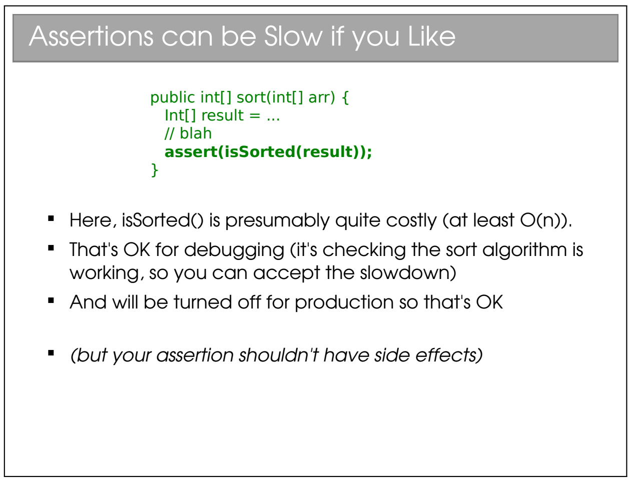
```
public float method(float x) throws InvalidInputException {
  .// Input sanitisation (precondition)
  if (x<0.f) throw new InvalidInputException();

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

  // Postcondition
  assert (result>=0);

  return result;
}
```



For the Last Word on Assertions...

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

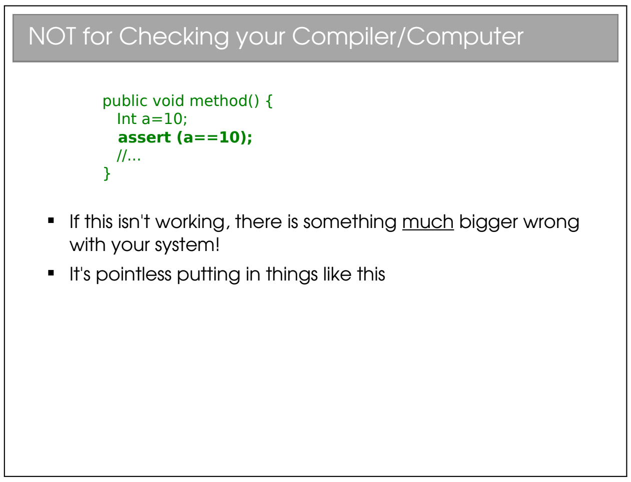
The distinction is subtle but important. The 'assert' is only used to test the correctness of the algorithm output when given a valid (positive) input. If the assertion fires, it's programmer error and not user error.



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);
  //...
}
```

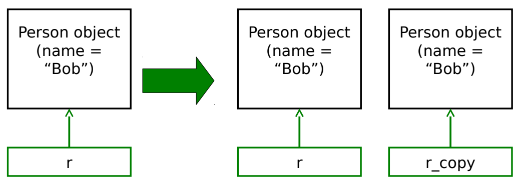
- If this isn't working, there is something <u>much</u> bigger wrong with your system!
- It's pointless putting in things like this

Lecture 9

Copying Objects

Cloning I

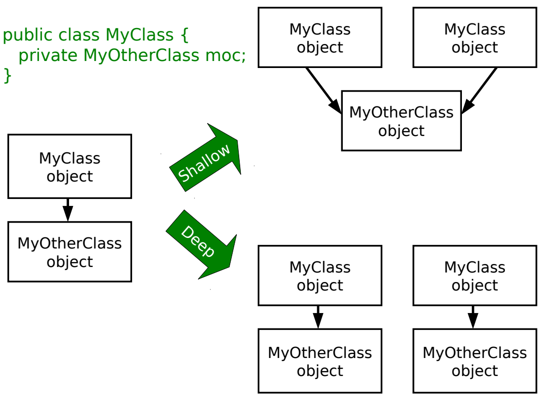
- Sometimes we really do want to copy an object


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

Shallow and Deep Copies

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



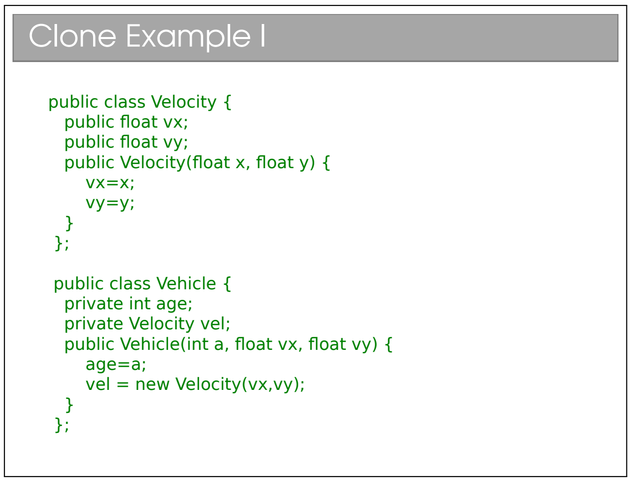
Cloning II

- 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)

Java Cloning

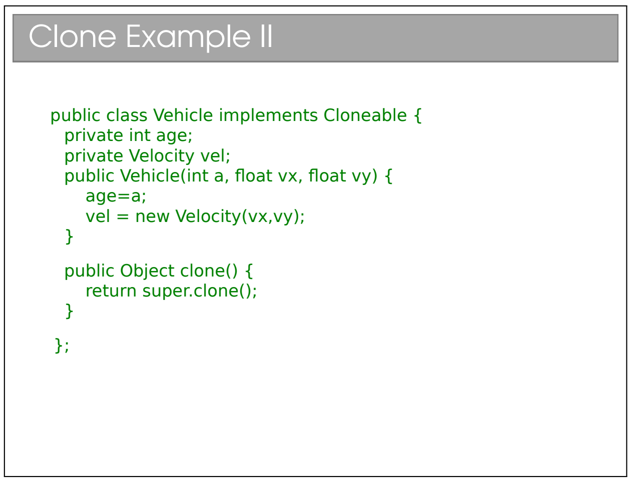
- So do you want shallow or deep?
  - The default implementation of clone() performs a **shallow** copy
  - But Java developers were worried that this might not be appropriate: they decided they wanted to know for sure that we'd thought about whether this was appropriate
- Java has a **Cloneable** interface
  - If you call clone on anything that doesn't extend this interface, it fails

Java is unusual in that it really, really wants you to use OOP. In your practicals you will have noticed that, even to do simple procedural stuff, you had to encase everything in a class—even the main() method. A further decision they made is that ultimately *all* classes will inherit from a special Object class. i.e. the top of all inheritance trees is Object even though we never explicitly say so in code...



Clone Example I

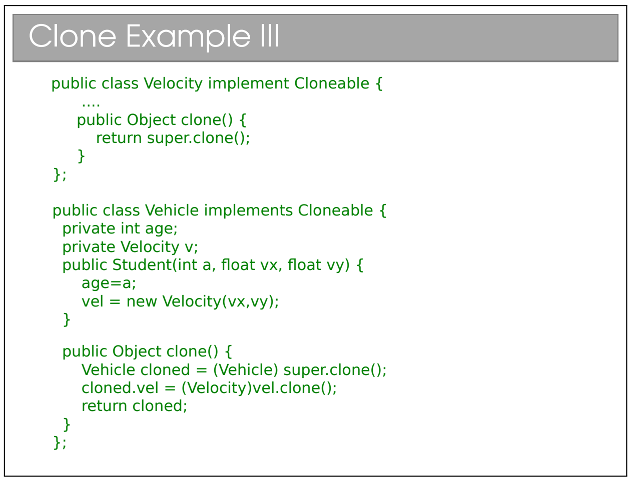
```
public class Velocity {
  public float vx;
  public float vy;
  public Velocity(float x, float y) {
     vx=x;
     vy=y;
  }
};

