# Software Development (CS2500)

**Lecture 24**: Creating Classes from Other Classes

M.R.C. van Dongen

November 29, 2010

## Contents

**1 Introduction** **1**

**2 Chair Wars Revisited** **2**
2.1 Introduction . . . . . 2
2.2 Brad Explains . . . . . 2

**3 Inheritance** **3**
3.1 Introduction . . . . . 3
3.2 Case Study . . . . . 4
3.3 The Class Diagram . . . . . 5

**4 The Fota Challenge** **6**
4.1 The Challenge . . . . . 6
4.2 Larry Presents his Solution . . . . . 7
4.3 Brad Presents his Solution . . . . . 9
4.4 Collecting the Prize . . . . . 11

**5 For Wednesday** **11**

# 1 Introduction

This lecture is about *inheritance*. Inheritance lets you share common code. The common code is written in a common superclass. The common superclass implements common behaviour. Subclasses inherit common behaviour from their superclass. We shall carry out two case studies.

# 2 Chair Wars Revisited

## 2.1 Introduction

Remember Larry and Brad? Brad's final solution had five classes:

- One `Shape` *superclass* for default, common `Shape` behaviour.
- A separate class `Square`, `Circle`, `Rectangle`, and `Amoeba` for each different shape.
- All these four classes were subclasses of `Shape`.
- All inherited default behaviour from `Shape`.
- The `Amoeba` class overrode behaviour for `playSound`, and `rotate`.
- By overriding these methods, `Amoeba` objects could do things in a different way.

Larry thought Brad's final class had lots of duplicated code: Larry thought that all subclasses had the same code for `playSound` and `rotate`. Of course code duplication is any software engineer's worst nightmare. If you start duplicating code then you end up with lots of almost similar copies. If global changes are required then you have to make a change for *each* copy. This has several disadvantages.

- As a result of this you lose time.
- For each file you edit, you can make errors. Clearly, making a change to one class file is less prone to errors than making the same change to several class files.

But then Brad explained his design.

## 2.2 Brad Explains

Figure 1 depicts Brad's final design. Brad had one superclass called `Shape` and four subclasses called `Square`, `Circle`, `Triangle`, and `Amoeba`. His superclass `Shape` class defines two methods called `rotate` and `playSound`. His subclasses also define these method. Brad thought that this was all copied and pasted into the subclass files. (Of course he hadn't a clue about OO design.)

In reality, Brad's subclasses `Square`, `Circle`, and `Triangle` *inherited* the behaviour of the methods `rotate` and `playSound` from the superclass `Shape`. The nice thing about `Java` inheritance is that it works without code duplication. What is more, it works *without* the need to write any code for the methods inside the subclasses.

Brad's `Amoeba` class worked differently. It also was a subclass of the `Shape` superclass but here the subclass *overrode* the definitions for the methods `rotate` and `playSound`. This was done by just writing down new definitions for `rotate` and `playSound`. By providing these new definition Brad provided more specific behaviour for these methods in the `Amoeba` class. All this was done without touching existing code.

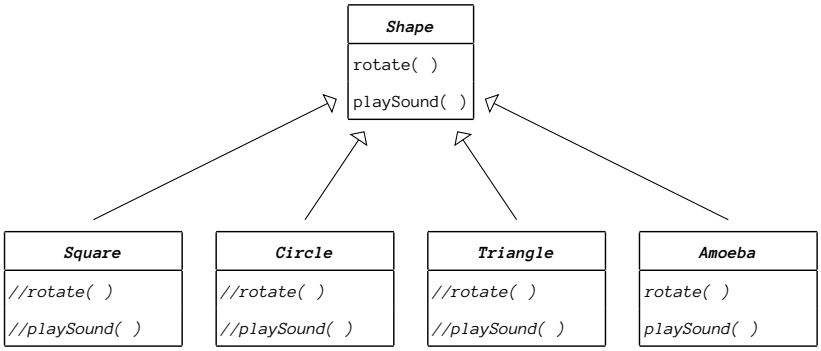


Figure 1: Brad's final class design.

# 3 Inheritance

## 3.1 Introduction

This section explains the basic ideas behind inheritance. By the end of this section you should understand enough about inheritance to start using it in Java.

