Introduction to Java (cs2514)

Lecture 9: Abstract Classes

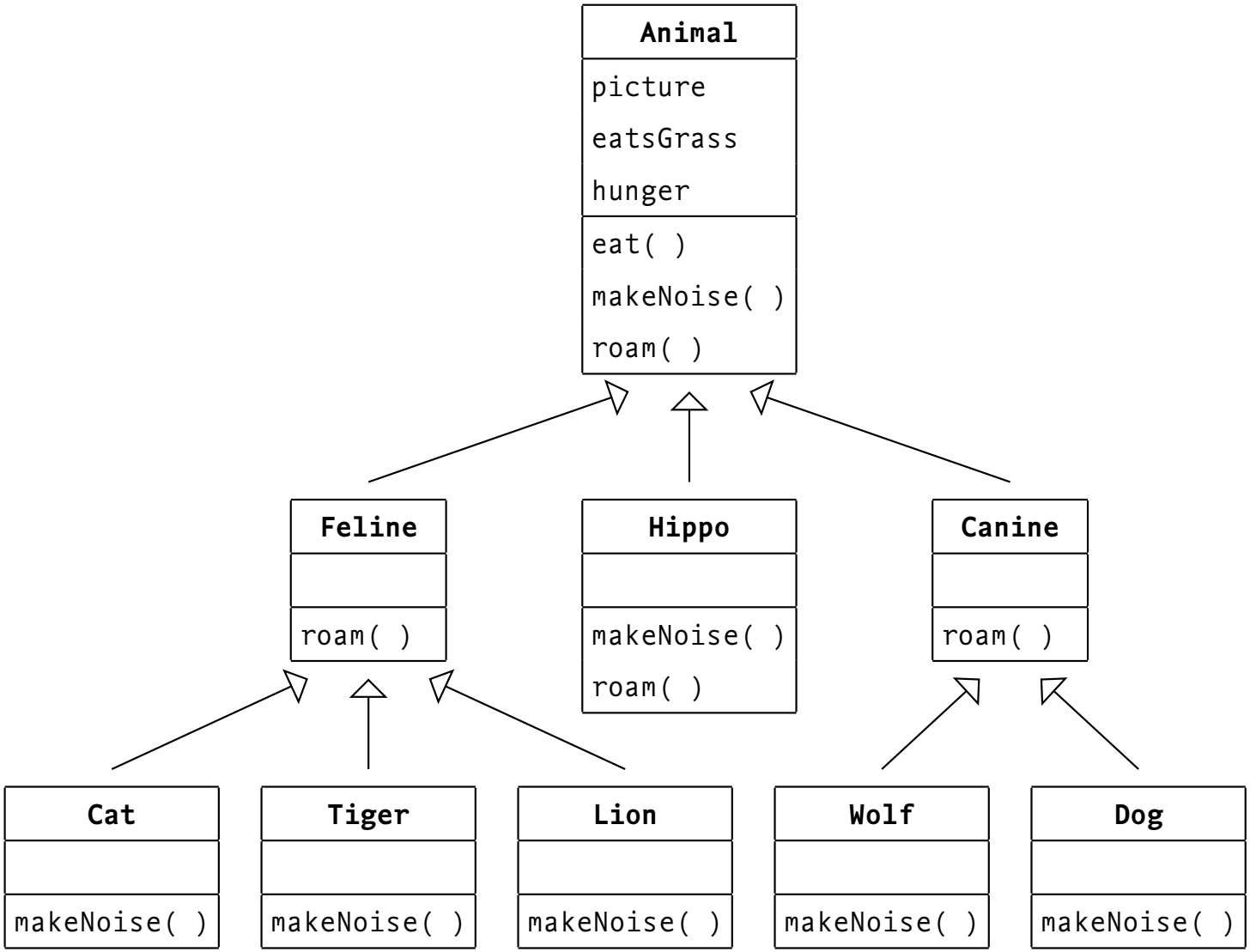
M. R. C. van Dongen

February 19, 2018

Outline

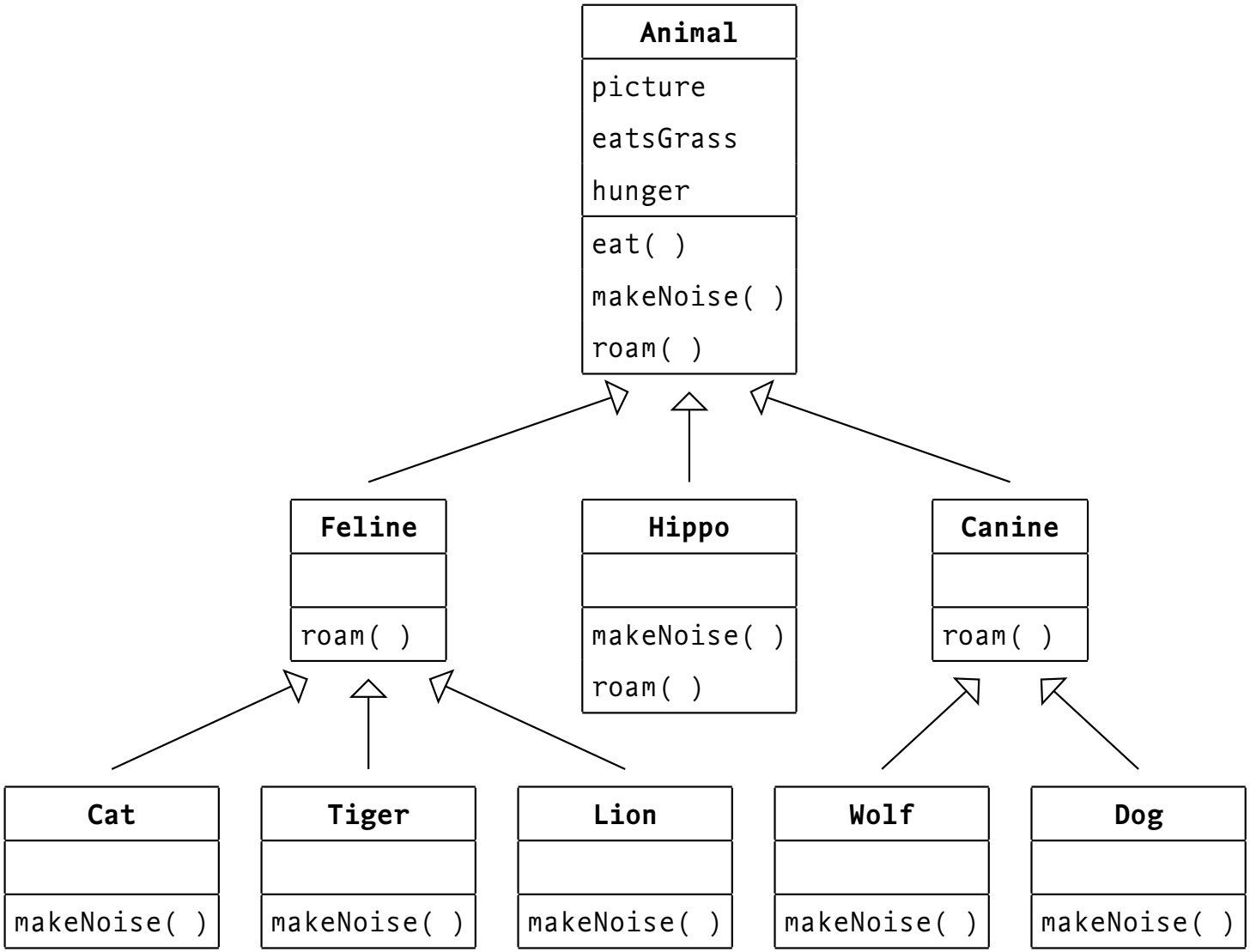
- Learn how to create abstract classes, which cannot be extended.
- Decide class membership with `instanceof`.
- Learn how to override some common methods:
  - `toString( );`
  - `equals( );`
- *Control* inheritance with `final` classes and methods.
- We study *multiple inheritance:*
  - We start with a case study;