 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();
  }

 };
```

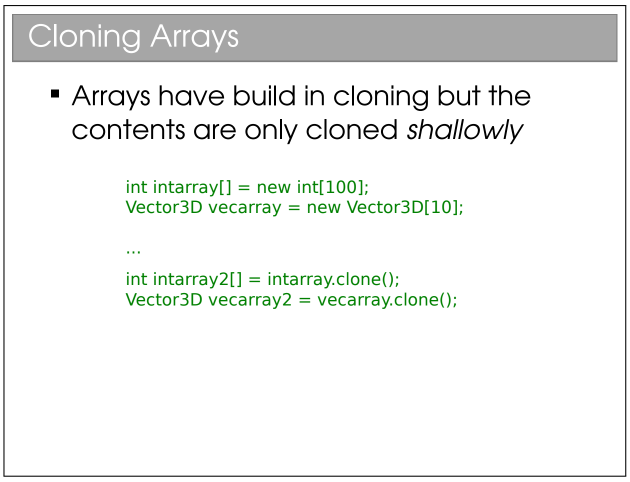
Here we fill in the clone() method using super.clone(). You can think of this as doing a byte-for-byte copy of an object in memory. Any primitive types (such as age) will therefore be copied. And references will also be copied, but not the objects they point to. Hence this much gets us a shallow copy.



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;
  }
};
```

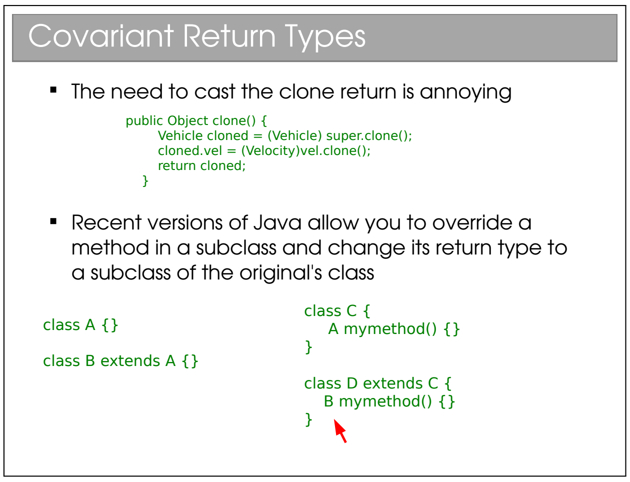
A deep clone requires that we clone the objects that are referenced (and they, in turn clone any objects they reference, and so on). Here we make Velocity cloneable and make sure to clone the member variable that Vehicle has.



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 C {
                                       A mymethod() {}
class B extends A {}                }

                                    class D extends C {
                                       B mymethod() {}
                                    }
```

This is a similar concept to the covariant parameter tyoes we met breifly in lecture 7. We saw Java does *not* support that, but it does support this. So if we have:

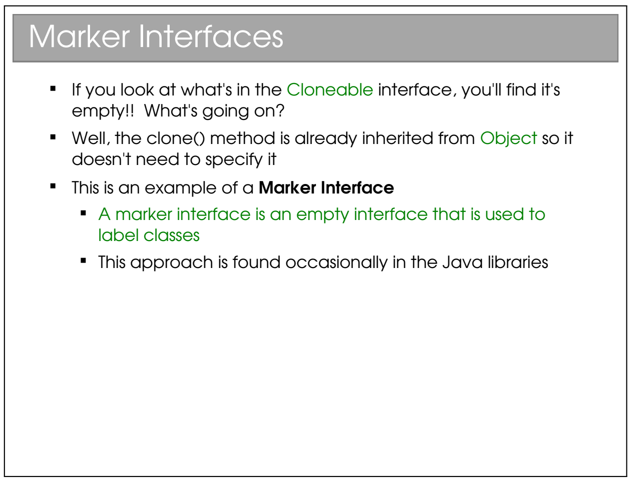
```
public class A {
   Object void work(Object o) {...}
}
```

then the following is not allowed (covariant parameter types):

```
public class B extends A {
   @Override
   public Object work(Person p) {...}
}
```

but this is (covariant return types):

```
public class C extends A {
   @Override
   public Person work(Object o) {...}
}
```

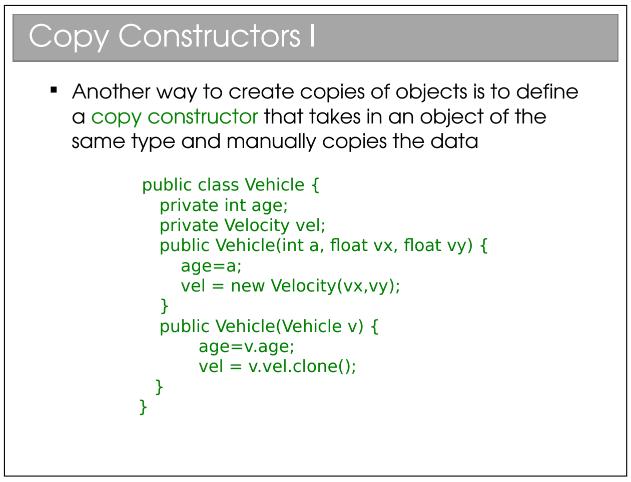


Marker Interfaces

- 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

You might also see these marker interfaces referred to as *tag interfaces*. They are simply a way to label or tag a class. They can be very useful, but equally they can be a pain (you can't dynamically tag a class, nor can you prevent a tag being inherited by all subclasses).

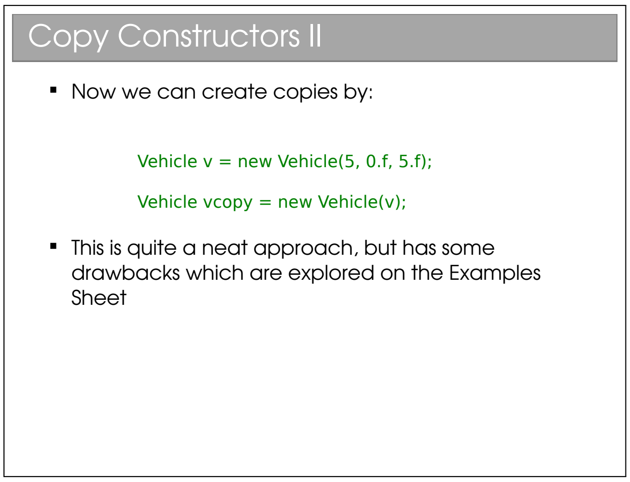
The `clone()` approach is unique to Java. It can be a bit of a headache, but it was meant to address the shortcomings of the de-facto copying approach in OOP, which is the use of copy constructors:



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:

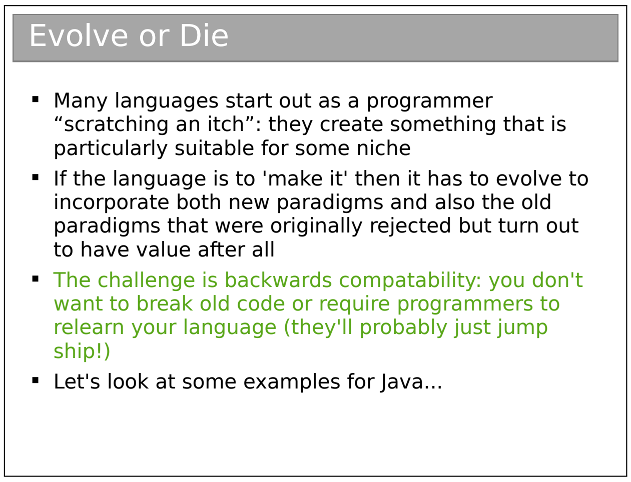
```
Vehicle v = new Vehicle(5, 0.f, 5.f);