There are two main advantages of *inheritance*:

- Inheritance increases the ability to reuse implementation effort.
- Inheritance separates class-specific code from more general code. As we saw in Lecture 6 this allowed Brad to make local changes in a class without affecting code in other classes.

Code is structured in classes so as to maximise code reuse. *Common* code is put in a common, *more abstract* class. The common, more abstract class is called the *superclass*. The code in the superclass is shared by *subclasses*. The subclasses are more *specific*. The functionality which is provided by the superclass is also provided by the subclasses.[1] So if the superclass has a method then so does the subclass. Here, the subclass is said to *inherit* the method from its superclass.

However, the subclass functionality may be more *specific*. For example, the subclass may implement a method in a *different* way. When this happens, the subclass is said to *override* the behaviour from its superclass. A subclass may also have *additional* behaviour which is not provided by its superclass. Again, this means that the subclass is more specific. Because a subclass is more specific than its superclass, the subclass is said to *extend* its superclass. (The functionality is extended.)

[1] Here functionality is used in an informal way. We really mean methods and attributes.

## 3.2 Case Study

The following example is instructive. Let's suppose we have a `Surgeon` and a `GP` class. Let's also suppose we have a `Doctor` class. *Both* `Surgeons` and `GPs` are `Doctors`: they are more specific.

- A `Surgeon` *is-a* `Doctor`.
- A `GP` *is-a* `Doctor`.
- So the `Surgeon` and `GP` classes *extend* the `Doctor` class.

*Both* `Surgeons` and `GPs` have a method called `treatPatient( )`. *Any* `Doctor` has it. *Both* have a property `worksAtHospital` (a `boolean`). *Any* `Doctor` has it. For a `Surgeon` it is `true`. For a `GP` it is `false`. For sake of the example, we shall assume that `worksAtHospital` is a `public` instance variable.[2]

The following is where `Surgeons` and `GPs` differ from `Doctors` *in general*:

**`Surgeon:`** The following is `Surgeon`-specific behaviour:

- A `Surgeon` has an additional `makeIncision( )` method. This is *special* behaviour which is not provided by all `Doctors`.
- A `Surgeon` has a *special* implementation for `treatPatient( )`. This special implementation *overrides* the default `treatPatient( )` implementation in the `Doctor` class.

**`GP:`** The following is `GP`-specific behaviour:

- A `GP` has a `boolean` instance variable called `makesHouseCalls`. To simplify this example, we shall assume that `makesHouseCalls` is a `public` instance variable.
- A `GP` has an *additional* method `giveAdvice( )`.

We put the *more general* code in the `Doctor` class. This is the code which *any* `Doctor` should provide: `Surgeons` and `GPs` in particular.

```Java
public class Doctor {
    public boolean worksAtHospital;

    public void treatPatient( ) {
        // Default patient treatment.
    }

    public void chargePatient( ) {
        // Let's face it, they all do.
    }
}
```

As a little joke, we also add a method called `chargePatient`, which should be provided by any `Doctor`.

We put the *more specific* code in the classes that *extend* the `Docter` class: the `Surgeon` and `GP` classes. The following is the code for the `Surgeon` class.

[2] But remember that this is poor practice.

```java
public class Surgeon extends Doctor {
    public Surgeon( ) {
        worksAtHospital = true;
    }

    @Override
    public void treatPatient( ) {
        // Specific patient treatment.
    }

    public void makeIncision( ) {
        // Additional behaviour.
    }
}
```

The keyword `extends` is new. It does just what is says on the tin: it states that the `Surgeon` class *extends* the `Doctor` class. Using this keyword tells `Java` that the `Surgeon` class is a *subclass* of the `Doctor` class.

We've already seen the '`@Override`' annotation. It is used to make explicit which methods are overridden. For example, we've been using the notation from Day 1 each time we overrode the method `toString` (we've been exploiting inheritance without realising it). Getting back to the example, overriding the method `treatPatient` tells `Java` that the new implementation of the method `treatPatient` *overrides* the default `treatPatient` behaviour from the `Doctor` class, which is the *superclass* of the `Surgeon` class.

Notice that the `Surgeon` class does not have a declaration for `worksAtHospital`: it *inherits* the attribute from the `Doctor` (super)class.

