CS1092 : Inheritance in Java
(#3 of 4)

Gavin Brown

Kilburn Building, rm 2.81

gavin.brown@cs.man.ac.uk

www.cs.man.ac.uk/~gbrown/teaching/java/

REMINDER OF YESTERDAY

Making use of polymorphism

Or even more useful, an array of Vehicle objects:

```
Vehicle [] vehicleList = new Vehicle[4];

vehicleList[0] = new Porsche();
vehicleList[1] = new Bicycle();
vehicleList[2] = new Plane();
vehicleList[3] = new Porsche();

for (int i=0; i<4; i++)
    System.out.println(vehicleList[i].numberOfWheels)
```

Let's run that code:

```
> java test
4
2
6
4
```

REMINDER OF YESTERDAY

Abstract class: Vehicle

```
public abstract class Vehicle
{
       protected int numberOfWheels;

       public int getNumWheels() {
           return numberOfWheels;
       }

       public abstract void turn();
}
```

Forces ALL subclasses to have a "turn()" method

Abstract classes : *Useful software engineering tool: when working in a team, write an abstract class and give it to a colleague to work from. You can provide some functionality, and impose some rules, like the above class making the rule that subclasses should have a turn() method - causes compile errors without it!*

Today

1. *dynamic method binding*
2. *'final' keyword to control inheritance*
3. *is-a and has-a rules*
4. *constructor chaining*
5. *class **Object** (including inheritance in the Java API)*

A new example

```
public class Cat {
   public void talk() {
      System.out.println(“Miaow!”);
   }
}


public class test {
   public static void main( String [] args ) {
      Cat tiddles = new Cat();
      tiddles.talk();
   }
}
```

```
> javac test.java
> java test
Miaow!
```

A new example

```
public abstract class Animal {
   public abstract void talk();
}
```

```
public class Cat extends Animal {
   public void talk() {
      System.out.println(“Miaow!”);
   }
}
```

```
public class Dog extends Animal {
   public void talk() {
      System.out.println(“Bark!”);
   }
}
```

Dynamic method binding

```
public abstract class Animal
public class Cat extends Animal
public class Dog extends Animal
```

```
Animal myPet = null;

Double d = Math.random();
if (d > 0.5)   myPet = new Dog();
        else   myPet = new Cat();

myPet.talk();
```

Bark or Miaow?

We don't know until we run the code, and neither does the compiler – so it DYNAMICALLY figures out which talk() method to call.

Dynamic method binding

```
public abstract class Employee
public class Worker extends Employee
public class Manager extends Employee
```

```
Employee bob = new Worker();

bob.paySalary();  //pays Bob as a worker

if (promotionReceived(bob)) {
    bob = new Manager();
}
bob.paySalary();  //pays Bob as a manager
```

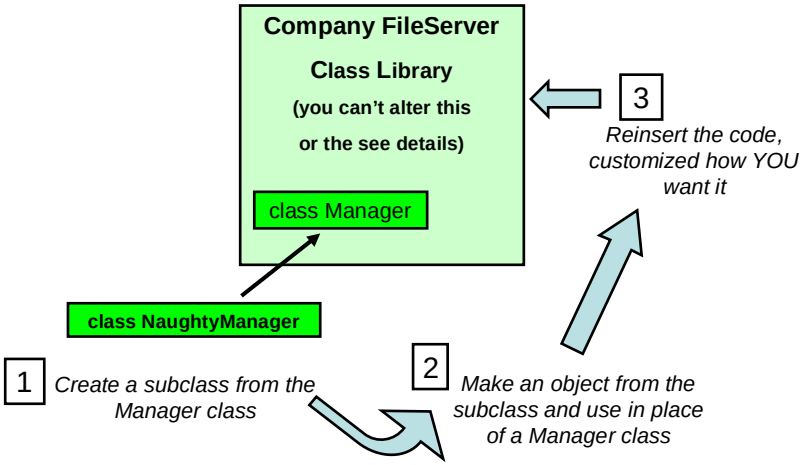
Same variable, same type (Employee), but a different method is called!

```
public abstract class Employee {
        private String taxCode;
        public Employee( String code ) {
                taxCode = code;
        }

        public abstract double salaryBeforeTax();

        public double salaryAfterTax() {
                double pay = salaryBeforeTax();

                if( taxCode.equals("LOW") )
                        pay = pay - (pay*0.24);  //low rate
                else
                        pay = pay - (pay*0.40);  //high rate

                return pay;
        }
}
```

```
public class Manager extends Employee {
        private double bonus;
        private double fixedWage = 40000;
        public Manager( double b ) {
                super("HIGH");
                bonus = b;
        }
        public double salaryBeforeTax() {
                return fixedWage+bonus;
        }
}
```

```
public class Worker extends Employee {
        private double numHours;
        private double hourlyRate=5.50;
        public Worker( double h ) {
                super("LOW");
                numHours = h;
        }
        public double salaryBeforeTax() {
                return numHours*hourlyRate;
        }
}
```