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

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

I won't go into detail on these here. Instead they are on the examples sheet.

Lecture 10

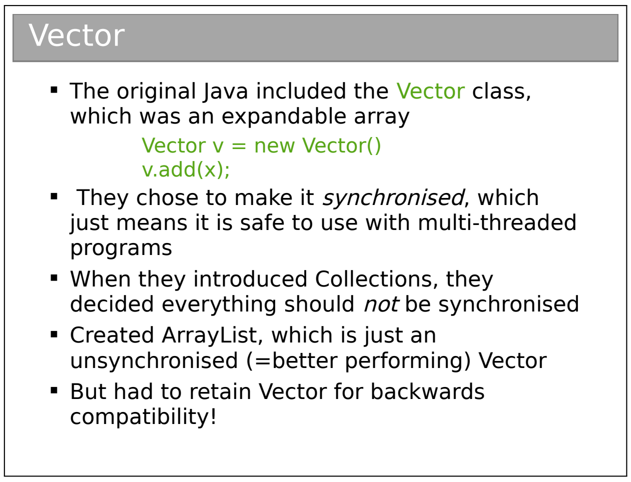
Language Evolution



Evolve or Die

- Many 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...

Ostensibly this course is about OOP, but in reality few languages can claim to be a pure implementation of any particular paradigm. Even ML offers you imperative programming. Actual languages are a mish-mash of concepts, some of which are inevitably retrofitted. This retrofitting tends to produce ugly syntax and unexpected quirks, so it’s good to explore some examples (in Java of course).



Vector

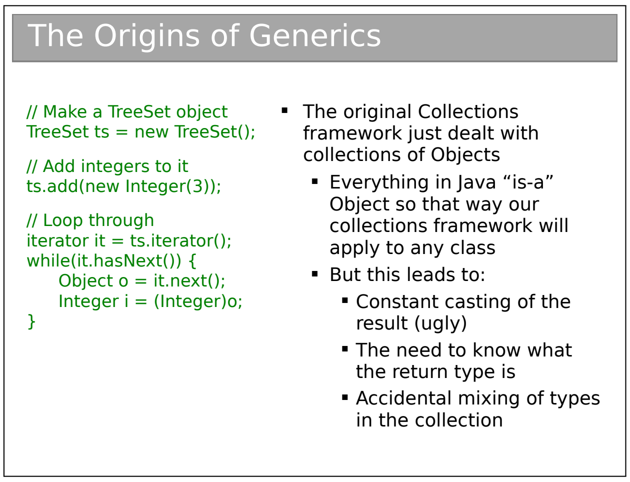
- The original Java included the Vector class, which was an expandable array

```
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
- But had to retain Vector for backwards compatibility!

Vector has no place in modern Java really, and if you are using it you should stop doing so, in favour of using ArrayList. If you need it to be synchronised, this can be done (see next year for those sticking around in the CST). The only reason Vector remains is backwards compatibility. It’s handy to know about it though, since it features in a lot of legacy code.

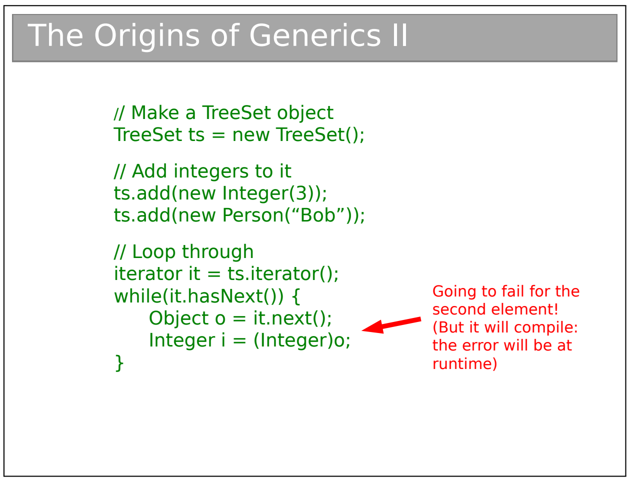
10.1 Generics



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

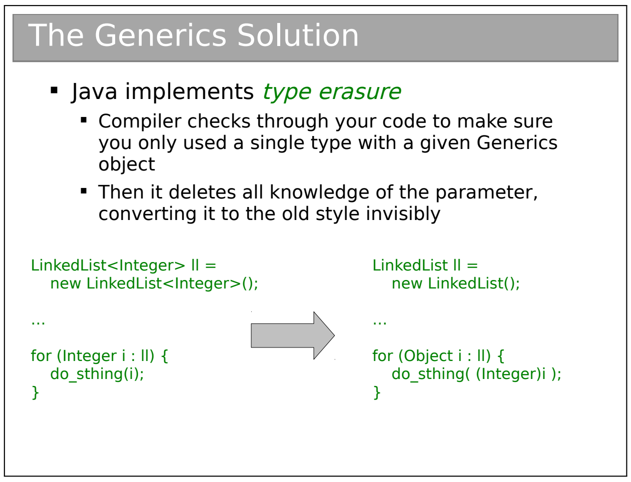


The Origins of Generics II

```
// 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)

This is pretty nasty. The OOP paradigm has let us write a flexible data structure that can handle us wrapping around various types, but it can't apply the restriction that all the types in one object should be the same. Additionally, all this casting makes for ugly code. This is what convinced the Java designers that parameterised types (Generics) were needed. But it was already a bit late: there was tons of established code using Collections (and still is). The Java designers were faced with the problem of updating the language to support parameterised types without breaking everything that went before.



