Introduction to Java (cs2514)

Lecture 7: Inheritance

M. R. C. van Dongen

February 6, 2017

Outline

- Revisit class and instance attributes.
- Revisit class and instance methods.
- Study *inheritance*.
  - With inheritance you can 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.

Relation with Owner

- `Java` has *class* and *instance* methods.
  - For a class method, you put `static` in the declaration.
- It also has *class* and *instance* variables.
  - For a class attribute, you put `static` in the declaration.
- *Instance methods & instance variables* are owned by instances.
  - There is one method/variable per instance of the class (one-to-one).
  - To access the method/variable you need the instance.
  - Instance attributes are for representing *object* state.
  - Instance methods are for *object* behaviour.
- *Class methods & class variables* are owned by the class.
  - There is one method/variable per class (one-to-one).
  - To access the method/variable you need the class.
  - Class attributes are for representing *class* state.
  - Class methods are for "class" behaviour.
- The class-to-instance (attribute/method) relation is one-to-many.

Encapsulation

- Consider an encapsulated (`private`) instance attribute, `attr`.
  - The attribute is only visible inside the class.
- Consider an instance, `instance`, of the defining class, *C*.
- Statements can only access/reference/see `instance.attr` if
  - They are defined inside *C*; and
  - They have reference to `instance`.
- There are two ways statements can reference to `instance`:

  Direct They can access the object reference `instance`; or

  Indirect They're in an instance method of *C* that was called using the object referrence `instance`.

Using the Object Reference

Direct Access

Java

```
public class Example {
    private int attribute;

    public Example( int initialValue ) {
        attribute = initialValue;
    }

    public static void main( ) {
        final Example good = new Example( 42 );
        final Example bad = new Example( 666 );

        bad.method( good );
        method( good, bad );
    }

    // artifical example only: this should never have been an instance method
    private void method( final Example object ) {
        object.attribute = 666;
    }

    private static void method( final Example first, final Example second ) {
        System.out.println( (first.attribute + second.attribute) );
    }
```

Using the Instance Method

Indirect Access

Java

```
public class Example {
    private int attribute;

    public Example( int initialValue ) {
        attribute = initialValue;
    }

    public static void main( ) {
        final Example example = new Example( 42 );
        example.method( ); // direct access: you need the dot notation
    }

    private void method( ) {
        // indirect access inside the instance method: no need for dot notation
        System.out.println( attribute ); // indirect access
    }
}
```

Using the Instance Method

Falling Back on Dot Notation

Java

```
public class Example {
    private int attribute;

    public Example( int initialValue ) {
        attribute = initialValue;
    }

    public static void main( ) {
        final Example example = new Example( 42 );
        example.method( ); // direct access: you need the dot notation
    }

    private void method( ) {
        // in instance method you can always fall back on this
        System.out.println( this.attribute ); // direct explict access
    }
}
```

Notation for Class

- The notation for class methods depends on where "you" are.
- You may always write '⟨`class`⟩`.`⟨`method`⟩`(` ⟨`arguments`⟩ `)`.'
- In the defining class you may write '⟨`method`⟩`(` ⟨`arguments`⟩ `)`.'
- Same for variables: you may always write '⟨`class`⟩`.`⟨`variable`⟩.'
- Inside the defining class you may also write '⟨`variable`⟩.'

Example

Java

```
public class Inside {
    public static int attribute;

    public static void method( ) {
        int var1 = attribute;
        int var2 = Inside.attribute;
        System.out.println( var1 + " = " + var2 );
    }
}
```

Java

```
public class Outside {
    public static void method( ) {
        // System.out.println( attribute ); // Not allowed.
        System.out.println( Inside.attribute );
    }
}
```

Example

Java

```
public class Inside {
    public static int attribute;

    public static void method( ) {
        int var1 = attribute;
        int var2 = Inside.attribute;
        System.out.println( var1 + " = " + var2 );
    }
}
```

Java

```
public class Outside {
    public static void method( ) {
        // System.out.println( attribute ); // Not allowed.
        System.out.println( Inside.attribute );
    }
}
```

Notation for Instances

- The notation for instance variables and methods is similar.
- You may always use '⟨`reference`⟩`.`⟨`method`⟩`(` ⟨`arguments`⟩ `)`.'
- You may use '⟨`method`⟩`(` ⟨`arguments`⟩ `)`' in defining class.
  - (Provided you're in an instance method.)
- For attributes you may write '⟨`reference`⟩`.`⟨`variable`⟩.'
- But in the defining class you may also write '⟨`attribute`⟩.'
  - (Provided you're in an instance method.)

Notation for Instances: Contined

- The dotless notation is only allowed inside instance methods.
- Inside instance methods you use '`this`' for the "current" object.
- Using '⟨`instance variable`⟩' without dot-notation means
  - '`this.`⟨`instance variable`⟩`.`'
- For instance methods this is the same.
- So '⟨`instance method`⟩`(` ⟨arguments⟩ `)`' means
  - '`this.`⟨`instance method`⟩`(` ⟨arguments⟩ `).`'

Example

Java

```
public class Inside {
    private int attribute;

    private static void classMethod( int var ) {
        System.out.println( var );
    }

    public void instanceMethod1( ) {
        classMethod( attribute );
    }

    public void instanceMethod2( ) {
        classMethod( this.attribute );
    }
}
```

Java

```
public class Outside {
    public static void main( String args[] ) {
        Inside inside = new Inside( );
        inside.instanceMethod1( );
        inside.instanceMethod2( );
    }
}
```

Simulating Instance Methods

Java

```
public class Simulation {
    private int attribute;

    public static void classMethod( Simulation current ) {
        System.out.println( current.attribute );
    }

    public void instanceMethod( ) {
        classMethod( this );
    }
}
```

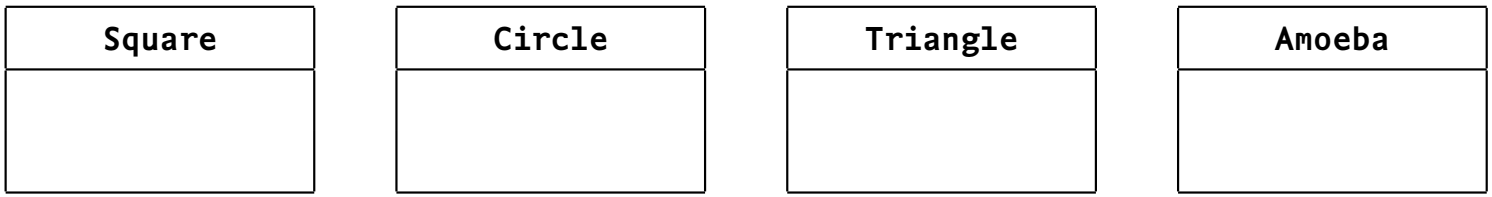
Java

```
public class Main {
    public static void main( String args[] ) {
        Simulation simulation = new Simulation( );
        // The following calls are effectively identical.
        simulation.instanceMethod( );
        Simulation.classMethod( simulation );
    }
}
```

Chair Wars Revisited

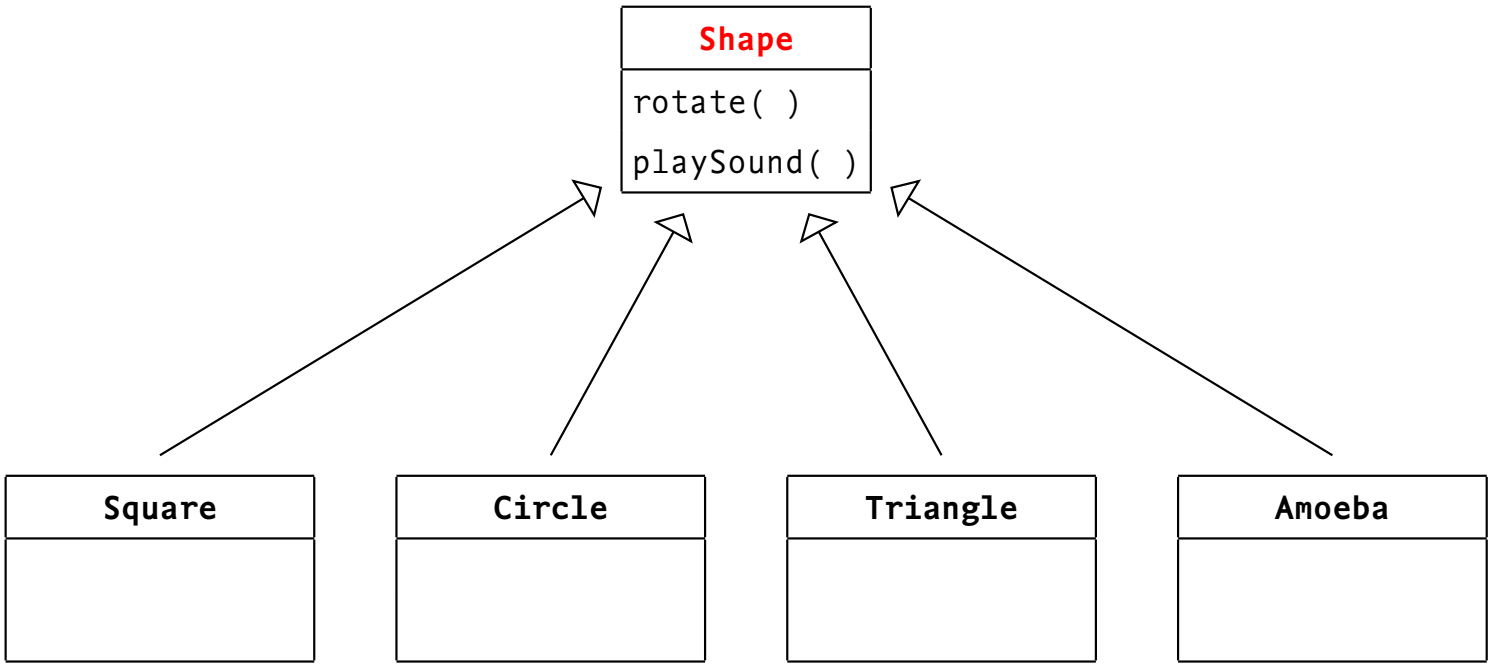
- Remember Larry and Brad?
- Brad's final solution had five classes:
  - One `Shape` *superclass* for default, common shape behaviour.
  - A dedicated class for each actual shape.
  - Each dedicated class was a *subclass* of the `Shape` class.
  - All, except for `Amoeba`, inherited all behaviour from `Shape`.
  - `Amoeba` *overrode* behaviour for `playSound`, and `rotate`.
  - This let `Amoeba` objects do things differently.
- Larry thought Brad's final class had lots of duplicated code:
  - "Your classes have same code for `playSound` and `rotate`."
  - "This makes it impossible to maintain your code."
  - "For each change, you need to edit 4 classes."
    - Editing n class files is n times more work than editing 1 file.
    - Each edit increases the probability of errors: more errors.
- But then Brad explained his design.