  - We see advantages and disadvanteges of different designs;
  - We end up with a disasterous design complication;
  - We learn how overcome the complication.
- We study the *strategy design pattern:*
  - Defines a class of related algorithms;
  - Encapsulates them;
  - Makes them interchangable.
- We learn three disign principles:
  - Encapsulate what varies;
  - Program to an interface;
  - Favour composition over inheritance.

Abstract Classes



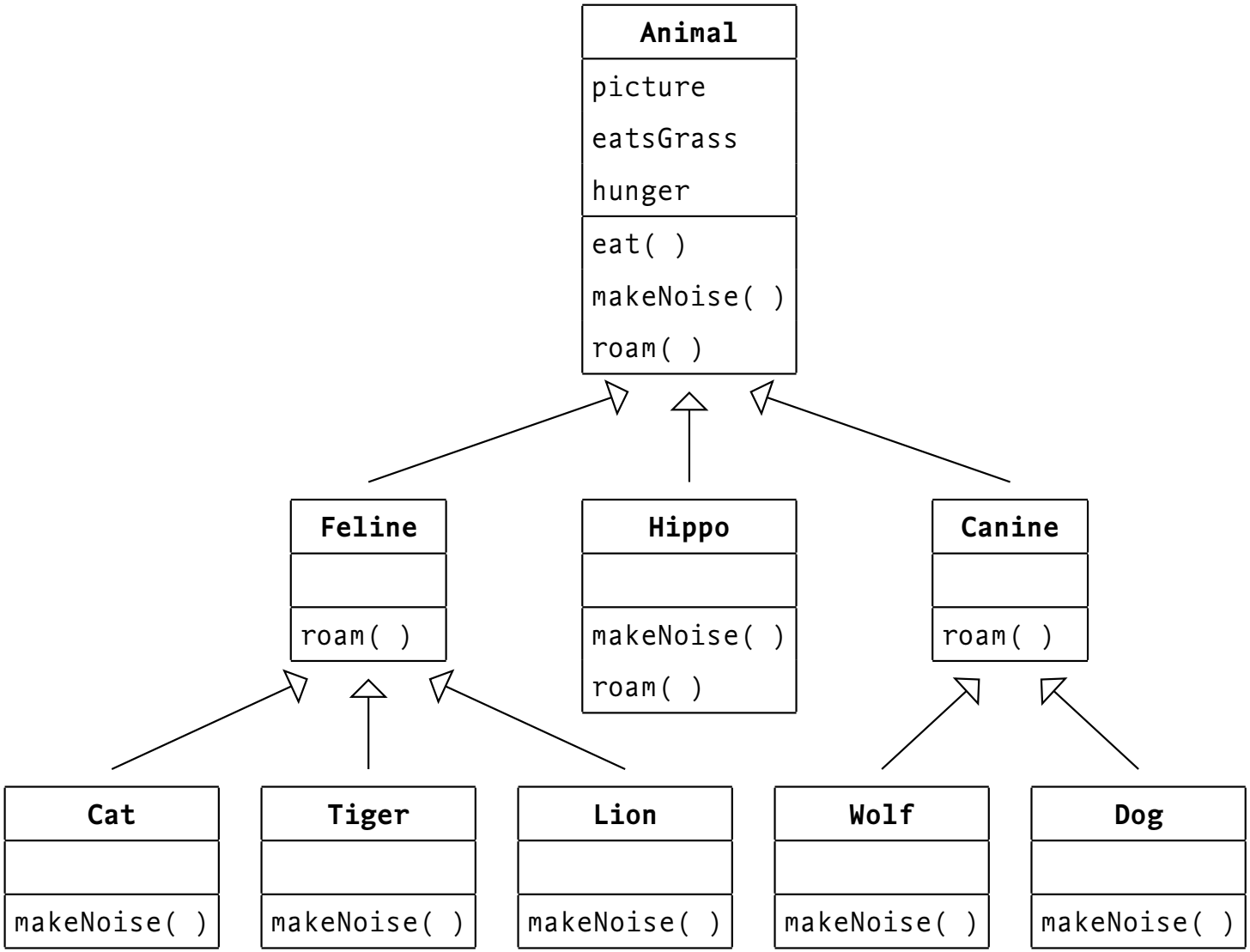
Abstract Classes

```
Hippo hippo = new Hippo( )
```