Controlling Inheritance – how it can be misused…

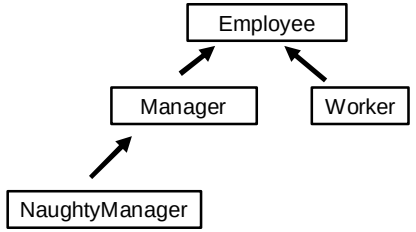


Meanwhile, deep in the Company Compute Server…

```
if (bob instanceof Manager)
       System.out.println("Valid Manager");
```

Evaluates to TRUE because Bob is a Manager and a NaughtyManager!

Polymorphism!



What can we do with this kind of power then…?!

```
public class NaughtyManager extends Manager {

      public NaughtyManager( int b ) {
            super(b);
      }

      public double salaryAfterTax() {
            double pay = salaryBeforeTax();

            pay = pay - (pay*0.24);  //low rate!

            return pay;
      }
}
```

```
public class test
{
  public static void main( String []args)
  {
     Employee bob = new Manager( 15000 );

     if (bob instanceof Manager)
        System.out.println("Pay Bob : " +bob.salaryAfterTax());

     bob = new NaughtyManager( 15000 );

     if (bob instanceof Manager)
        System.out.println("Pay naughty Bob : "+bob.salaryAfterTax());
  }
}
```

```
> javac test.java
> java test
Pay Bob : 33000.0
Pay naughty Bob : 41800.0
```

Bob is now on the lower tax bracket!

```
public class Employee {
. . .
        final public double salaryAfterTax() {
                double pay = salaryBeforeTax();

                if( taxCode.equals("LOW") )
                        pay = pay - (pay*0.24);  //low rate
                else
                        pay = pay - (pay*0.40);  //high rate

                return pay;
        }
. . .
```

Make the method 'final' – and he can't do it any more!

```
> javac test.java
NaughtyManager.java:14: salaryAfterTax() in NaughtyManager cannot
override salaryAfterTax() in Employee; overridden method is final
        public double salaryAfterTax() {
                      ^
1 error
```

Or we could make the entire class 'final' !

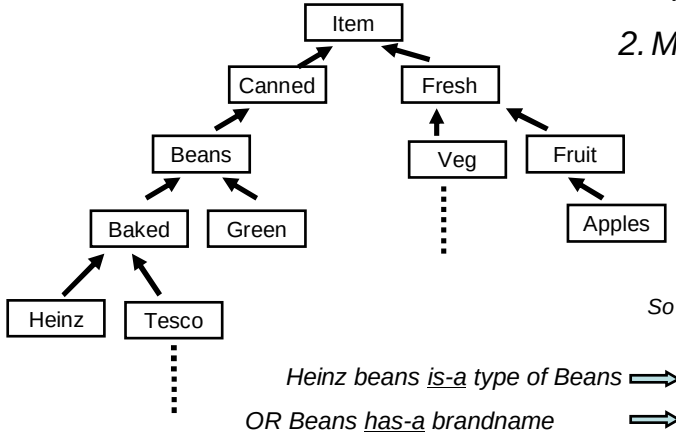
```
public final class Manager {

. . .
}
```

```
> javac test.java
NaughtyManager.java:1: cannot inherit from final Manager
public class NaughtyManager extends Manager {
                                    ^
1 error
```