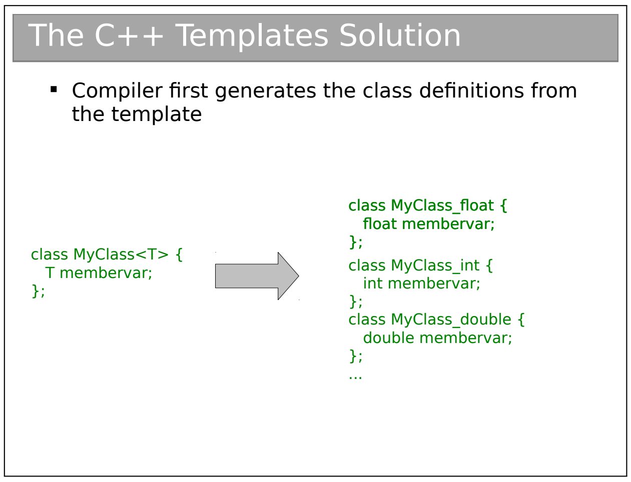
The Generics Solution

- 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 style 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 );
}
```

So now we see why we can't use primitives as parameters: whatever we put there must be castable to Object, which primitives simply aren't.



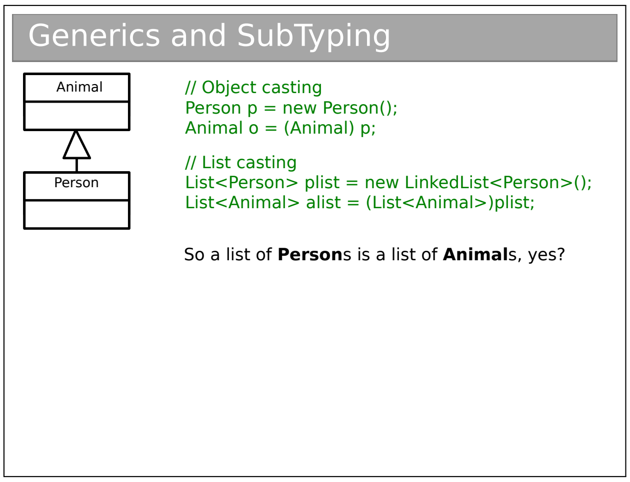
The C++ Templates Solution

- 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;
};
...
```

C++ doesn't suffer from the same problem since it just generates a special class for each instance you request.



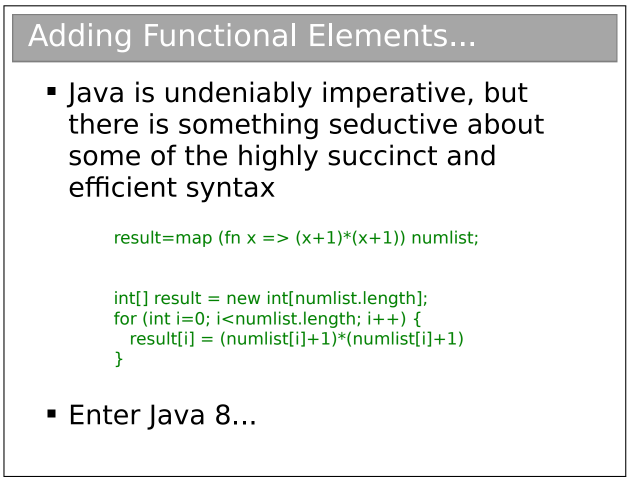
Generics and SubTyping

Animal
Person

```
// 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?

10.2 Java 8



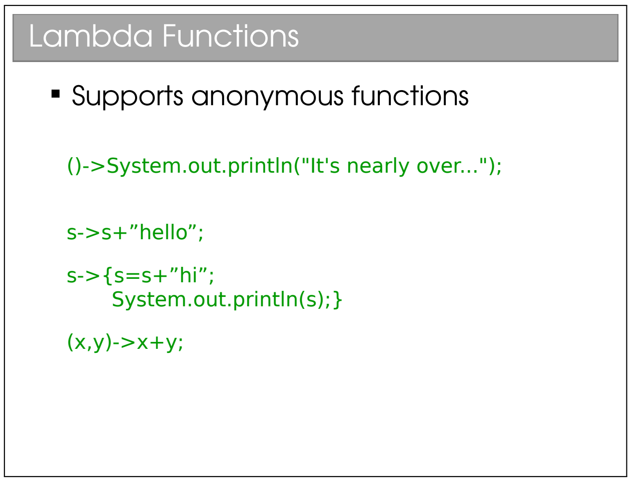
Adding Functional Elements...

- 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...

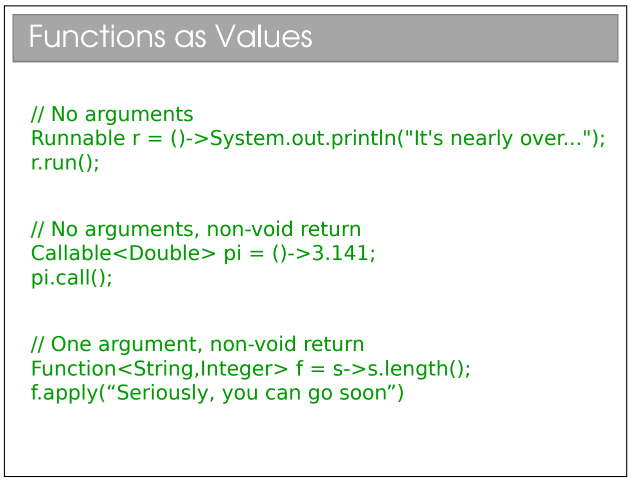
Java 8 is a major Java release. A lot has been added, some of it controversial. Not much of it relates to OOP but we discuss it partly for interest and partly because it emphasises how (big) languages tend to just subsume multiple paradigms, blurring the boundaries more and more



Lambda Functions

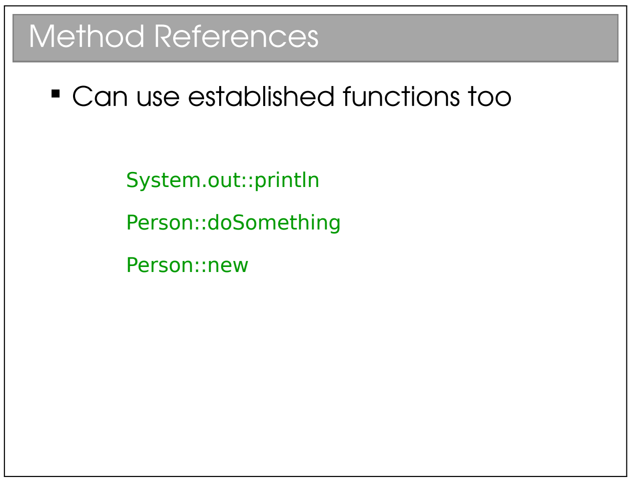
- Supports anonymous functions