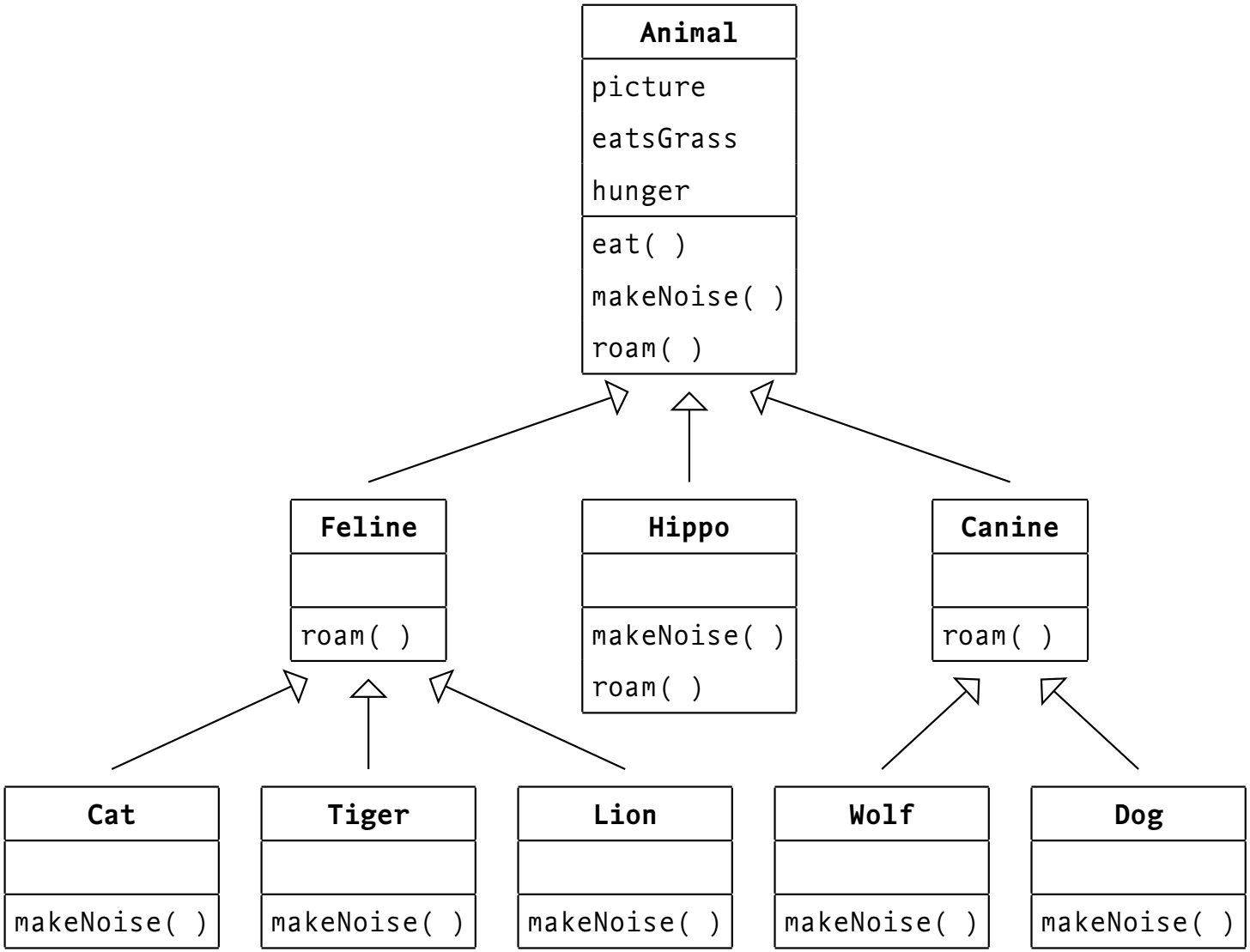
Abstract Classes

```
Hippo hippo = new Hippo( );
```