The following is the code for the `GP`. Notice that the `GP` class does not have a declaration for `treatPatient` because it *inherits* the default implementation of this method from its *superclass*, the `Doctor` class.

```java
public class GP extends Doctor {
    public boolean makesHouseCalls;

    public GP( boolean makesHouseCalls ) {
        worksAtHospital = false;
        this.makesHouseCalls = makesHouseCalls;
    }

    public void giveAdvice( ) {
        // Additional behaviour.
    }
}
```

## 3.3 The Class Diagram

Figure 2 depicts the class design of this example. The superclass, `Doctor`, defines the common attributes and methods. The subclasses are `Surgeon` and `GP`. The attributes and methods which are inherited by the subclasses are commented out. The `GP` inherits all common attributes and methods. The `Surgeon` inherits the common attribute `worksAtHospital` and the common method `chargePatient( )`. It overrides the method `treatPatient( )`. The `Surgeon` class defines a class-specific method called `makeIncision( )`. The `GP` class defines a class-specific method called `giveAdvice( )`. In addition it defines a class-specific attribute which is called `makesHouseCalls`.

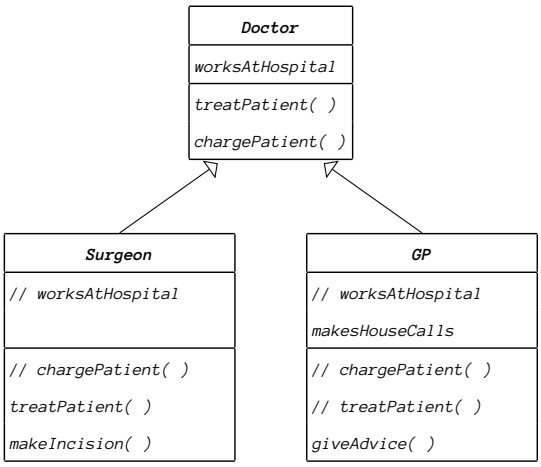


Figure 2: Class design of `Doctor` example.

# 4 The Fota Challenge

This section studies another example. The example is about implementing an application for Fota Wildlife Park. The main actors in this example are Larry and Brad, who also featured in the presentation of Lecture 6. The following is the rough outline of this section, which is presented as a play in four acts.

**Act I:** The Challenge.

**Act II:** Larry Presents his Solution.

**Act III:** Brad Presents his Solution.

**Act IV:** Collecting the prize.

For ease of presentation most code violates one or several coding conventions. This does not mean this is acceptable coding practice. The only reason why this is done is to simplify the presentation.

## 4.1 The Challenge

Fota Wildlife Park has lots of animals: a lion, a cat, a wolf, a tiger, a dog, and they're even getting a hippo. Each animal:

- Has a picture `String`;
- Has a certain kind of food: grass or meat;
- Has an integer hunger level, which corresponds to the amount of food they need per day;

- Eats;
- Makes noise; and
- Has a roaming behaviour.

Larry and Brad have been asked to work on this application. The programmer with the most impressive implementation wins a prize: fish and chips at Lennoxes.

Since his Aeron defeat in Lecture 6, Larry has secretly been taking Java lessons with Amy, who snatched the Aeron from Larry and Brad. He has just started learning about inheritance. He knows inheritance is the key to solving this problem. He just knows he will beat Brad.

Brad was also delighted with this application. This was a textbook example of an inheritance application. He knew this can't be too difficult.

## 4.2 Larry Presents his Solution

Larry quickly identifies the objects: the animals. Following Brad's shape-application example, he created an `Animal` class. He putall the common methods and attributes in this class. This is what it looked like.

```Java
public class Animal {
    public String  picture;
    public boolean eatsGrass;
    public int     hunger;

    ⟨more⟩
}
```

The rest of his class was as follows:

```Java
public Animal( String picture,
               boolean eatsGrass,
               int hunger ) {
    this.picture   = picture;
    this.eatsGrass = eatsGrass;
    this.hunger    = hunger;
}

public void eat( ) {          // Default eating behaviour.
    System.out.println( "Eating " + hunger + " portions of " + food( ) + "." );
}
private String food( ) {
    return (eatsGrass ? "grass" : "meat");
}
public void makeNoise( ) { } // Should be overridden.
public void roam( ) { }      // Should be overridden.
public String toString( ) {
    ⟨omitted⟩
}
```