```
()->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

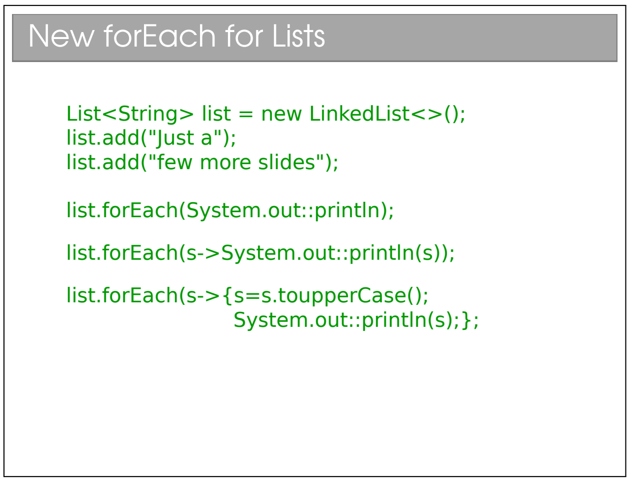
```
// No arguments
Runnable r = ()->System.out.println("It's nearly over...");
r.run();

// 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

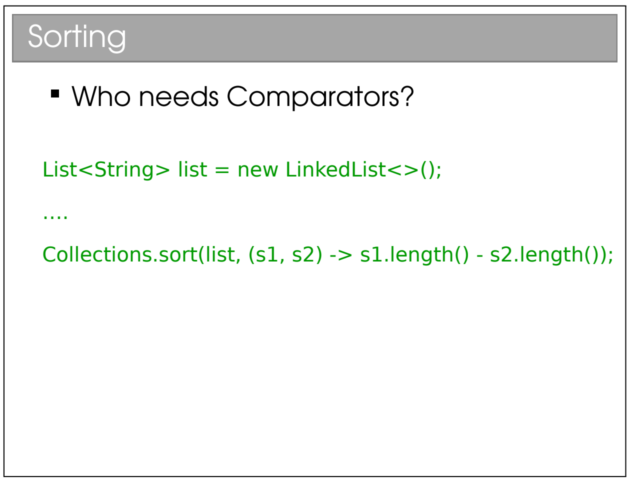
```
System.out::println
Person::doSomething
Person::new
```



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);};
```

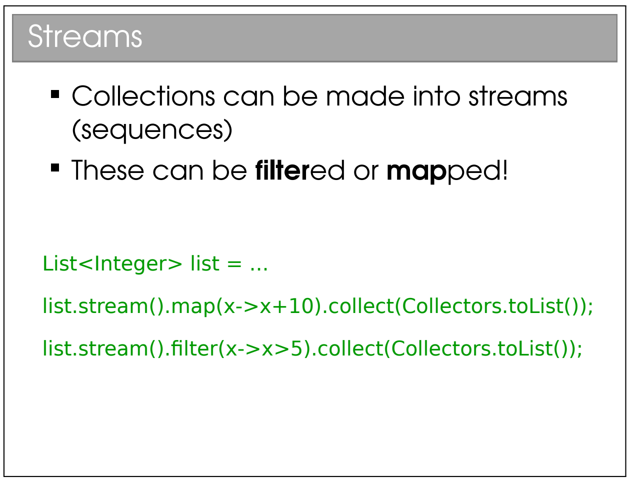
Note this is effectively our beloved ‘map‘ function from ML!



Sorting

- Who needs Comparators?

```
List<String> list = new LinkedList<>();
....
Collections.sort(list, (s1, s2) -> s1.length() - s2.length());
```



Streams

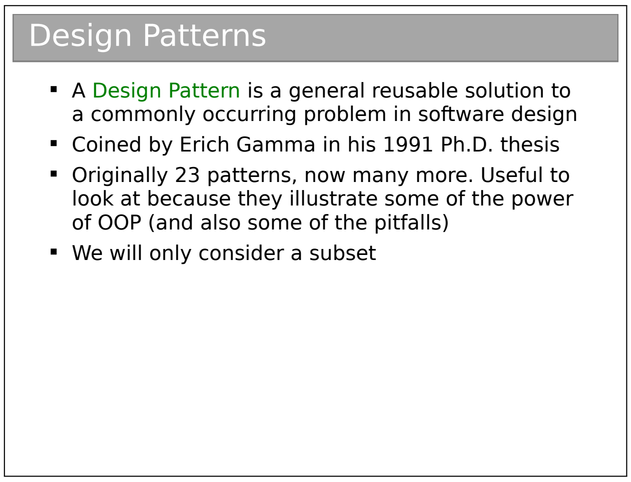
- Collections can be made into streams (sequences)
- These can be **filter**ed or **map**ped!

```
List<Integer> list = ...
list.stream().map(x->x+10).collect(Collectors.toList());
list.stream().filter(x->x>5).collect(Collectors.toList());
```

This is a more explicit introduction of the ‘filter‘ and ‘map‘ features seen in functional programming (you didn’t actually meet ‘filter‘ formally in FoCS, but it just filters a list according to some supplied predicate). However, notice how ugly the syntax has become...

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

Coding anything more complicated than a toy program usually benefits from forethought. After you've coded a few medium-sized pieces of object-oriented software, you'll start to notice the same general problems coming up over and over. And you'll start to automatically use the same solutions to them. We need to make sure that set of default solutions is a good one!

In his 1991 PhD thesis, Erich Gamma compared this to the field of architecture, where recurrent problems are tackled by using known good solutions. The follow-on book (**Design Patterns: Elements of Reusable Object-Oriented Software, 1994**) identified a series of commonly encountered problems in object-oriented software design and 23 solutions that were deemed elegant or good in some way. Each solution is known as a *Design Pattern*:

A Design Pattern is a general reusable solution to a commonly occurring problem in software design.

The modern list of design patterns is ever-expanding and there is no shortage of literature on them. In this course we will look at a few key patterns and how they are used.

11.0.1 So Design Patterns are like coding recipes?

No. Creating software by stitching together a series of Design Patterns is like painting by numbers — it's easy and it probably works, but it doesn't produce a Picasso! Design Patterns are about intelligent solutions to a series of generalised problems that you *may* be able to identify in your software. You might find they don't apply to your problem, or that they need adaptation. You simply can't afford to disengage your brain (sorry!).