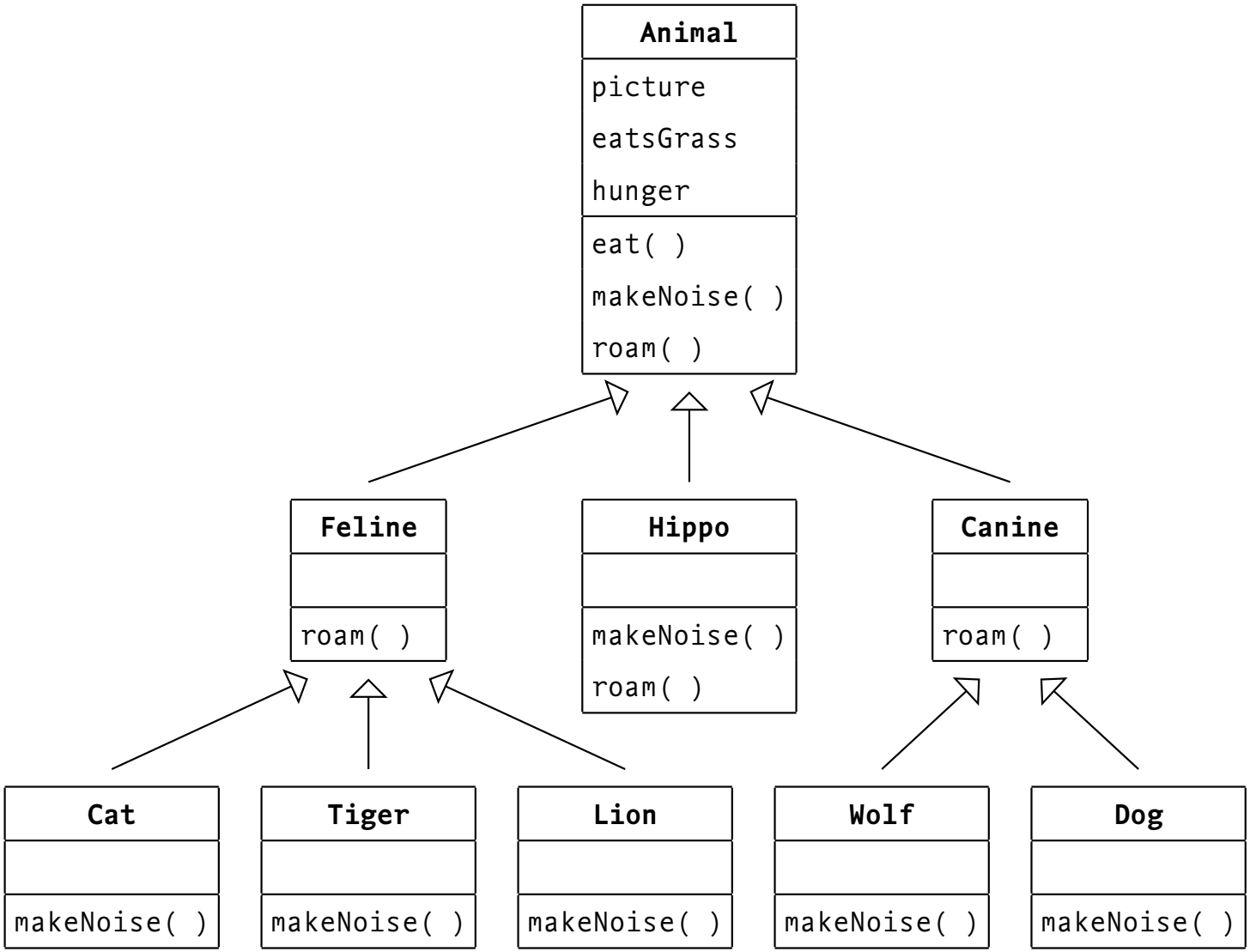
Abstract Classes

```
Cat cat = new Cat( )
```