Brad's Final Design



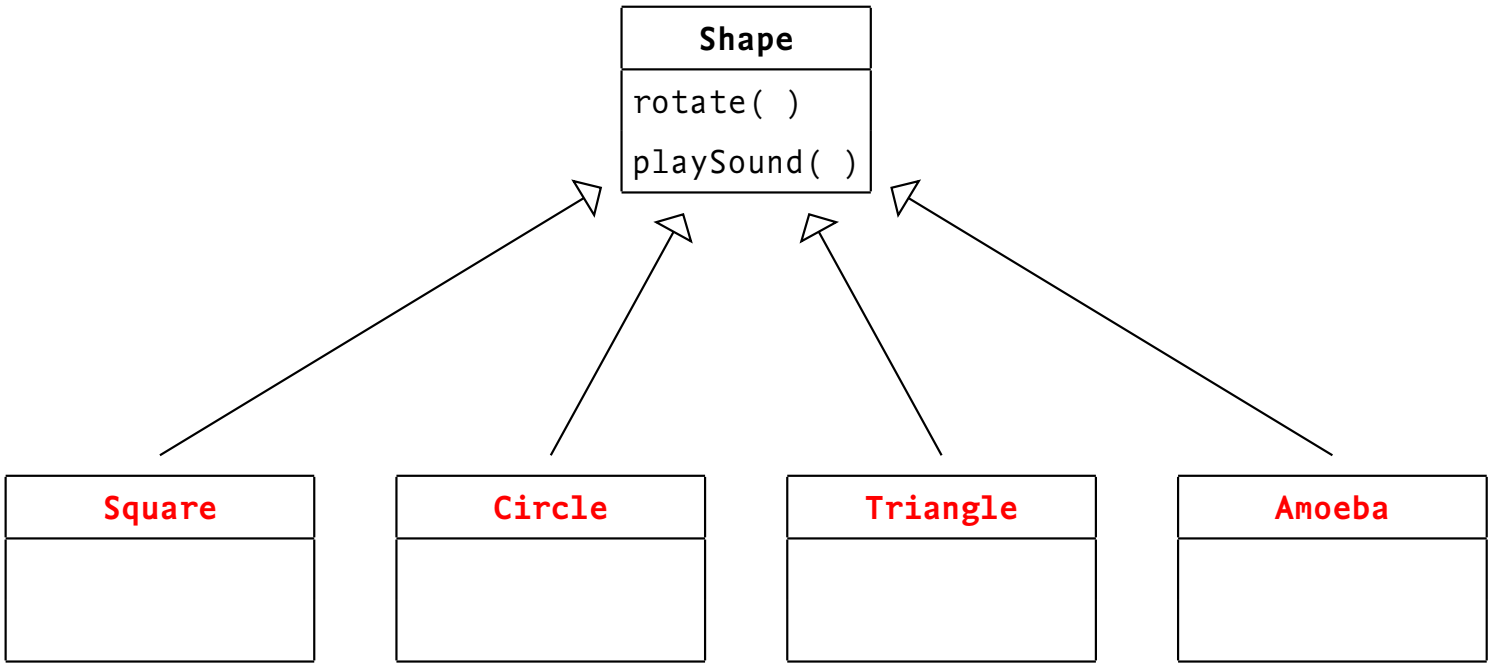
Brad's Final Design

Superclass



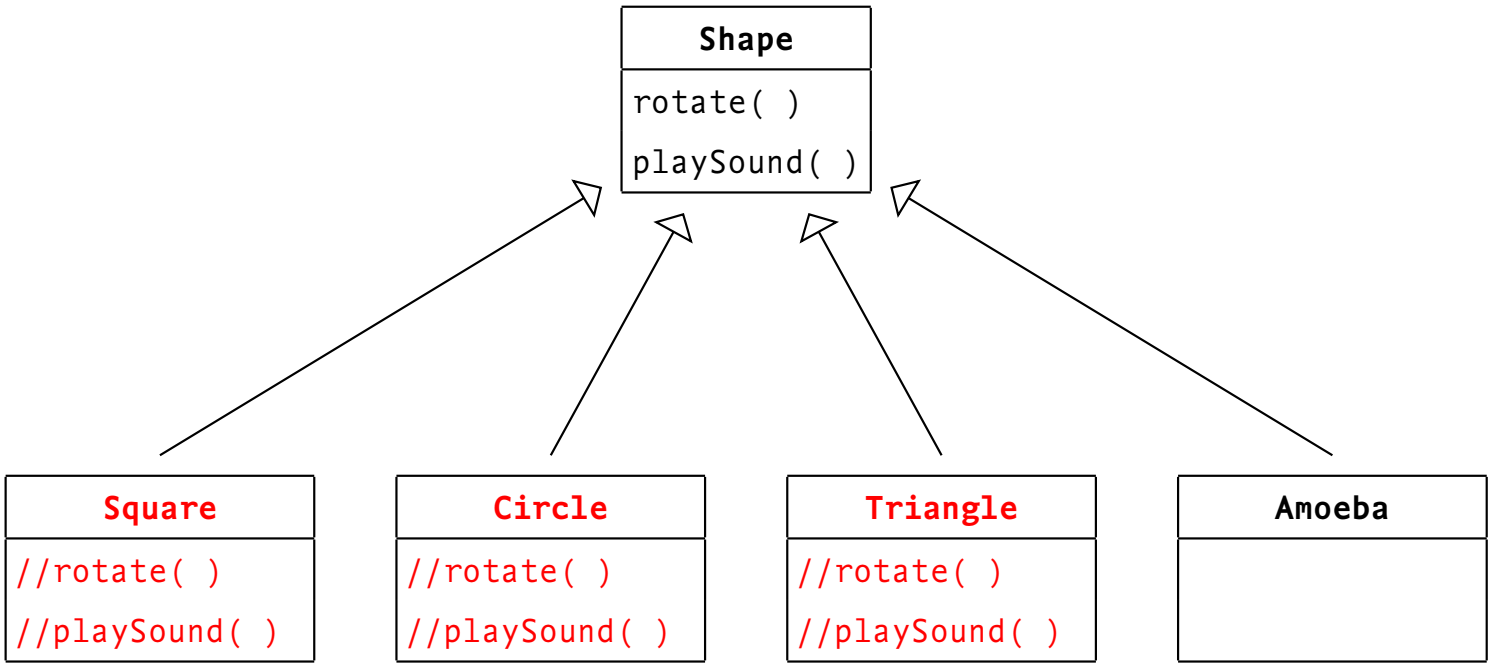
Brad's Final Design

Subclasses



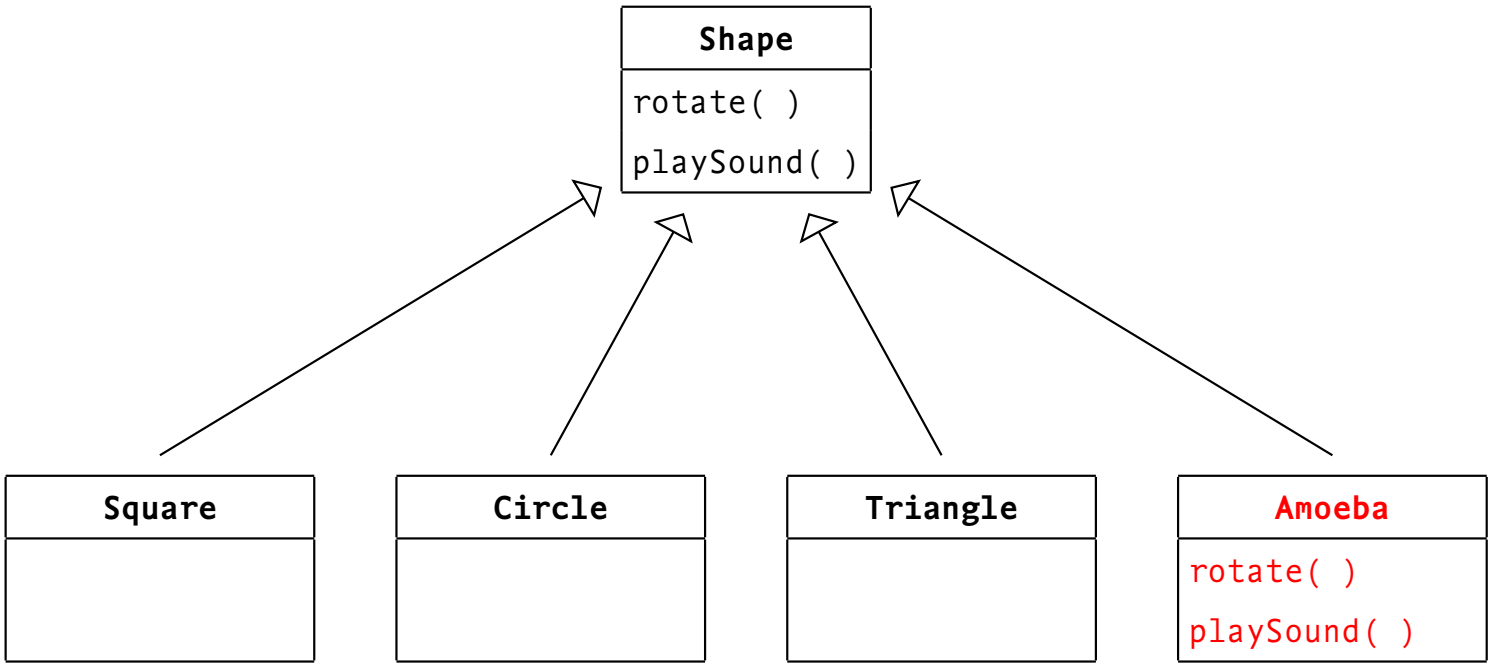
Brad's Final Design

Inherit



Brad's Final Design

Overrides



Inheritance

- There are two main advantages of *inheritance*:
  - Increases ability to reuse implementation effort.
  - Separates class-specific from general code.
- Code is structured in classes so as to maximise 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:*
  - Subclass provides same functionality as its superclass.
  - So if the superclass has a method then so does the subclass.
  - Here, the subclass *inherits* the method from its superclass.
  - However, the subclass functionality may be more *specific.*
    - E.g., the subclass may implement a method in a *different* way.
    - Here, the subclass *overrides* the method of its superclass.
  - Subclasses may also have more specific, *additional* behaviour.
- A subclass is said to *extend* its superclass.