11.0.2 Why Bother Studying Them?

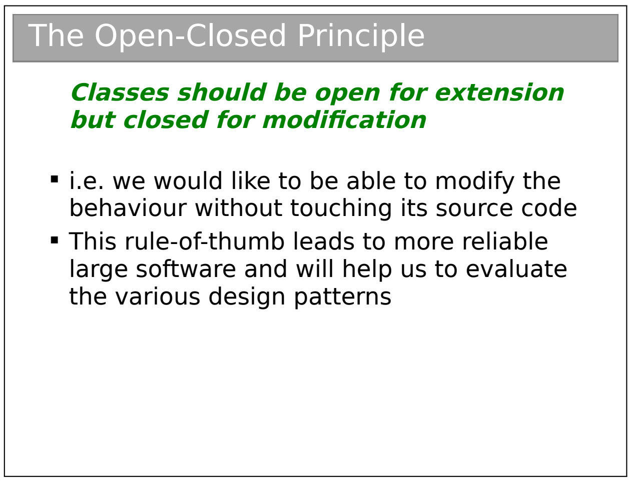
Design patterns are useful for a number of things, not least:

1. They encourage us to identify the fundamental aims of given pieces of code
2. They save us time and give us confidence that our solution is sensible
3. They demonstrate the power of object-oriented programming
4. They demonstrate that naïve solutions are bad
5. They give us a common vocabulary to describe our code

The last one is important: when you work in a team, you quickly realise the value of being able to succinctly describe what your code is trying to do. If you can replace twenty lines of comments[1] with a single word, the code becomes more readable and maintainable. Furthermore, you can insert the word into the class name itself, making the class self-describing.

[1]You are commenting your code liberally, aren't you?

11.0.3 The Open-Closed Principle



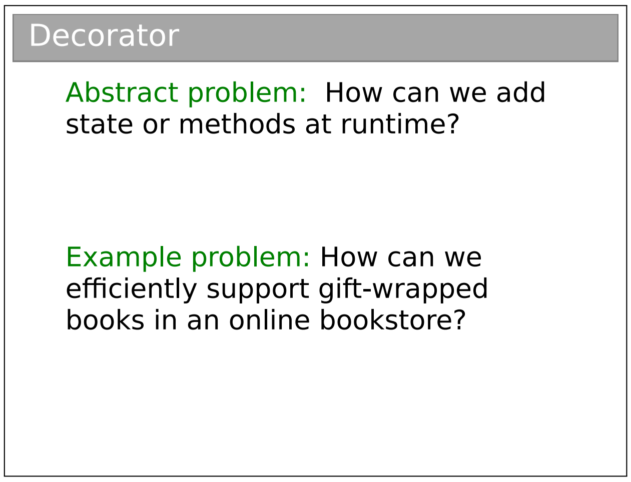
The Open-Closed Principle

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

To help understand why this is helpful, it's useful to think about multiple developers using a software library. If they want to alter one of the classes in the library, they could edit its source code. But this would mean they had a customised version of the library that they wouldn't be able to update when new (bug-reduced) versions appeared. A better solution is to use the library class as a base class and implement the minor changes that are desired in the custom child. So, if you're writing code that others will use (and you should *always* assume you are in OOP) you should make it easy for them to extend your classes and discourage direct editing of them.

11.0.4 The Decorator Pattern



Decorator

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

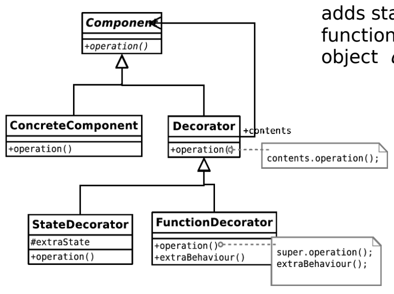
Example problem: How can we efficiently support gift-wrapped books in an online bookstore?

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

Solution 2: Extend Book to create WrappedBook.

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.

Decorator in General



- The decorator pattern adds state and/or functionality to an object *dynamically*

So we take an object and effectively give it extra state or functionality. I say 'effectively' because the actual object in memory is untouched. Rather, we create a new, small object that 'wraps around' the original. To remove the wrapper we simply discard the wrapping object. Real world example: humans can be 'decorated' with contact lenses to improve their vision.

Note that we can use the pattern to add state (variables) or functionality (methods), or both if we want. In the diagram above, I have explicitly allowed for both options by deriving StateDecorator and FunctionDecorator. This is usually unnecessary — in our book seller example we only want to decorate one thing so we might as well just put the code into Decorator.

11.0.5 The Singleton Pattern

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.

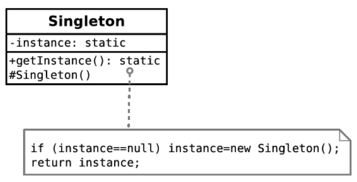
A valid solution to this is to make sure you close the database connection after using it, so you can just create Database objects every time you have a query. However, what if you forgot to close it? And what if making the connection was slow (they always are in computer time...).

Instead we exploit our access modifiers and create a private constructor (to ensure no-one can create objects at will) and add in a static member (the only instance we will ever have). Finally, we include a static getter for this member.

Ideally the instantiation of the Database should be *lazy*—i.e. only done on the first call to the getter.

Singleton in General

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



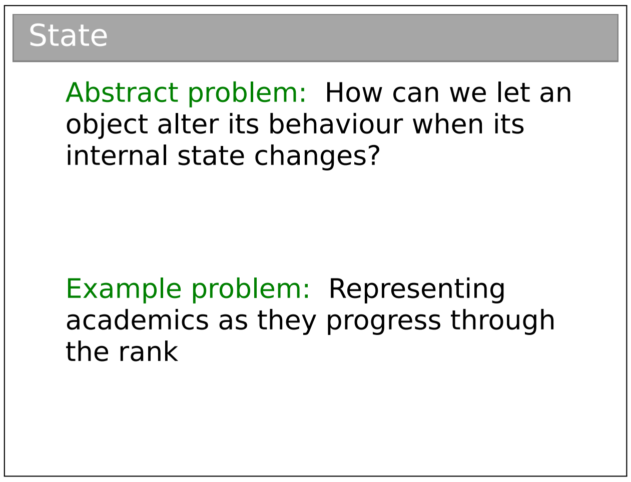
```
if (instance==null) instance=new Singleton();
return instance;
```

There is a caveat with Java. If you choose to make the constructor protected (this would be useful if you wanted a singleton base class for multiple applications of the singleton pattern, and is actually the 'official' solution) you have to be careful.

Protected members are accessible to the class, any subclasses, *and all classes in the same package*. Therefore, any class in the same package as your base class will be able to instantiate Singleton objects at will, using the new keyword!

Additionally, we don't want a crafty user to subclass our singleton and implement Cloneable on their version. How could you ensure this doesn't happen?

11.0.6 The State Pattern



State

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

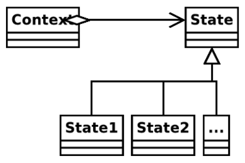
Example problem: Representing academics as they progress through the rank

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

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.