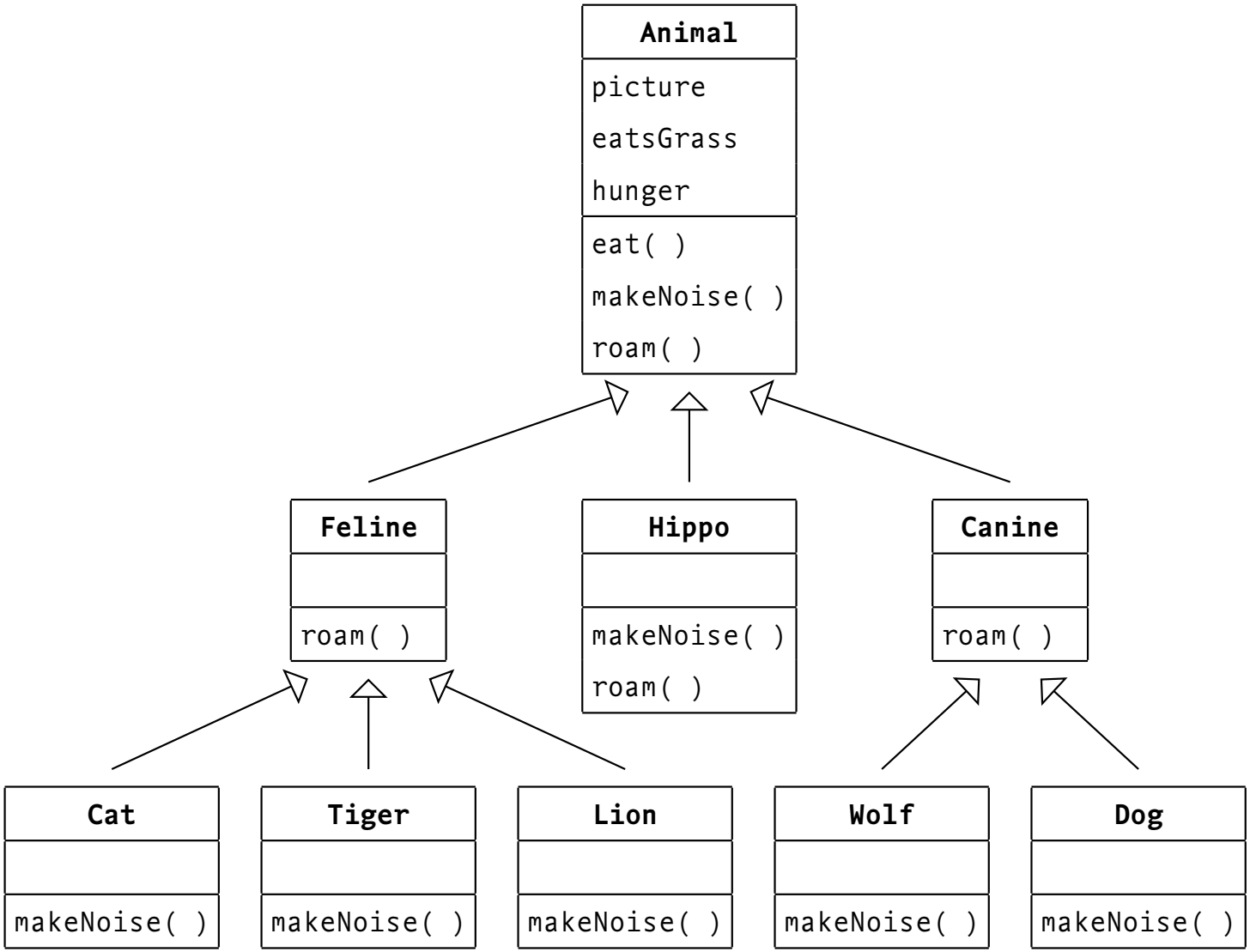
Abstract Classes

```
Cat cat = new Cat( );
```
√



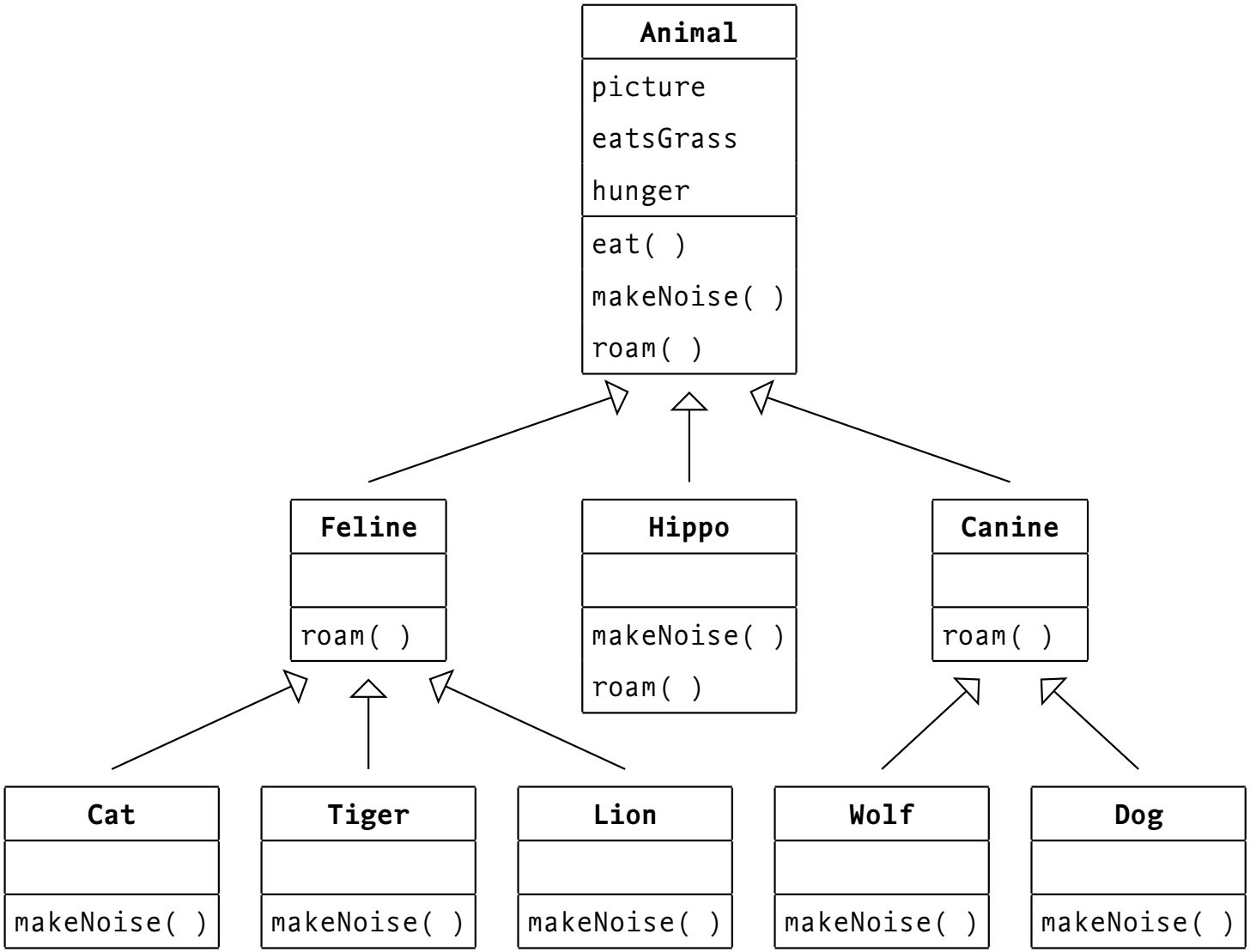
Abstract Classes

```
Animal animal = new Cat( )
```