Example

- 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* have a method called `treatPatient( )`.
- *Both* have a property `worksAtHospital` (a `boolean`).
  - *Any* `Doctor` has it.
  - For a `Surgeon` it is `true`.
  - For a `GP` it is `false`.
- `Surgeons` and `GPs` differ from `Doctors` *in general*:

  Surgeon:
  - Has additional `makeIncision( )` method.
  - Has *special* implementation for `treatPatient`.
    - *Overrides* default `treatPatient( )` implementation.

  GP:
  - Has additional attribute `makesHouseCalls`.
  - Has additional method `giveAdvice( )`.

Example Continued

The Common, More General Code

- We put the *more general* code in the `Doctor` class.
- This is the code that *any* `Doctor` should have:

Java

```
public class Doctor {
    public boolean worksAtHospital;

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



}
```

Example Continued

The Common, More General Code: Did we Forget Anything?

- We put the *more general* code in the `Doctor` class.
- This is the code that *any* `Doctor` should have:

Java

```
public class Doctor {
    public boolean worksAtHospital;

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



}
```

Example Continued

The Common, More General Code: Think of Something Any Doctor Does

- We put the *more general* code in the `Doctor` class.
- This is the code that *any* `Doctor` should have:

Java

```
public class Doctor {
    public boolean worksAtHospital;

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



}
```

Example Continued

The Common, More General Code: Right!

- We put the *more general* code in the `Doctor` class.
- This is the code that *any* `Doctor` should have:

Java

```
public class Doctor {
    public boolean worksAtHospital;

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

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

Example Continued

The More Specific Code: The Surgeon Class

Java

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

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

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

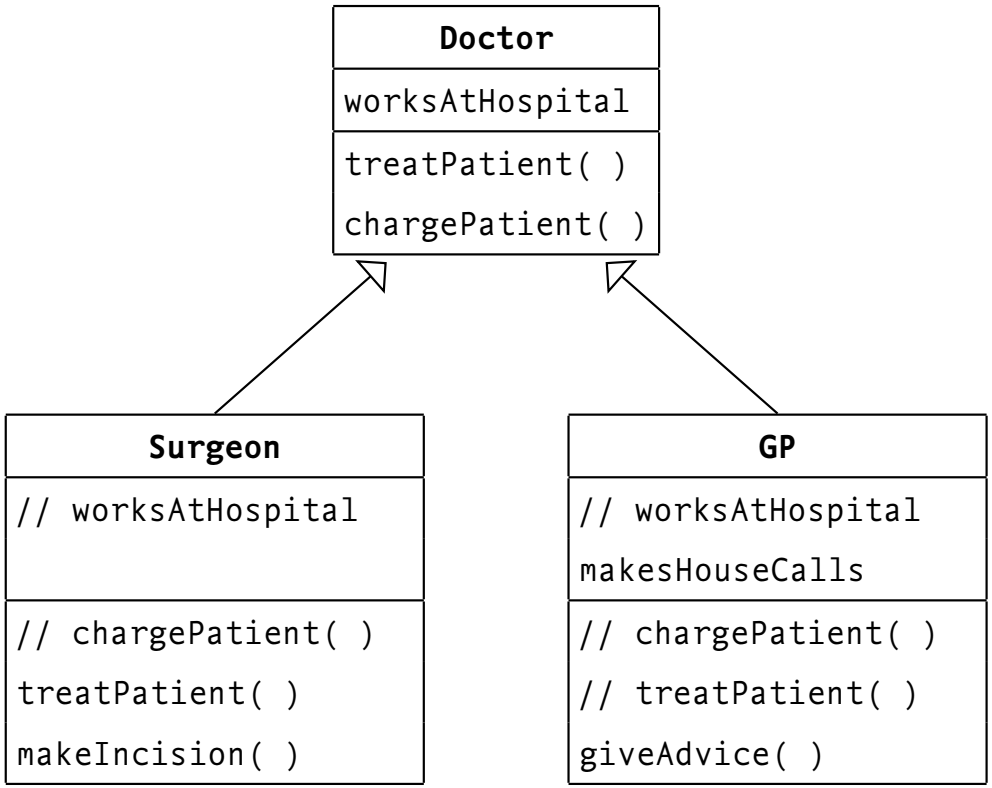
Example Continued

The More Specific Code: The GP Class

Java

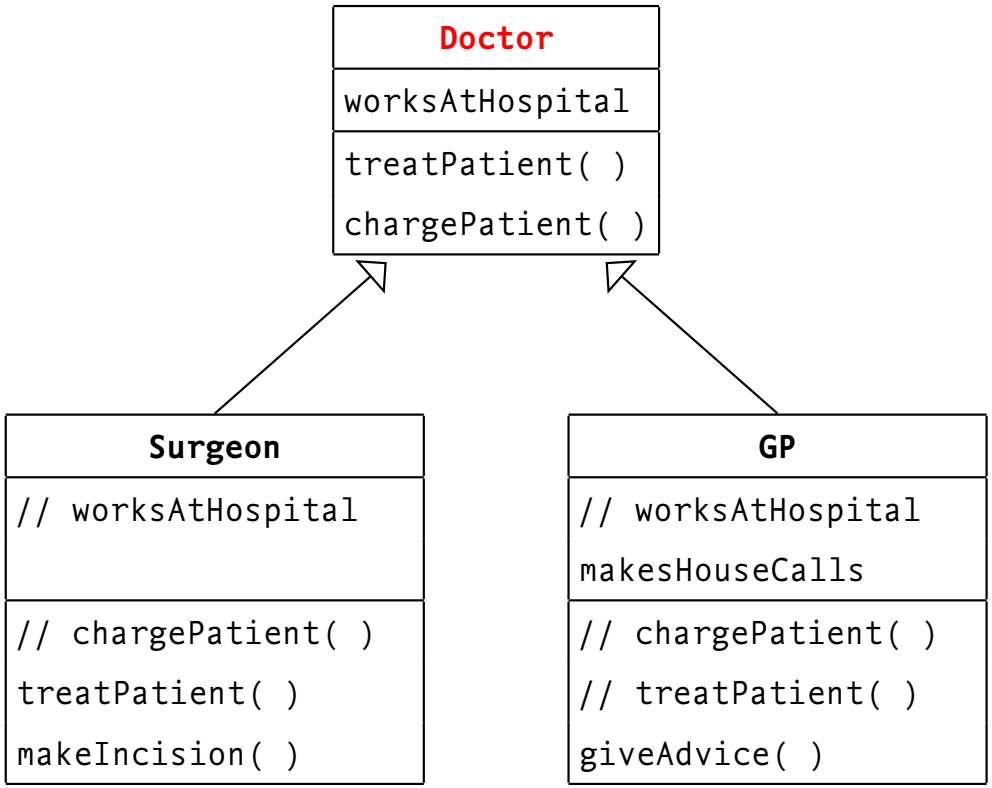
```
public class GP extends Doctor {
    public boolean makesHouseCalls;