is-a and has-a rules: trade-offs to be made…
Too many classes are costly…

1. Speed!

2. Memory!



So we represent it as…

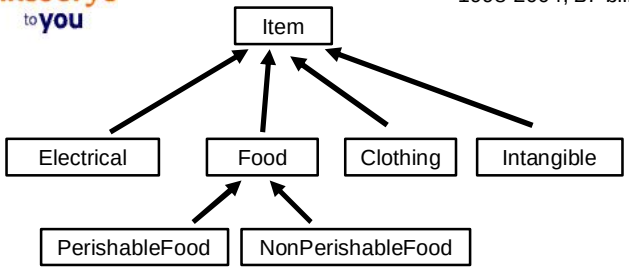
Heinz beans is-a type of Beans ⇒ *a class*

OR Beans has-a brandname ⇒ *a variable*

```
public class Beans {
    private String brandname;
```

accenture > Sainsbury's to you

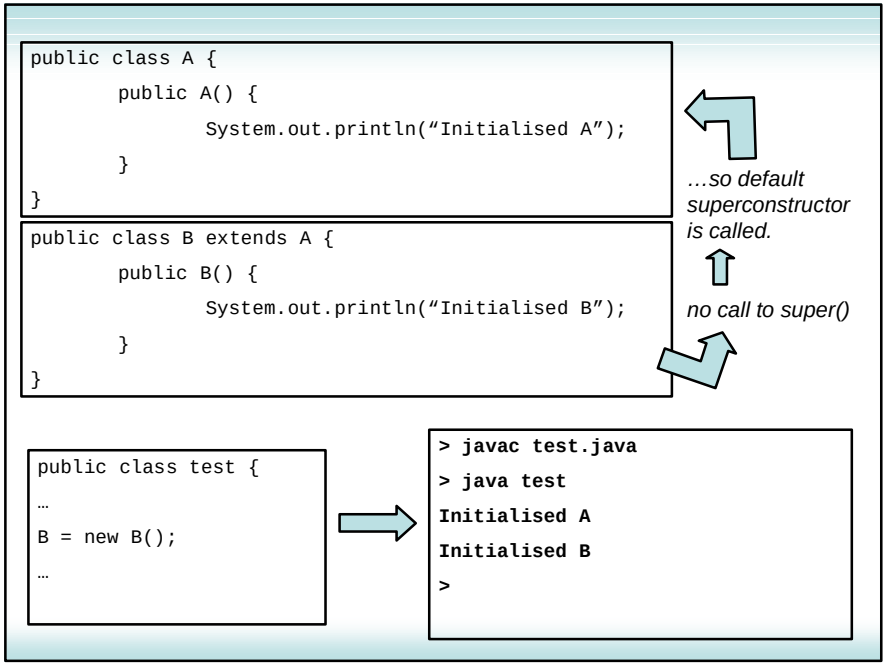
- Accenture-Sainsburys project.
- Automate stock inventory.
- 1998-2004, £7 billion.