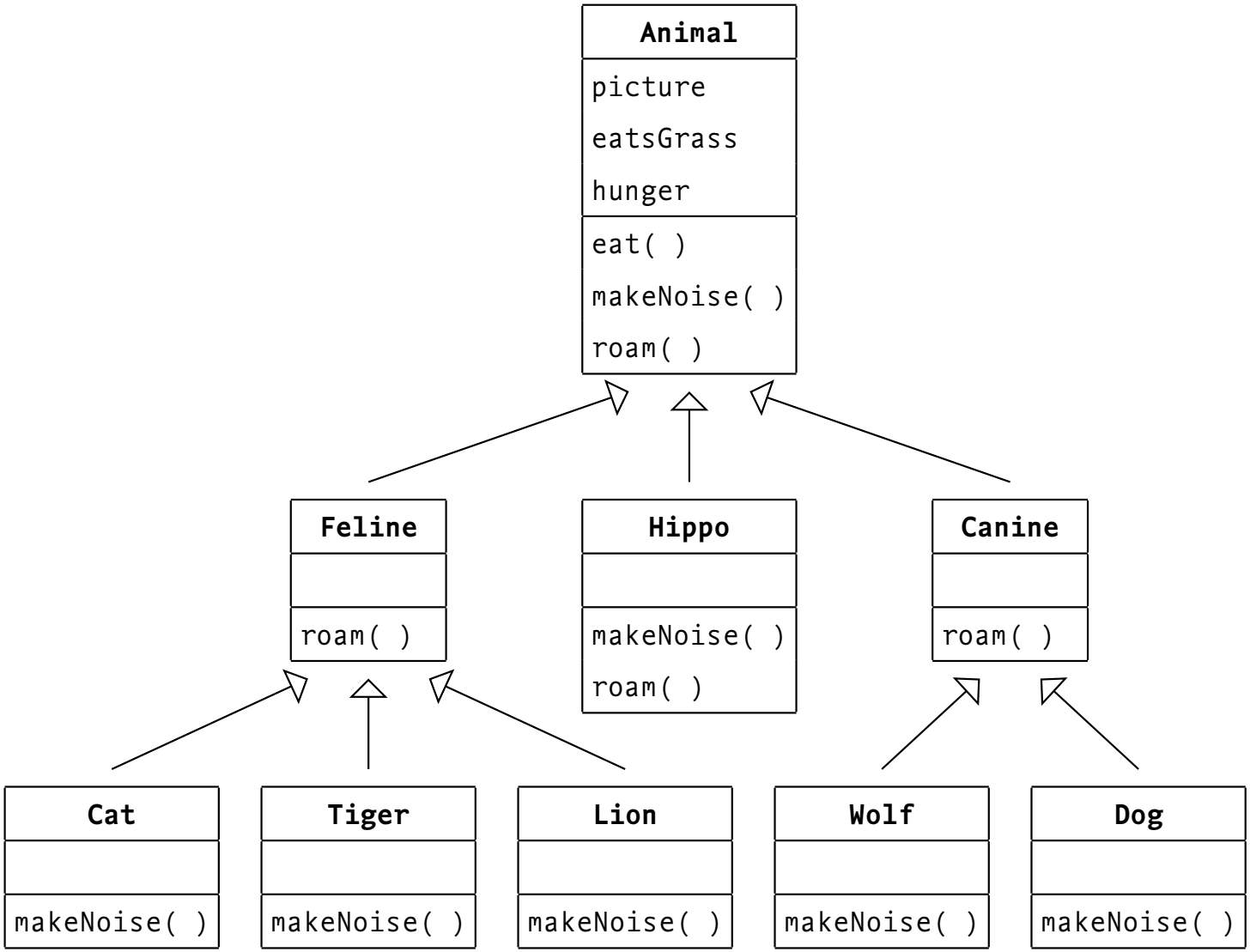
Abstract Classes

```
Animal animal = new Cat( );
```