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

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

The Class Diagram



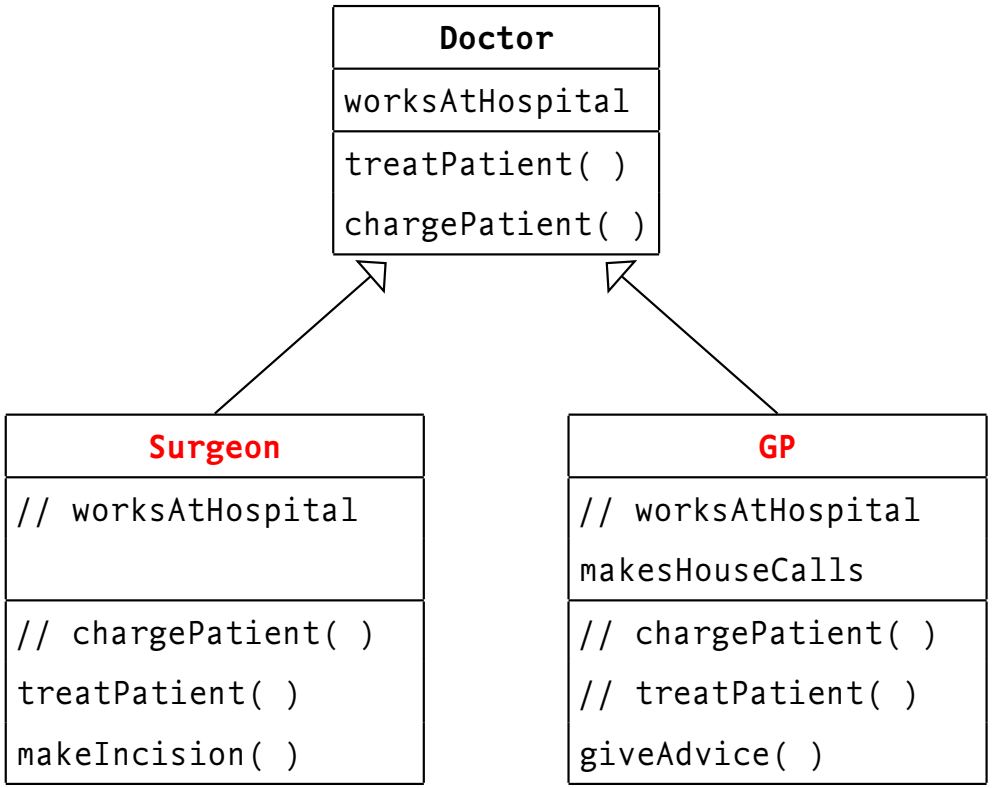
The Class Diagram

Superclass



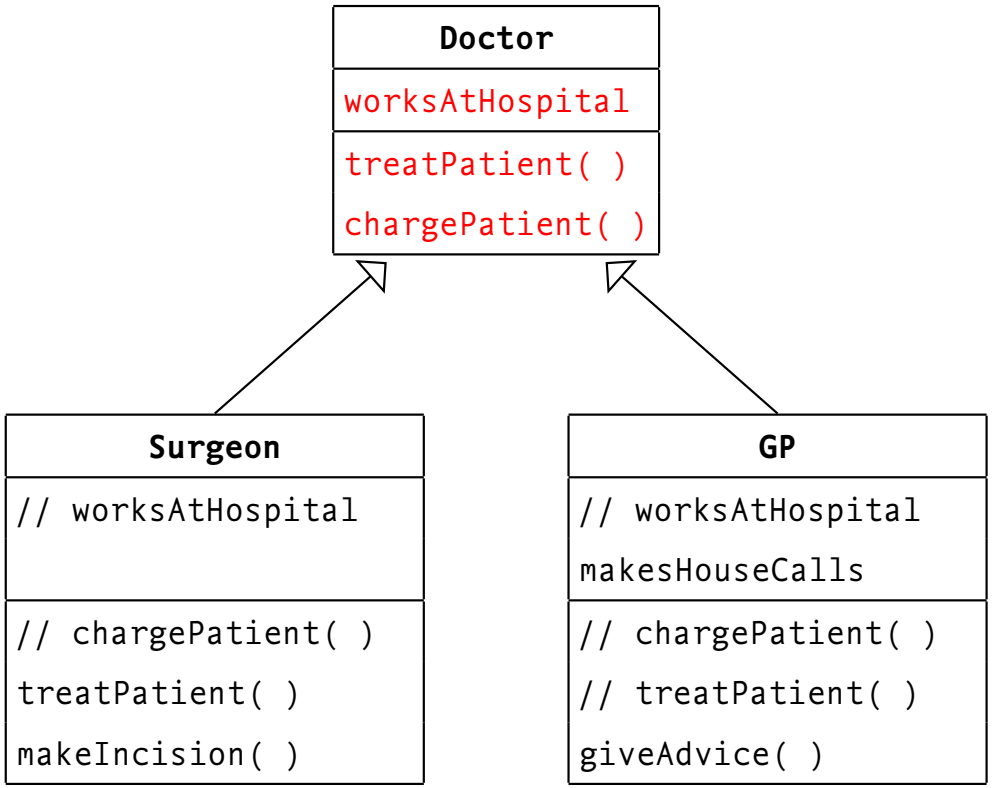
The Class Diagram

Subclasses



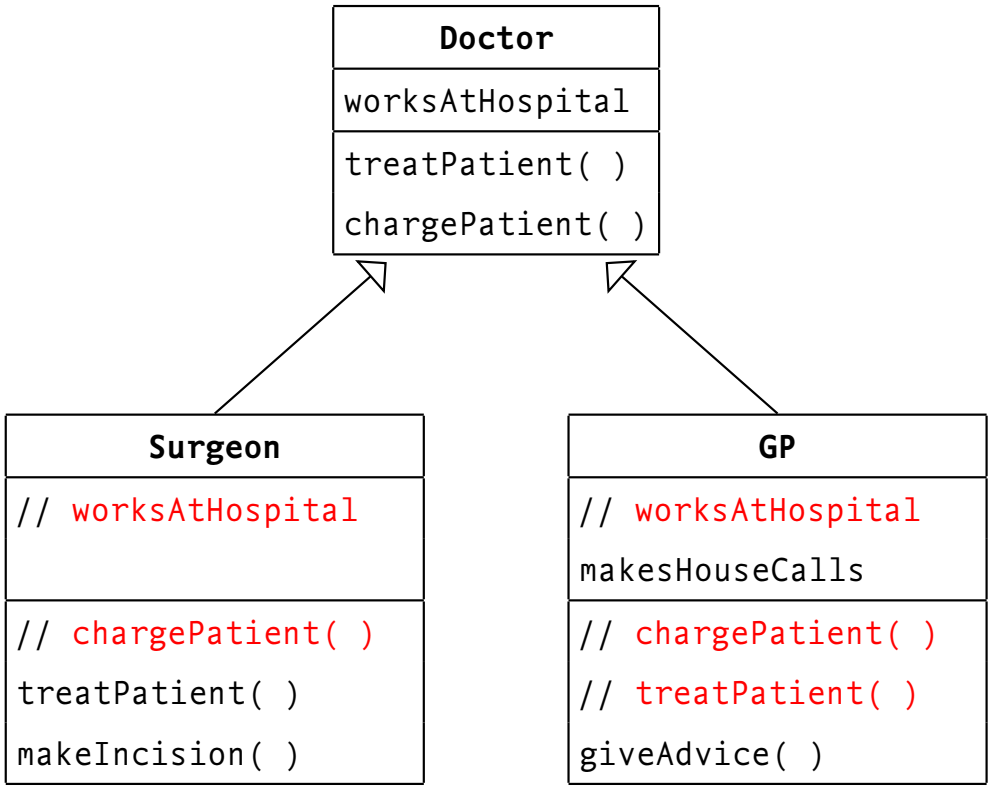
The Class Diagram

Common Methods and Attributes



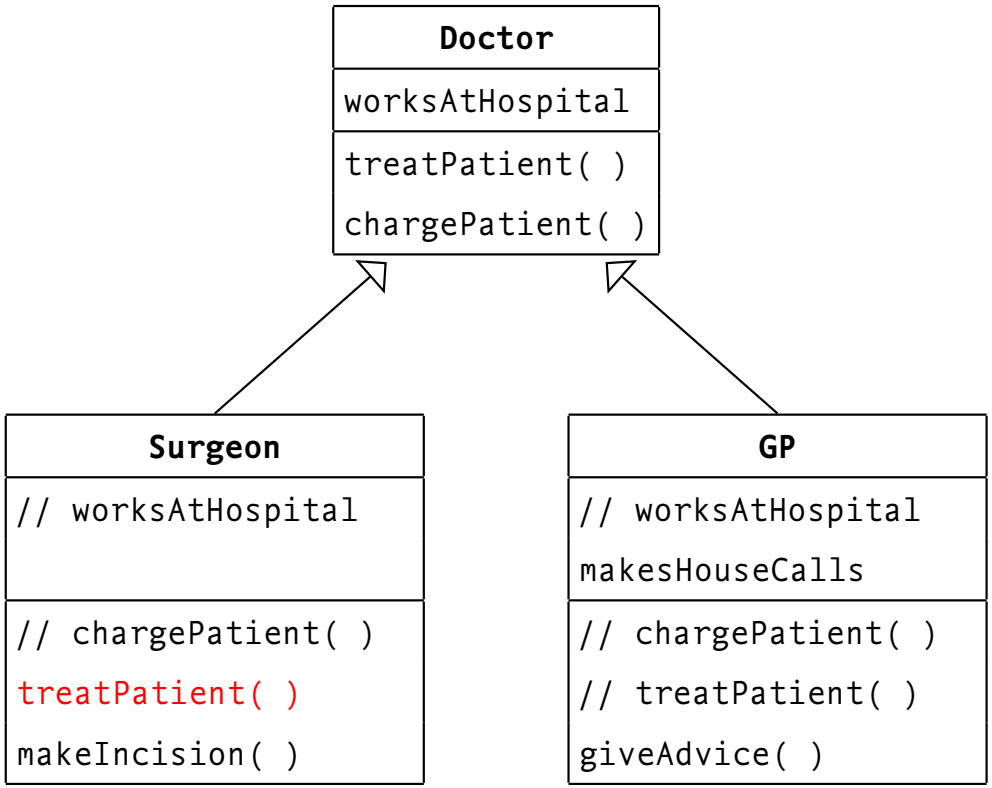
The Class Diagram

Inherit



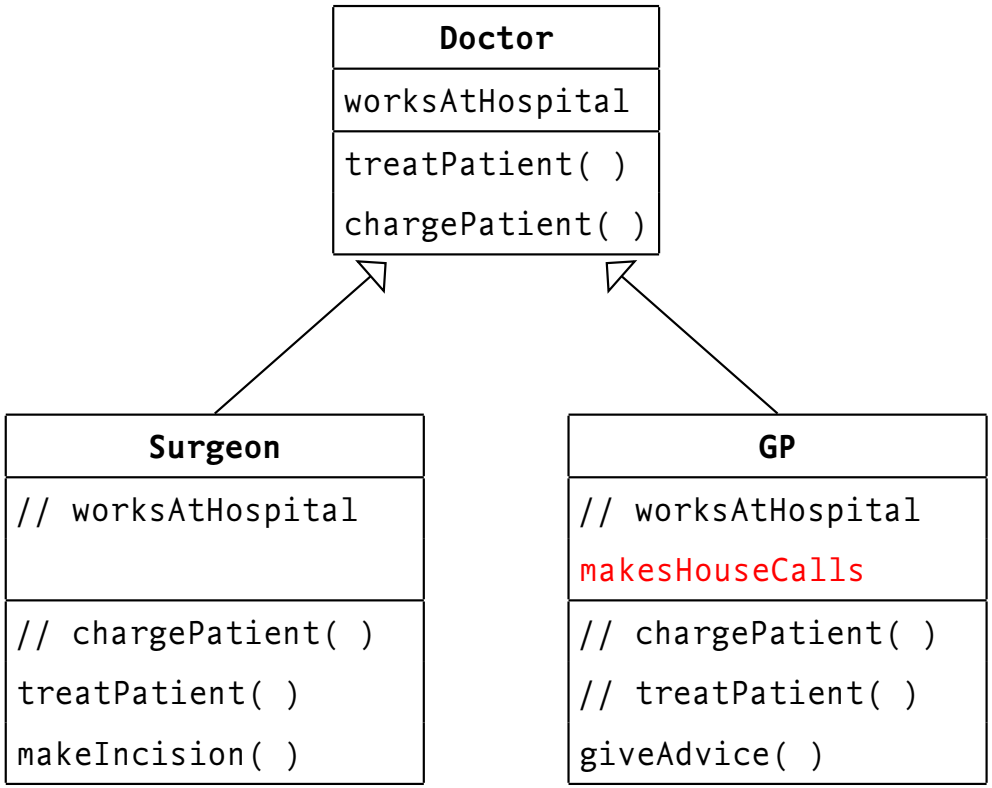
The Class Diagram

Overrides



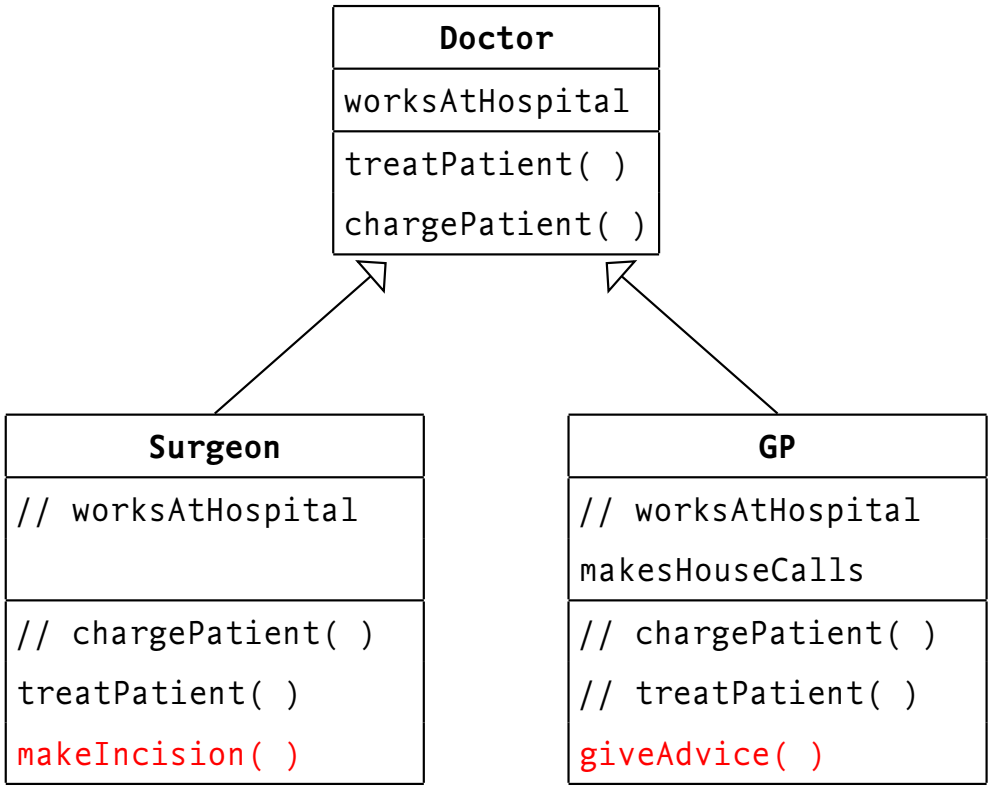
The Class Diagram

Specific Attribute



The Class Diagram

Specific Methods



The Fota Challenge

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.

Act I: The Challenge

Act I: The Challenge

Act I: The Challenge

Act I: The Challenge

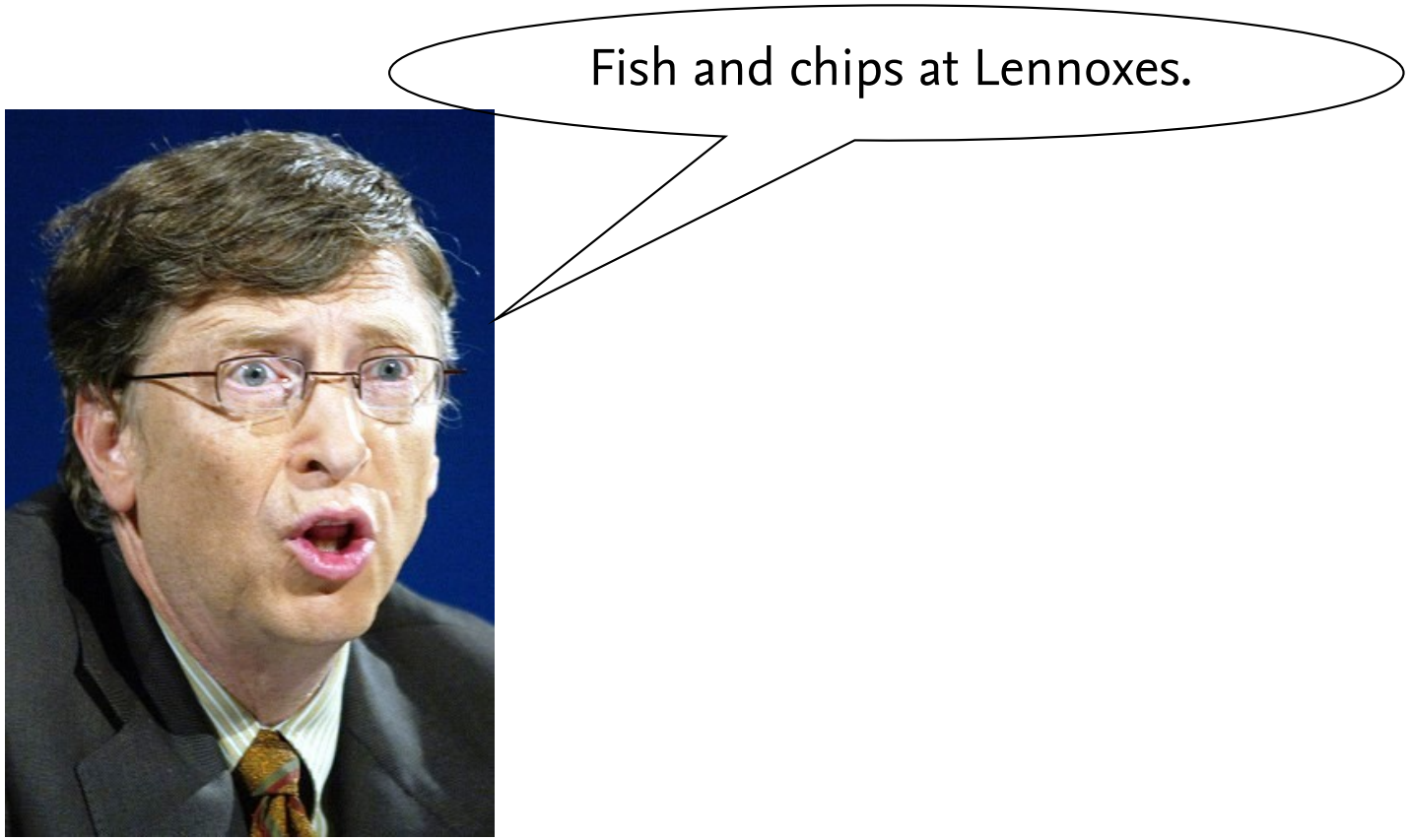
The Spec

- Fota Wildlife Park has lots of animals:
  - A lion;
  - A cat;
  - A wolf;
  - A tiger;
  - A dog; and
  - They're expecting a hippo.
- Each animal:
  - Has a picture `String`;
  - Has a certain kind of food: grass or meat;
  - Has an integer hunger level;
  - Eats;
  - Makes noise; and
  - Has a roaming behaviour.

Act I: The Challenge

Act I: The Challenge

Act I: The Challenge



Introducing the Contestants: Meet Larry

- Larry has been taking Java lessons with Amy.
- He has just started to learn about inheritance.
- He knows inheritance is the key to solving this problem.
- He just knows he will beat Brad.

Introducing the Contestants: Meet Brad

- Brad is just delighted with this application.
- This is a textbook example of inheritance.
- He knows this can't be too difficult.

Meanwhile in Larry's Cubicle

- Larry quickly carries out a noun analysis.
- Tells him the main actors are animals.
- He now knows the main class should be `Animal`.
- He puts all the common methods and attributes in this class.

Meanwhile in Larry's Cubicle

- Larry quickly carries out a noun analysis.
- Tells him the main actors are animals.
- He now knows the main class should be `Animal`.
- He puts all the common methods and attributes in this class.