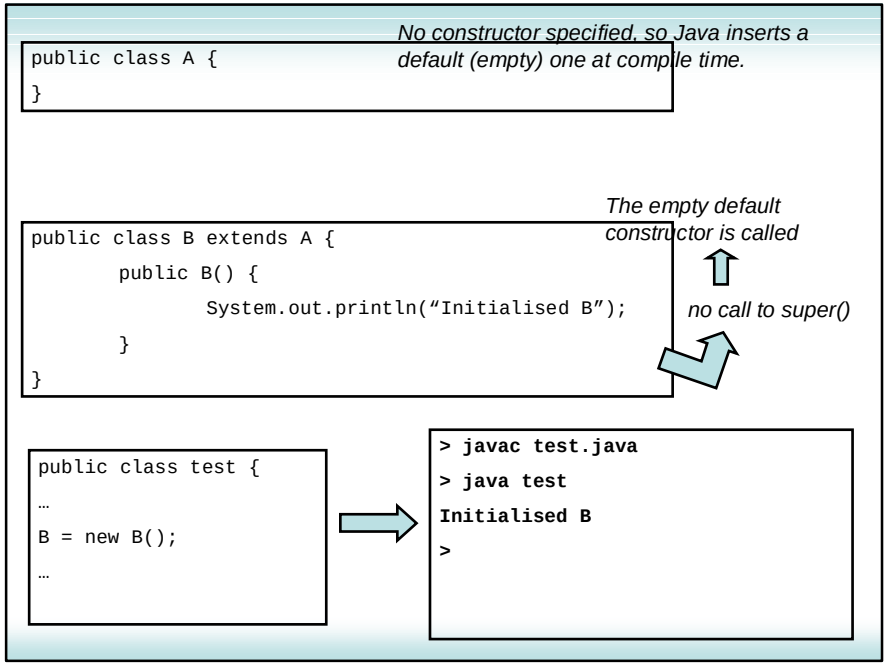


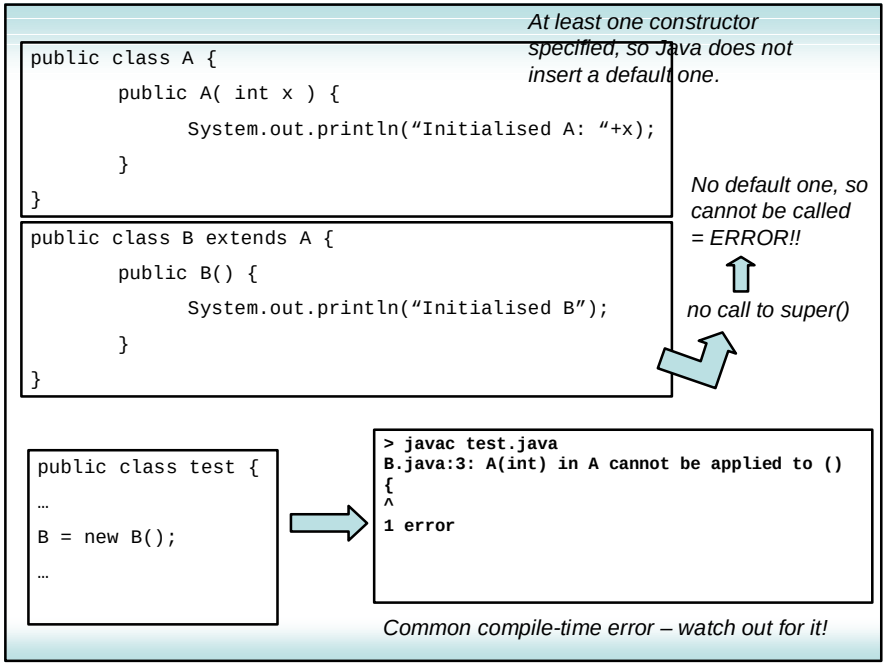
Follow the is-a and has-a rules sometimes, but in reality trade-offs are made, according to **speed**, **memory usage**, **business & pricing models**.

Priorities differ according to business needs – so will the class design.

5 minutes…







```
public class A {
        public A( int x ) {
              System.out.println("Initialised A: "+x);
        }
}
```

At least one constructor specified, so Java does not insert a default one.

```
public class B extends A {
        public B() {
              super(99999);
              System.out.println("Initialised B");
        }
}
```

call to super(int)

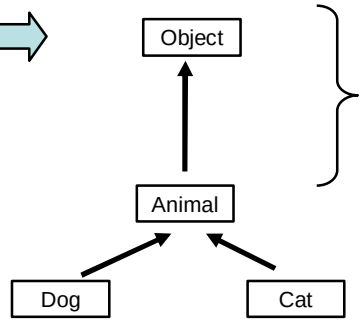
```
public class test {
…
B = new B();
…
```

```
> javac test.java
> java test
Initialised A: 99999
Initialised B
```

A class called "Object"

Because of constructor chaining, the constructor up here gets called first.

This does all the memory allocations necessary for your objects.



Every Java class implicitly extends "Object".

…automatically done for you!

If you could make the subclass constructor be called first, your objects would be working without any memory allocation!

Inheritance in the Java API…

Today

1. *dynamic method binding*
2. *'final' keyword to control inheritance*
3. *is-a and has-a rules*
4. *constructor chaining*
5. *class **Object** (including inheritance in the Java API)*

Tomorrow

1. *Case study – investments*
2. *More on using these tools when working in a team*
3. *Wrapping up and relation to rest of the course*