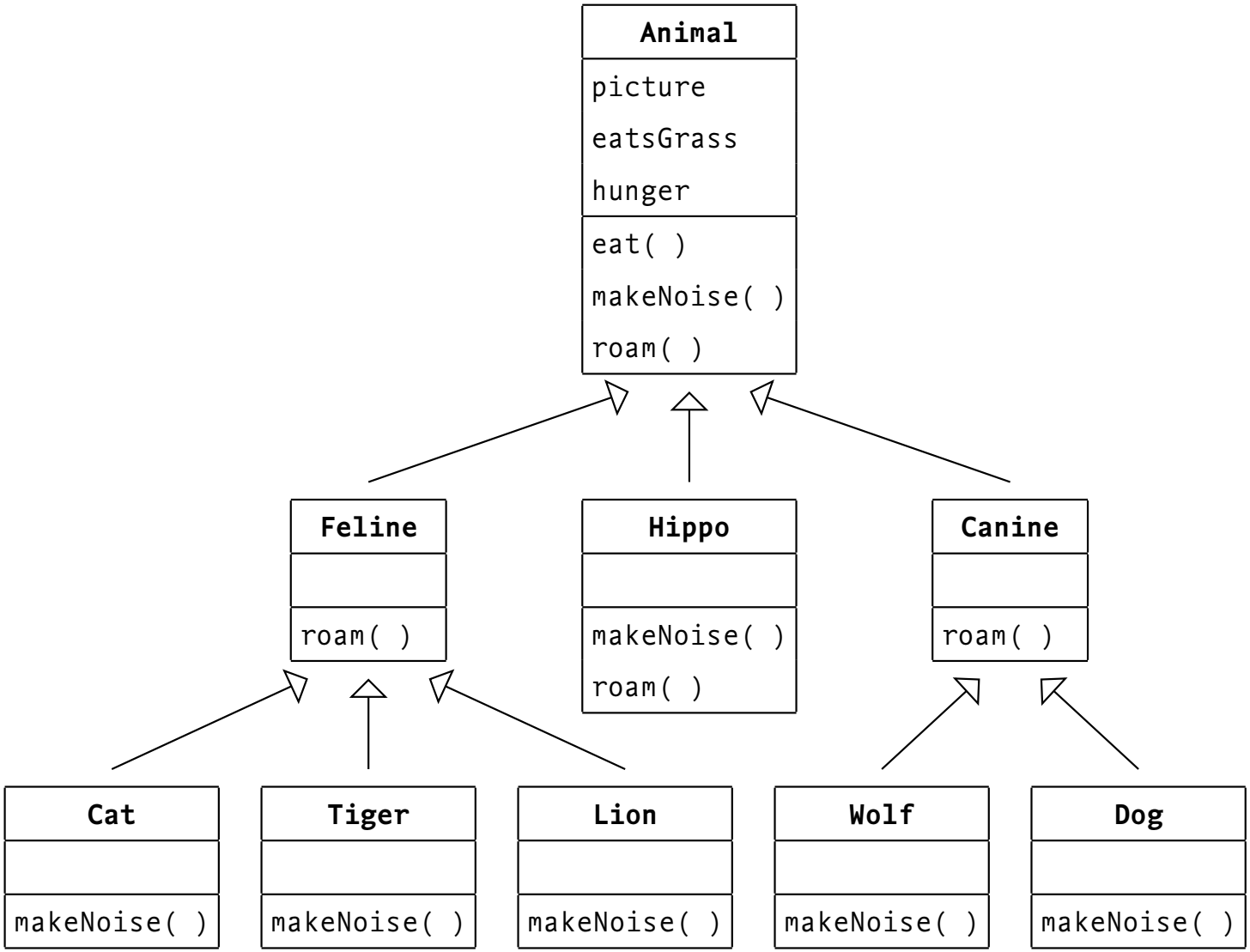
Abstract Classes

```
Animal animal = new Animal( )
```



Abstract Classes

```
Animal animal = new Animal( )???
```



Why do we Need the `Animal` Class?

- We need it for inheritance, so we can:
  - Share common code, and
  - Define a common API for `Animals`.
- We need it for polymorphism, so we can:
  - Write code that will still work if we add subclasses.

Some Classes Should Never be Instantiated

- We never wanted the `Animal` class to be instantiated.
- We want `Cat` and `Dog` objects, but not `Animal` objects.
- The spell `abstract` prevents classes from being instantiated.

Java

```
public abstract class Animal {
    ...
}
```

- Now `javac` won't let you instantiate abstract classes:

Don't Try This at Home

```
Animal animal = new Animal( );
```

Subclasses can be Abstract Too

Java

```
public abstract class Canine extends Animal {
    ...
}
```

Abstract and Concrete Classes

- A class is *abstract* if it's defined with the keyword `abstract`.
- Otherwise it is *concrete*.

Abstract and Concrete Classes (Continued)

- You can still use abstract polymorphic reference variables.

Java

```
Dog dog = new Dog( );
Cat cat = new Cat( );
Animal animal = dog;
animal = cat;
```

- But, you can only instantiate concrete classes.

Java

```
Cat cat = new Cat( );
Animal dog = new Dog( );
```

- Instantiating an abstract base class *array* is also allowed.