Java

```
public class Animal {
    public final String  picture;
    public final boolean eatsGrass;
    public final int     hungerLevel;

    ⟨more⟩
}
```

Meanwhile in Larry's Cubicle

Java

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

public void eat( ) {          // Default eating behaviour.
    System.out.println( "Eating " + hungerLevel + " 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⟩
}
```

Meanwhile in Larry's Cubicle

Java

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

    public Hippo( ) {
        picture = HIPPO_PICTURE;
        eatsGrass = true;
        hungerLevel = HIPPO_HUNGER_LEVEL;
    }

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

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

Meanwhile in Larry's Cubicle

Java

```
import java.util.ArrayList;

public class Main {
    public static void main( final String[] args ) {
        final 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 Presents His Solution

Unix Session

```
$
```

Larry Presents His Solution

Unix Session

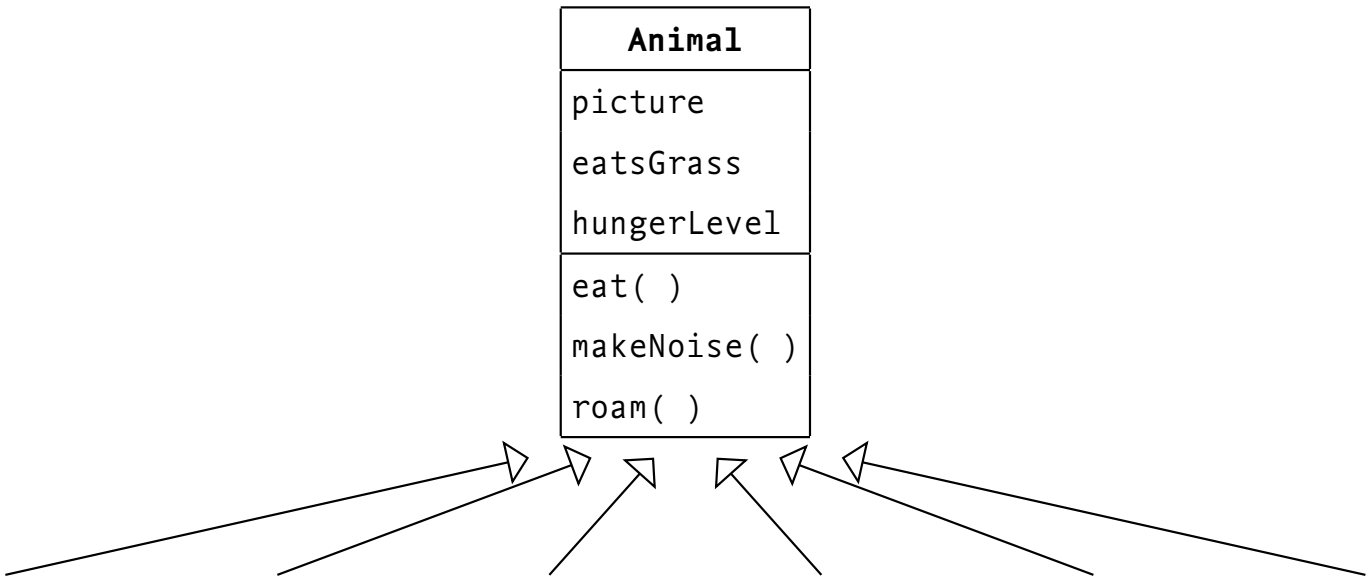
```
$ java Main
```

Larry Presents His Solution

Unix Session

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

Larry's Class Diagram



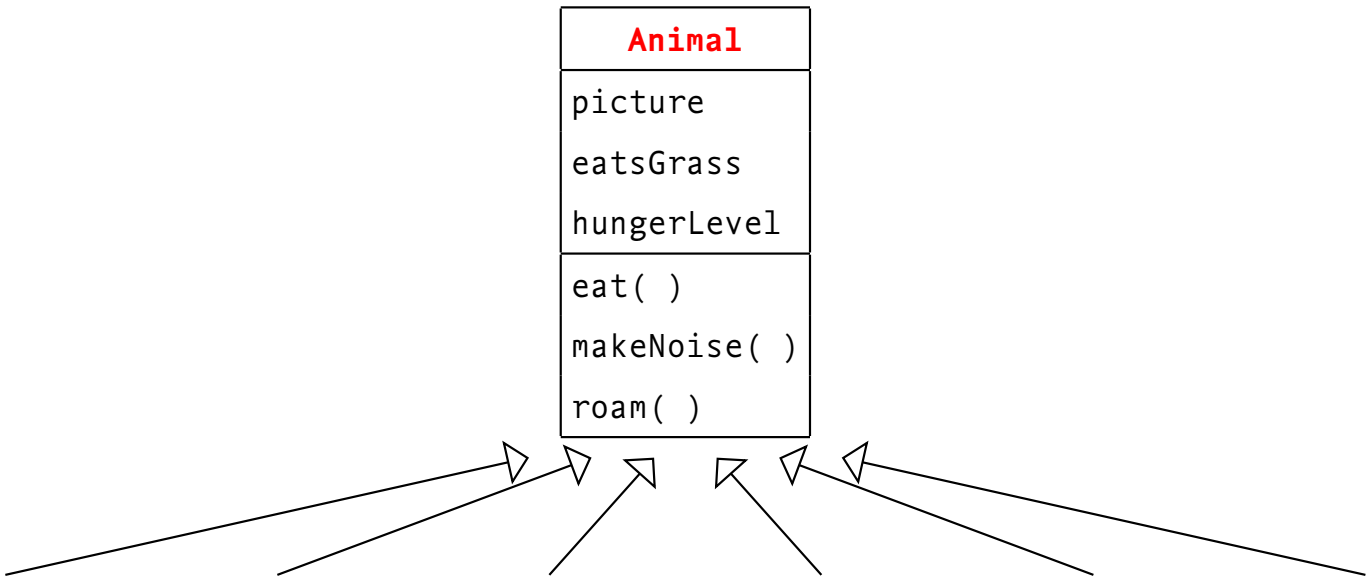
Cat	Tiger	Lion	Wolf	Dog	Hippo

// picture
// eatsGrass
// hungerLevel
	// picture
	// eatsGrass
	// hungerLevel
		// picture
		// eatsGrass
		// hungerLevel
			// picture
			// eatsGrass
			// hungerLevel
				// picture
				// eatsGrass
				// hungerLevel
					// picture
					// eatsGrass
					// hungerLevel

// eat( )
makeNoise( )
roam( )
	// eat( )
	makeNoise( )
	roam( )
		// eat( )
		makeNoise( )
		roam( )
			// eat( )
			makeNoise( )
			roam( )
				// eat( )
				makeNoise( )
				roam( )
					// eat( )
					makeNoise( )
					roam( )

Larry's Class Diagram

Superclass



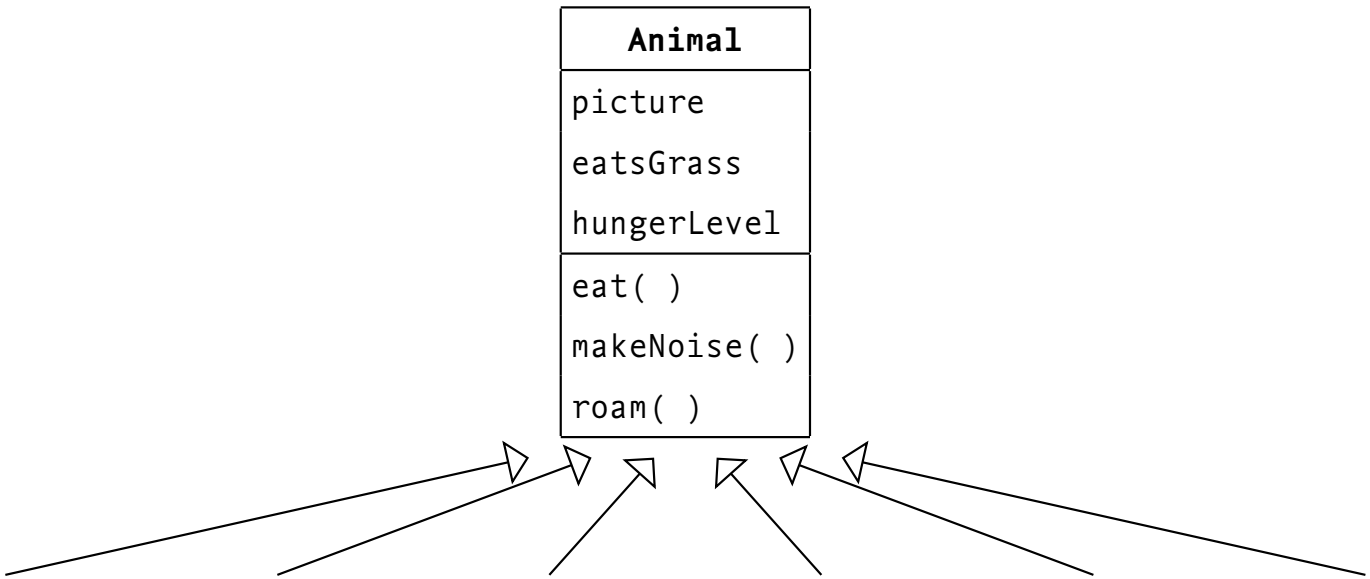
Cat	Tiger	Lion	Wolf	Dog	Hippo

// picture
// eatsGrass
// hungerLevel
	// picture
	// eatsGrass
	// hungerLevel
		// picture
		// eatsGrass
		// hungerLevel
			// picture
			// eatsGrass
			// hungerLevel
				// picture
				// eatsGrass
				// hungerLevel
					// picture
					// eatsGrass
					// hungerLevel

// eat( )
makeNoise( )
roam( )
	// eat( )
	makeNoise( )
	roam( )
		// eat( )
		makeNoise( )
		roam( )
			// eat( )
			makeNoise( )
			roam( )
				// eat( )
				makeNoise( )
				roam( )
					// eat( )
					makeNoise( )
					roam( )