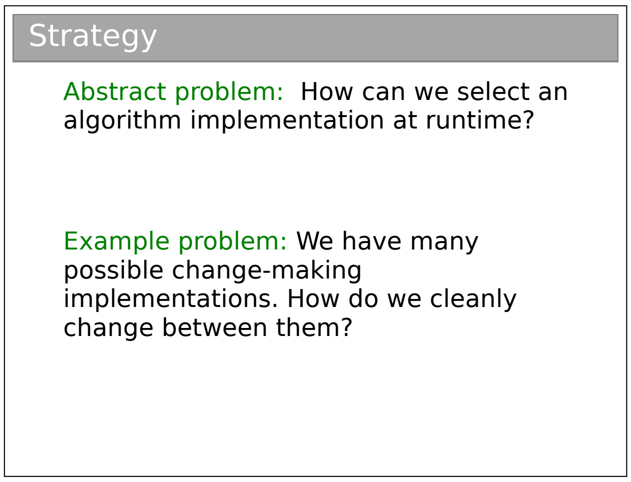
Solution 3: (State) Make Academic a concrete class that has-a AcademicRank as a member. Use AcademicRank as a base for Lecturer, Professor, etc., implementing the rank-specific behaviour in each..

State in General



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

11.0.7 The Strategy Pattern



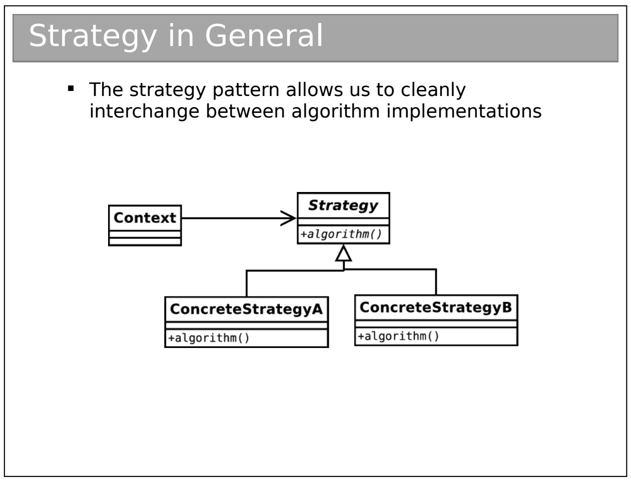
Strategy

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.



Strategy in General

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

Note that this is essentially the same UML as the State pattern! The *intent* of each of the two patterns is quite different however:

- State is about encapsulating behaviour that is linked to specific internal state within a class.
- Different states produce different outputs (externally the class behaves differently).
- State assumes that the state will continually change at run-time.
- The usage of the State pattern is normally invisible to external classes. i.e. there is no setState(State s) function.

- Strategy is about encapsulating behaviour in a class. This behaviour does not depend on internal variables.
- Different concrete Strategys may produce exactly the same output, but do so in a different way. For example, we might have a new algorithm to compute the standard deviation of some variables. Both the old algorithm and the new one will produce the same output (hopefully), but one may be faster than the other. The Strategy pattern lets us compare them cleanly.
- Strategy in the strict definition usually assumes the class is selected at compile time and not changed during runtime.
- The usage of the Strategy pattern is normally visible to external classes. i.e. there will be a setStrategy(Strategy s) function or it will be set in the constructor.

However, the similarities do cause much debate and you will find people who do not differentiate between the two patterns as strongly as I tend to.

11.0.8 The Composite Pattern

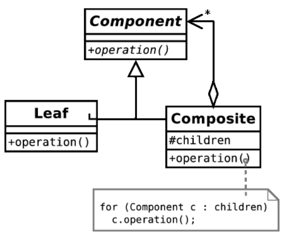
Composite

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

The solution is fairly straightforward. We want to be able to treat a group of DVDs to just like a single DVD, so BoxSet inherits from DVD. To avoid repeating the description information and to keep pricing in sync, BoxSet must also have access to the constituent DVD objects.

Composite in General



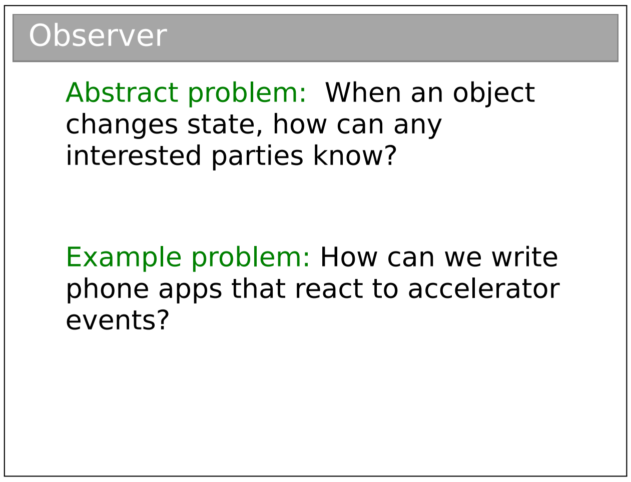
```
for (Component c : children)
  c.operation();
```

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

If you're still awake, you may be thinking this looks like the Decorator pattern, except that the new class supports associations with multiple DVDs (note the * by the arrowhead). Plus the intent is different—we are not adding new functionality to objects but rather supporting the same functionality for groups of those objects.

If you try to make a graphical representation of composites, you'll end up with some form of tree with each composite a node and each single entity a leaf. Many texts use this terminology when discussing the composite pattern.

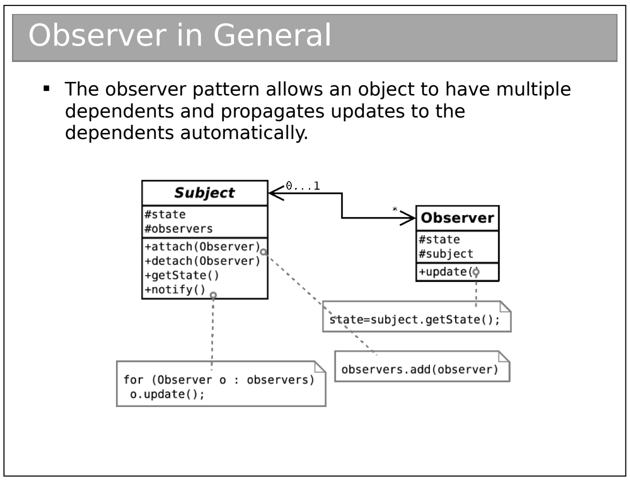
11.0.9 The Observer Pattern



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 in General

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

This pattern is used regularly, but is particularly useful for event-based programs. The process is analogous to a magazine subscription: you *subscribe* with the publisher in order to receive *publish* events (magazines) as soon as they are available. In design patterns parlance, you are an observer of the publisher, who is the subject. It should be clear that this is also a very important pattern for the various proxy implementations if the source information might change during use.