Next Larry started implementing his `Animal` subclasses. The following was his first subclass, the `Hippo` class. (The other classes were similar in design.)

```Java
public class Hippo extends Animal {
    private static final int HIPPO_HUNGER = 10;
    private static final String HIPPO_PICTURE = "hippo.jpg";

    public Hippo( ) {
        picture = HIPPO_PICTURE;
        eatsGrass = true;
        hunger = HIPPO_HUNGER;
    }

    public void roam( ) {
        System.out.println( "I'm Lazy: not roaming." );
    }

    public void makenoise( ) {
        System.out.println( "Grunt." );
    }
}
```

Note that the instance variables `picture`, `eatsGrass`, and `hunge` are not defined in the `Hippo` class but are inherited from the `Animal` class.

Larry's `main` class looked as follows:

```Java
import java.util.ArrayList;

public class Main {
    public static void main( String[] args ) {
        ArrayList<Animal> animals = new ArrayList<Animal>( );

        animals.add( new Dog( ) );
        animals.add( new Cat( ) );
        animals.add( new Hippo( ) );
        for (Animal animal : animals) {
            System.out.println( "next: " + animal );
            animal.roam( );
            animal.eat( );
            animal.makeNoise( );
        }
    }
}
```

Larry quickly tested his application and all looked fine. Het got the following output:

```Unix Session
$ java Main
next: Animal[ picture = dog.jpg, eatsGrass = meat, hunger = 4 ]
Roaming in my pack.
Eating 4 portions of meat.
Arf. Arf.
next: Animal[ picture = cat.jpg, eatsGrass = meat, hunger = 1 ]
Roaming alone.
Eating 1 portions of meat.
Mew. Mew.
next: Animal[ picture = hippo.jpg, eatsGrass = grass, hunger = 10 ]
I'm Lazy: not roaming.
Eating 10 portions of grass.
```

Figure 3 depicts Larry's class design in the form of a class diagram. When Larry showed his class design and the output of his program to his boss, his boss wasn't impressed. "The hippo makes no noise." he said.

Larry couldn't understand it. He thought he had properly overridden the method for noise making

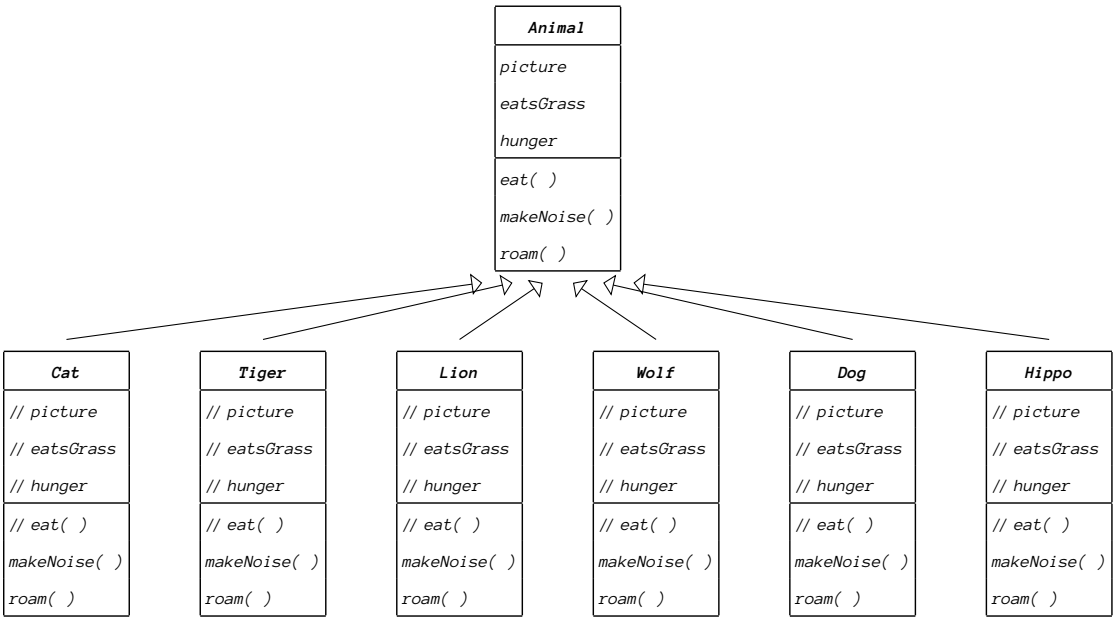


Figure 3: Larry's class design.