Larry's Class Diagram

Subclasses



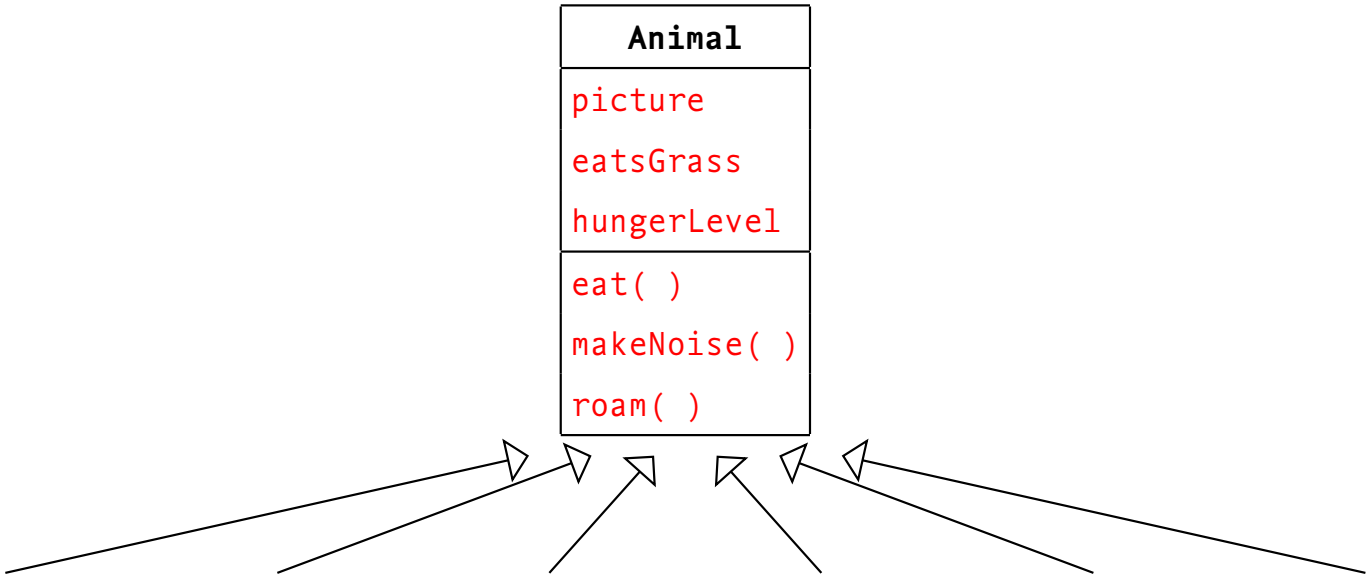
Cat	Tiger	Lion	Wolf	Dog	Hippo

// picture
// eatsGrass
// hungerLevel
	// picture
	// eatsGrass
	// hungerLevel
		// picture
		// eatsGrass
		// hungerLevel
			// picture
			// eatsGrass
			// hungerLevel
				// picture
				// eatsGrass
				// hungerLevel
					// picture
					// eatsGrass
					// hungerLevel

// eat( )
makeNoise( )
roam( )
	// eat( )
	makeNoise( )
	roam( )
		// eat( )
		makeNoise( )
		roam( )
			// eat( )
			makeNoise( )
			roam( )
				// eat( )
				makeNoise( )
				roam( )
					// eat( )
					makeNoise( )
					roam( )

Larry's Class Diagram

Common Methods and Attributes



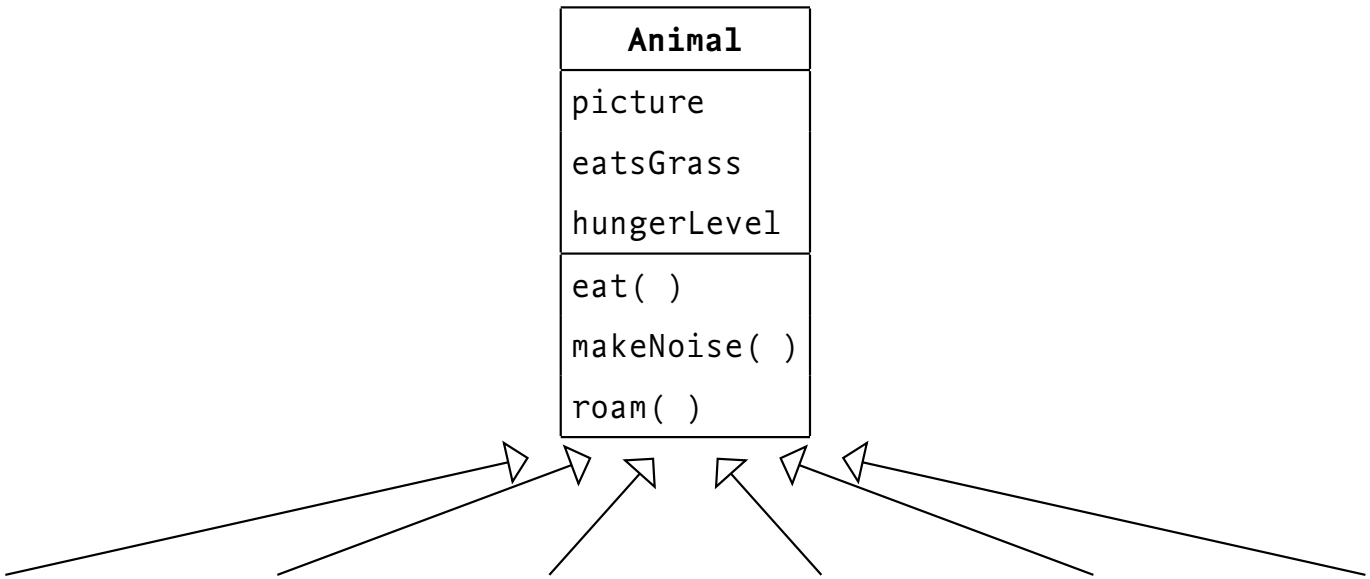
Cat	Tiger	Lion	Wolf	Dog	Hippo

// picture
// eatsGrass
// hungerLevel
	// picture
	// eatsGrass
	// hungerLevel
		// picture
		// eatsGrass
		// hungerLevel
			// picture
			// eatsGrass
			// hungerLevel
				// picture
				// eatsGrass
				// hungerLevel
					// picture
					// eatsGrass
					// hungerLevel

// eat( )
makeNoise( )
roam( )
	// eat( )
	makeNoise( )
	roam( )
		// eat( )
		makeNoise( )
		roam( )
			// eat( )
			makeNoise( )
			roam( )
				// eat( )
				makeNoise( )
				roam( )
					// eat( )
					makeNoise( )
					roam( )

Larry's Class Diagram

Inherit



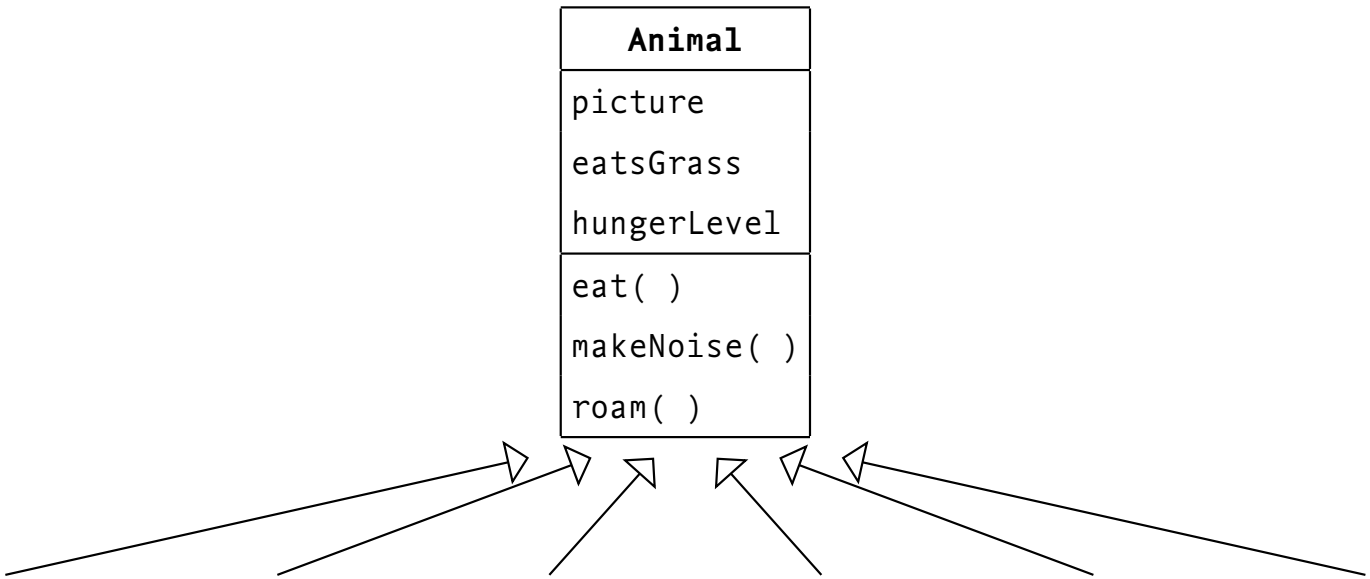
Cat	Tiger	Lion	Wolf	Dog	Hippo

// picture
// eatsGrass
// hungerLevel
	// picture
	// eatsGrass
	// hungerLevel
		// picture
		// eatsGrass
		// hungerLevel
			// picture
			// eatsGrass
			// hungerLevel
				// picture
				// eatsGrass
				// hungerLevel
					// picture
					// eatsGrass
					// hungerLevel

// eat( )
makeNoise( )
roam( )
	// eat( )
	makeNoise( )
	roam( )
		// eat( )
		makeNoise( )
		roam( )
			// eat( )
			makeNoise( )
			roam( )
				// eat( )
				makeNoise( )
				roam( )
					// eat( )
					makeNoise( )
					roam( )

Larry's Class Diagram

Override



Cat	Tiger	Lion	Wolf	Dog	Hippo

// picture
// eatsGrass
// hungerLevel
	// picture
	// eatsGrass
	// hungerLevel
		// picture
		// eatsGrass
		// hungerLevel
			// picture
			// eatsGrass
			// hungerLevel
				// picture
				// eatsGrass
				// hungerLevel
					// picture
					// eatsGrass
					// hungerLevel

// eat( )
makeNoise( )
roam( )
	// eat( )
	makeNoise( )
	roam( )
		// eat( )
		makeNoise( )
		roam( )
			// eat( )
			makeNoise( )
			roam( )
				// eat( )
				makeNoise( )
				roam( )
					// eat( )
					makeNoise( )
					roam( )

Did Larry Win?

Did Larry Win?

Did Larry Win?

Unix Session

```
next: Animal[ picture = hippo.jpg, eatsGrass = grass, hungerLevel = 10 ]
I'm Lazy: not roaming.
Eating 10 portions of grass.
```

Larry Doesn't Get It

- Larry couldn't understand it.
- He had overridden the `Hippo`'s noise method.