In an Android smartphone, the system provides a subject in the form of a SensorManager object, which is actually a singleton (only one manager at any time). So we get it by calling:

```
SensorManager sManager = (SensorManager)
   getSystemService(SENSOR_SERVICE);
```

We then register with it with a line like:

```
sManager.registerListener(this,
        sManager.getDefaultSensor(
          Sensor.TYPE_ACCELEROMETER),
        SensorManager.SENSOR_DELAY_NORMAL);
```

Our class must implement SensorEventListener, which forces us to specify a onSensorEvent() method. Whenever the system gets a new accelerometer reading, it cycles over all the objects that have registered with it, feeding them the new reading.

11.0.10 Classifying Patterns

Often patterns are classified according to what their intent is or what they achieve. The original book defined three classes:

Creational Patterns . Patterns concerned with the creation of objects (e.g. Singleton, Abstract Factory).

Structural Patterns . Patterns concerned with the composition of classes or objects (e.g. Composite, Decorator, Proxy).

Behavioural Patterns . Patterns concerned with how classes or objects interact and distribute responsibility (e.g. Observer, State, Strategy).

11.0.11 Other Patterns

You've now met a few Design Patterns. There are plenty more (23 in the original book and many, many more identified since), but this course will not cover them. What has been presented here should be sufficient to:

- Demonstrate that object-oriented programming is powerful.
- Provide you with (the beginnings of) a vocabulary to describe your solutions.
- Make you look critically at your code and your software architectures.
- Entice you to read further to improve your programming.

Of course, you probably won't get it right first time (if there even is a 'right'). You'll probably end up *refactoring* your code as new situations arise. However, if a Design Pattern *is* appropriate, you should probably use it.

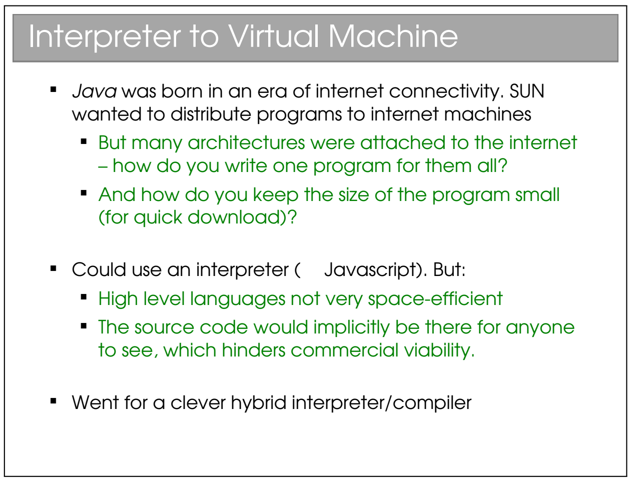
11.0.12 Performance

Note that all of the examples here have concentrated on structuring code to be more readable and maintainable, and to incorporate constraints structurally where possible. At no point have we discussed whether the solutions *perform* better. Many of the solutions exploit runtime polymorphic behaviour, for example, and that carries with it certain overheads.

This is another reason why you can't apply Design Patterns blindly. [This is a good thing since, if it wasn't true, programming wouldn't be interesting, and you wouldn't get jobs!].

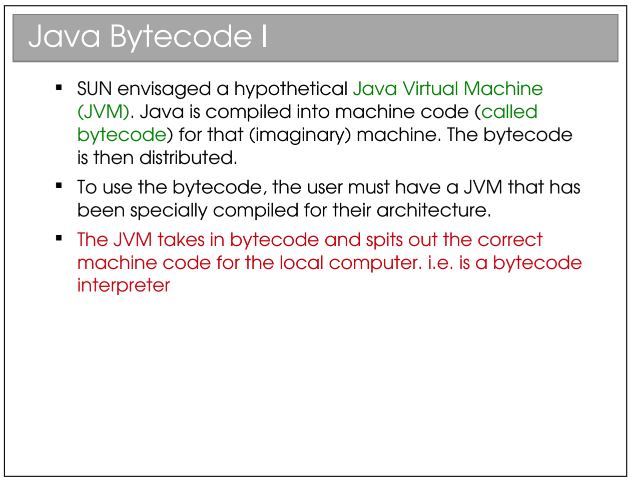
Appendix I: Java, the JVM and Bytecode

Java is known for its cross-platform abilities, which has given it strong internet credentials. Being able to send a file compiled on one machine to another machine with a different architecture and have it run is a neat trick. It shouldn't work because the machine code for one machine shouldn't make sense to another.



Interpreter to Virtual Machine

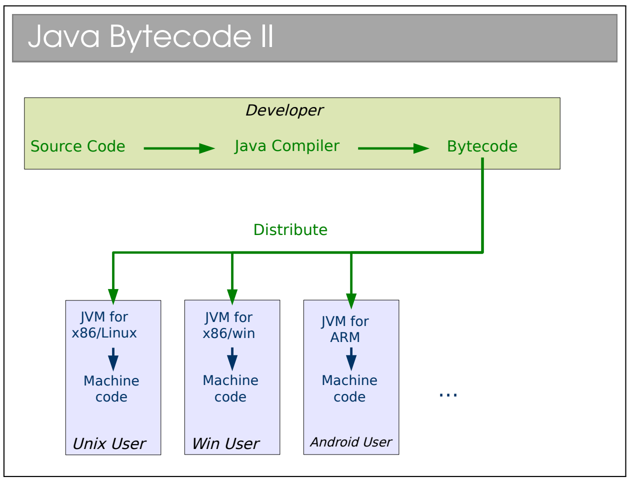
- *Java* was born in an era of internet connectivity. SUN wanted to distribute programs to internet machines
  - But many architectures were attached to the internet – how do you write one program for them all?
  - And how do you keep the size of the program small (for quick download)?
- Could use an 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

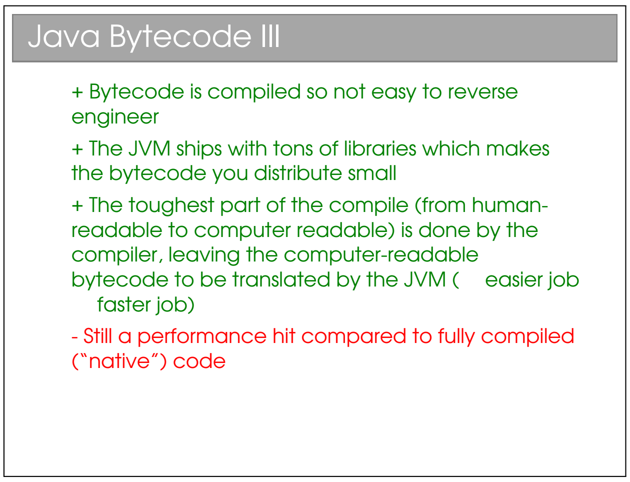


Java Bytecode I

- 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

So the trick is to *partially* compile the Java code to a machine code for a universal machine (that doesn't actually exist). To actually *use* this special machine code ("bytecode") a machine must translate from bytecode to its own local machine code. To that it must have a Java Virtual Machine (JVM) installed that knows the translation.





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