in the `Hippo` class. When he asked Amy at his nect `Java` lesson, she quickly discovered the error. There was a typo in his `Hippo` class. Instead of `makeNoise` Larry had typed `makenoise`. Amy told him how to prevent such errors. "All you have to do is add the '`@Override`' annotation." she said. Larry couldn't stand it. Again he had made a fool of himself.

## 4.3 Brad Presents his Solution

Brad had read about Lennoxes in the 2009 Edition of *The Lonely Planet*. Eating there is supposed to be a lifetime experience. He was very keen on winning this prize. Brad's design was completely different from Larry's design. Brad noticed that there are really three kinds of animals: Canines: animals with dog-like behaviour; Felines: animals with cat-like behaviour; and Others: animals with other behaviour. He decided to build this in to his class design.

Brad's `Animal` class was identical to Larry's. However, Brad created two more classes: `Canine` and `Feline`. Both classes *extend* the `Animal` class. The reason why Brad introduced these classes is really neat. His reasoning was as follows. *All* `Canines` eat meat. In addition, *all* `Canines` roam in packs. All this is common dog-like behaviour and should be *shared* for all `Canine` animals: `Dog` and `Wolf` objects. The following shows Brad's implementation of the `Canine` class. Being a seasoned `Java` programmer Brad didn't forget the '`@Override`' annotation before the method `roam`, which is overridden by the `Canine` class.

```java
public class Canine extends Animal {
    public Canine( )    { eatsGrass = false;}

    @Override
    public void roam( ) { System.out.println( "Roaming in my pack." ); }
}
```

Brad's `Feline` class was similar. *All* `Felines` eat meat. In addition, *all* `Felines` roam alone. This is common cat-like behaviour which should be shared by all `Felines`: `Cats`, `Tigers`, and `Lions`. The following is Brad's `Feline` class.

```java
public class Feline extends Animal {
    public Feline( )    { eatsGrass = false;}

    @Override
    public void roam( ) { System.out.println( "Roaming alone." );}
}
```

Brad's design is really clever. His design factors out all common `Canine` behaviour. As a result his `Dog` and `Wolf` classes have become very simple.

- All `Canines` inherit the roaming behaviour, which is now only implemented once. This is a much better solution than that of Larry, who implemented the same roaming method for his `Dog` and `Wolf` class *by duplicating code*.
- By default, `eatsGrass` is `false` for all `Canines`. As a consequence there is no need to initialise `eatsGrass` in the `Dog` and `Wolf` classes. Compared to Larry, who initialised `eatsGrass` for his `Dog` class and his `Wolf` class by duplicating code, Brad has come up with a better solution.

The following is Brad's `Dog` class. It's really simple.

```java
public class Dog extends Canine {
    private static final int DOG_HUNGER = 4;
    private static final String DOG_PICTURE = "dog.jpg";
    public Dog( ) {
        picture = DOG_PICTURE;
        // eatsGrass is false by default.
        hunger = DOG_HUNGER;
    }
    // Inherits eating  behaviour from Aninmal class.
    // Inherits roaming behaviour from Canine  class.
    @Override
    public void makeNoise( ) { System.out.println( "Arf. Arf." ); }
}
```

The `Dog` class is a *subclass* of the `Canine` class: a `Dog` *is-a* `Canine`. Therefore, the `Dog` class *extends* the `Canine` class. This lets the `Dog` class inherit all default behaviour from the `Canine` class. (Brad's `Wolf` class is similar and is not shown.)