Java

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

Larry Doesn't Get It

- Larry couldn't understand it.
- He had overridden the `Hippo`'s noise method.
- But Amy discovered the error.

Java

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

Larry Doesn't Get It

- Larry couldn't understand it.
- He had overridden the `Hippo`'s noise method.
- But Amy discovered the error.
- There was a typo in his `Hippo` class.

Java

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

Larry Doesn't Get It

- Larry couldn't understand it.
- He had overridden the `Hippo`'s noise method.
- But Amy discovered the error.
- There was a typo in his `Hippo` class.

Java

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

Larry Doesn't Get It

- Larry couldn't understand it.
- He had overridden the `Hippo`'s noise method.
- But Amy discovered the error.
- There was a typo in his `Hippo` class.

Java

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

Larry Doesn't Get It

- Larry couldn't understand it.
- He had overridden the `Hippo`'s noise method.
- But Amy discovered the error.
- There was a typo in his `Hippo` class.

Java

```
@Override // Makes sure we actually override an existing superclass method
public void makeNoise( ) {
    System.out.println( "Grunt." );
}
```

Meanwhile at Brad's Laptop

- Brad had read about Lennoxes in the *Lonely Planet*.
- Eating there is supposed to be a lifetime experience.
- He is very keen on winning this prize.
- Brad's design is completely different from Larry's.

Meanwhile at Brad's Laptop

- Brad had read about Lennoxes in the *Lonely Planet*.
- Eating there is supposed to be a lifetime experience.
- He is very keen on winning this prize.
- Brad's design is completely different from Larry's.
- He notices 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.

Meanwhile at Brad's Laptop

- Brad had read about Lennoxes in the *Lonely Planet*.
- Eating there is supposed to be a lifetime experience.
- He is very keen on winning this prize.
- Brad's design is completely different from Larry's.
- He notices 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 decides to build this into his class design.

Meanwhile at Brad's Laptop

- Brad creates two additional classes: `Canine` and `Feline`.
- Both *extend* the `Animal` class.

Java

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

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

Meanwhile at Brad's Laptop

- Brad creates two additional classes: `Canine` and `Feline`.
- Both *extend* the `Animal` class.
- All `Canines` eat meat.

Java

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

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

Meanwhile at Brad's Laptop

- Brad creates two additional classes: `Canine` and `Feline`.
- Both *extend* the `Animal` class.
- All `Canines` eat meat.
- All `Canines` roam in packs.

Java

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

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

Meanwhile at Brad's Laptop

- Brad creates two additional classes: `Canine` and `Feline`.
- Both *extend* the `Animal` class.
- All `Canines` eat meat.
- All `Canines` roam in packs.

Java

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

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

Meanwhile at Brad's Laptop

- Brad creates two additional classes: `Canine` and `Feline`.
- Both *extend* the `Animal` class.

Java

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

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

Meanwhile at Brad's Laptop

- Brad creates two additional classes: `Canine` and `Feline`.
- Both *extend* the `Animal` class.
- All `Felines` eat meat.

Java

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

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

Meanwhile at Brad's Laptop

- Brad creates two additional classes: `Canine` and `Feline`.
- Both *extend* the `Animal` class.
- All `Felines` eat meat.
- All `Felines` roam alone.

Java

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

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

Meanwhile at Brad's Laptop

- Brad's design is really clever.
- His design *factors out all common* `Canine` *behaviour.*
- This simplifies the `Canine` subclasses.
  - All `Canines` inherit the roaming behaviour.
  - By default, `eatsGrass` is `false` for all `Canines`.

Meanwhile at Brad's Laptop

Java

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

Meanwhile at Brad's Laptop

Java

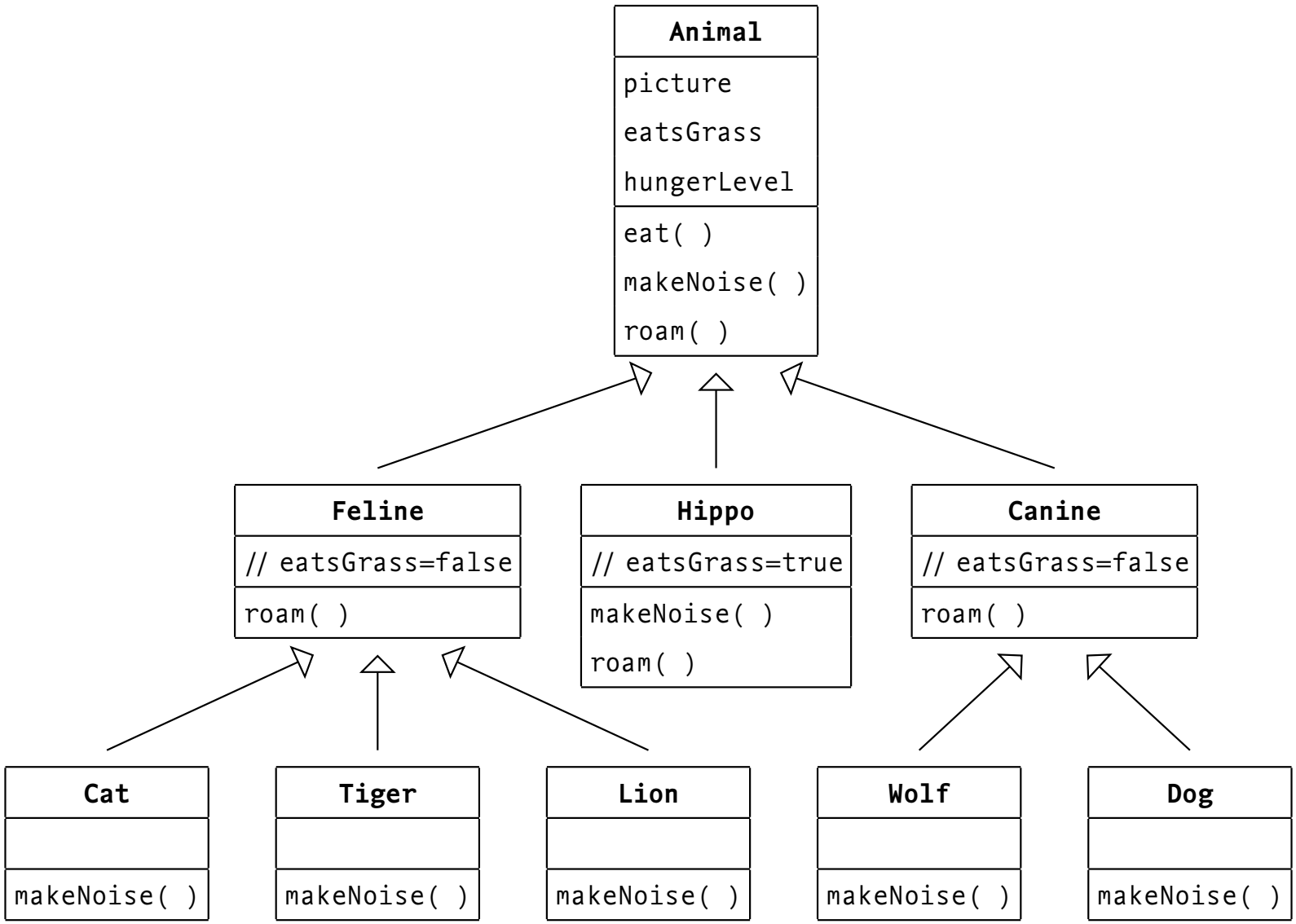
```
public class Cat extends Feline {
    private static final int CAT_HUNGER_LEVEL = 1;
    private static final String CAT_PICTURE = "cat.jpg";
    public Cat( ) {
        picture = CAT_PICTURE;
        // eatsGrass is false by default.
        hungerLevel = CAT_HUNGER_LEVEL;
    }
    // Inherits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

Meanwhile at Brad's Laptop

Java

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

Brad's Class Diagram



Collecting the Prize

Collecting the Prize

Collecting the Prize

Collecting the Prize: Cats love Fish

How Brad Has Implemented the Canine Class

Poor Style

Java

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

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

How Brad Has Implemented the Canine Class

Poor Style

Java

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

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

How Brad Has Implemented the Canine Class

Superclass Implementation Violates Encapsulation

Java

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

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

How Brad Has Implemented the Canine Class

Superclass Attributes are Mutable and Cannot be Private

Java

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

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

Calling the Superclass Constructor

Should be First Call in Constructor

Java

```
public class Canine extends Animal {
    private static final boolean EATS_GRASS = false;

    public Canine( final String picture, final int hungerLevel ) {
        super( picture, EATS_GRASS, hungerLevel );
    }

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

Calling the Superclass Constructor

Superclass Implementation Respects Encapsulation

Java

```
public class Canine extends Animal {
    private static final boolean EATS_GRASS = false;

    public Canine( final String picture, final int hungerLevel ) {
        super( picture, EATS_GRASS, hungerLevel );
    }

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

Calling the Superclass Constructor

Superclass Attributes are *Private* and *Immutable*

Java

```
public class Canine extends Animal {
    private static final boolean EATS_GRASS = false;

    public Canine( final String picture, final int hungerLevel ) {
        super( picture, EATS_GRASS, hungerLevel );
    }

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

Revised Animal Class

All Attributes are private

Java

```
public class Animal {
    private final String  picture;
    private final boolean eatsGrass;
    private final int     hungerLevel;

    public Animal( final String picture,
                   final boolean eatsGrass,
                   final int hungerLevel ) {
        this.picture     = picture;
        this.eatsGrass   = eatsGrass;
        this.hungerLevel = hungerLevel;
    }

    // ...
}
```

Question Time

Questions Anybody?

For Friday

- Study the presentation.
- Study Chapter 7 from the book.

Acknowledgements

- This lecture is partially based on
  - [Sierra, and Bates 2004].

Bibliography I


Sierra, Kathy, and Bert Bates [2004]. *Head First Java*. O'Reilly. ISBN: 978-0-596-00712-6.

About this Document

- This document was created with `pdflatex`.
- The LaTeX document class is `beamer`.