Brad proceeded by implementing the `Cat`, `Tiger`, and `Lion` classes as subclasses of the `Feline` class. The following is what his `Cat` class looked like. Again lots of code is shared. Notice that this time the `Cat` class *extends* the `Feline` class. (Brad's `Tiger` and `Lion` classes are similar to his `Cat` class and are not shown.)

```java
public class Cat extends Feline {
    private static final int CAT_HUNGER = 1;
    private static final String CAT_PICTURE = "cat.jpg";
    public Cat( ) {
        picture = CAT_PICTURE;
        // eatsGrass is false by default.
        hunger = CAT_HUNGER;
    }
    // Inherits eating  behaviour from Aninmal class.
    // Inherits roaming behaviour from Feline  class.
    @Override
    public void makeNoise( ) { System.out.println( "Mew. Mew." ); }
}
```

Most of Brad's work goes into writing the `Hippo` class. The main reason is that less code can be shared. The following is Brad's `Hippo` class. Notice that it *extends* the `Animal` class.

```java
public class Hippo extends Animal {
    // ⟨constants omitted⟩
    public Hippo( ) {
        picture = HIPPO_PICTURE;
        eatsGrass = true;
        hunger = HIPPO_HUNGER;
    }
    // Inherits eating  behaviour from Aninmal class.
    @Override
    public void roam( ) { System.out.println( "I'm lazy: not roaming." ); }
    @Override
    public void makeNoise( ) { System.out.println( "Grunt." ); }
}
```

Figure 4 depicts Brad's class design in the form of a class diagram. This time — this is the norm — the inherited attributes and methods are not listed (by writing them down and commenting them out). Note that Brad's design is much cleaner compared to that of Larry's. The only classes that override the method `roam( )` are `Feline`, `Canine`, and `Hippo`. The subclasses of `Feline` and `Canine` inherit the method `roam( )` from their (immediate) superclass. These subclasses only have to override the method `makeNose( )`. All other methods and attributes are inherited from the `Animal` class. The `Hippo` class also overrides the method `roam`. Compared to Larry's design there are fewer method overrides.

### 4.4 Collecting the Prize

When Brad showed his application to his boss, his boss was very impressed and sent him off to Lennoxes to collect his prize.

## 5 For Wednesday

Don't forget to prepare for the test. For Wednesday: study the notes and Chapter 7, Pages 166–179.

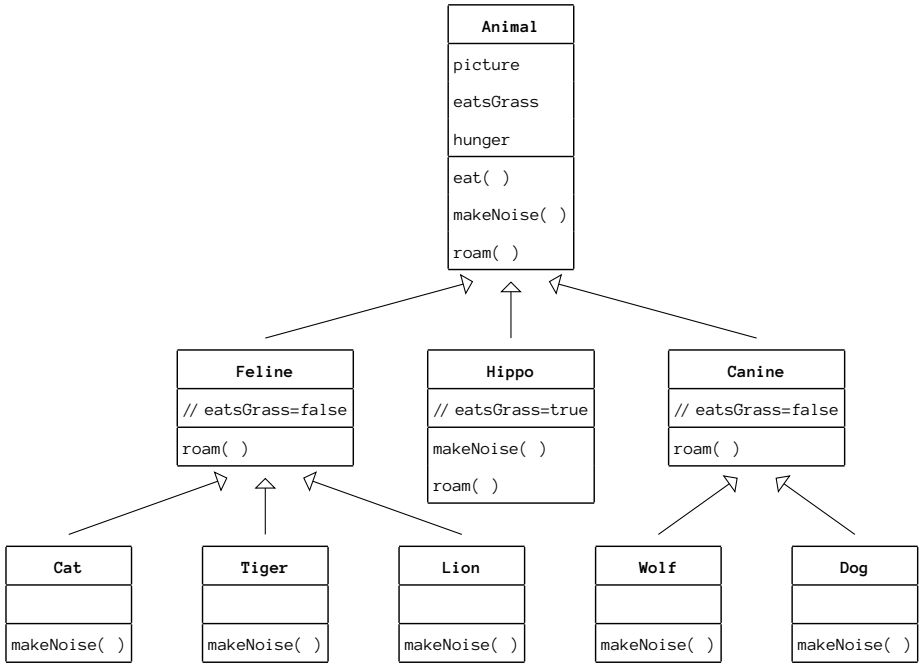


Figure 4: Brad's class design.