Java

```
Animal[] animals = new Animal[ 3 ];
```

Abstract Methods

- `Java` also has *abstract methods*.
  - They are defined in abstract classes,
  - They are defined with the keyword `abstract`, and
  - They have no body.

Java

```
public abstract void roam( );
```

Why Have Abstract Methods

- Abstract classes *must* be *extended*.
- Abstract methods *must* be *overridden*.
  - They define the nature of the common protocol.
  - They don't require a default implementation.
  - Saves you from forgetting to implement proper behaviour.

Why Have Abstract Methods

- Abstract classes *must* be *extended*.
- Abstract methods *must* be *overridden*.
  - They define the nature of the common protocol. √
  - They don't require a default implementation. √
  - Saves you from forgetting to implement proper behaviour. √

Implementing Abstract Methods

- Abstract methods have no body.
  - They only occur in abstract classes.
  - They have no default behaviour.
- Each concrete subclass needs the behaviour for its API.
- Therefore, you *have to* implement the abstract method.
- You implement an abstract method by providing a body.
  - This is called *overriding* the method.
  - This may be done in any class on the shortest path from concrete class to the abstract class that defines the abstract method.
  - So, implementing in abstract subclasses is allowed.
  - Of course, a method may be overridden, and overridden, ...

How does this Work?

Implementing the Method

Java

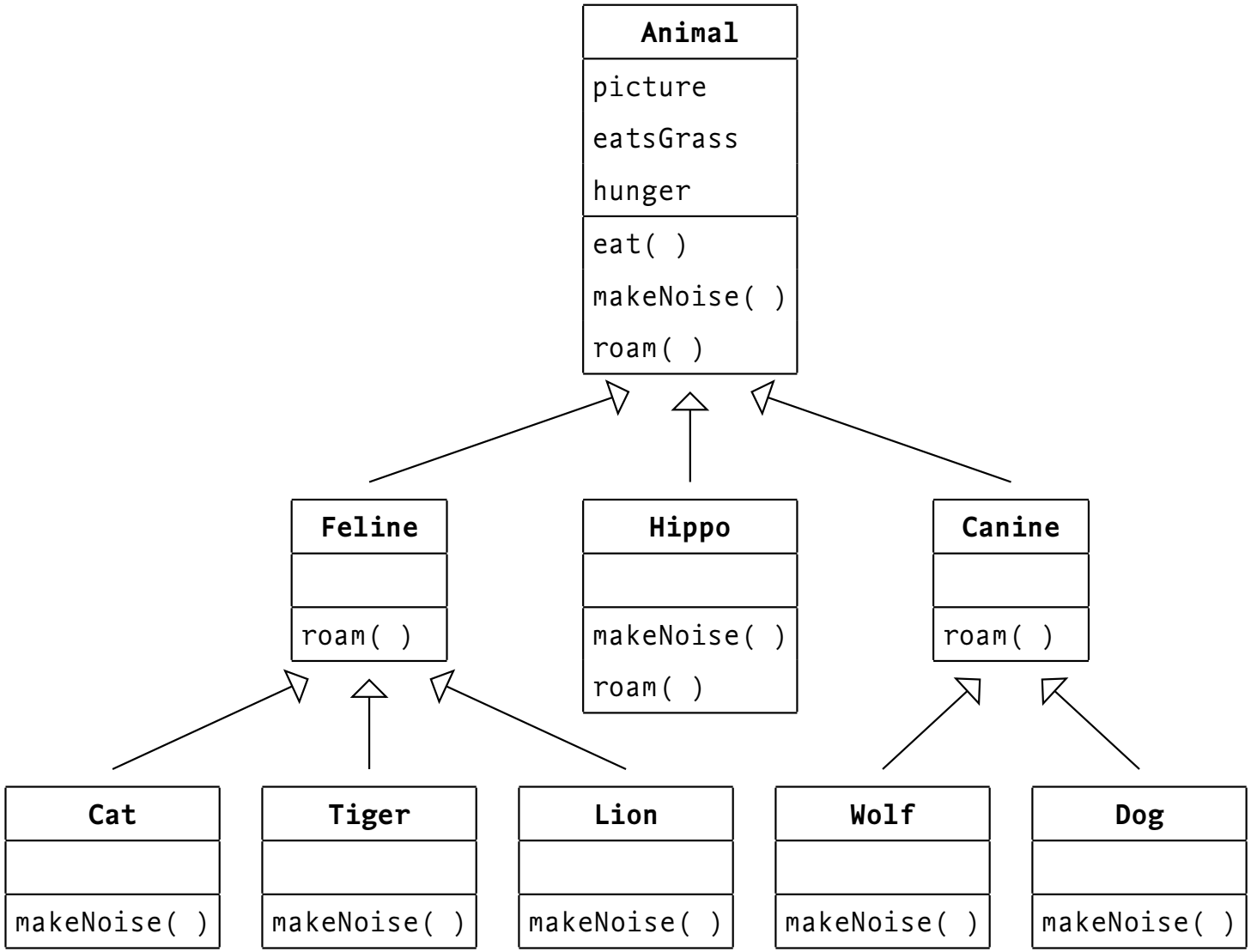
```
public abstract class Animal {
    public abstract void makeNoise( );
}
```

Java

```
public class Dog extends Animal {
    @Override
    public void makeNoise( ) { ... }
}
```

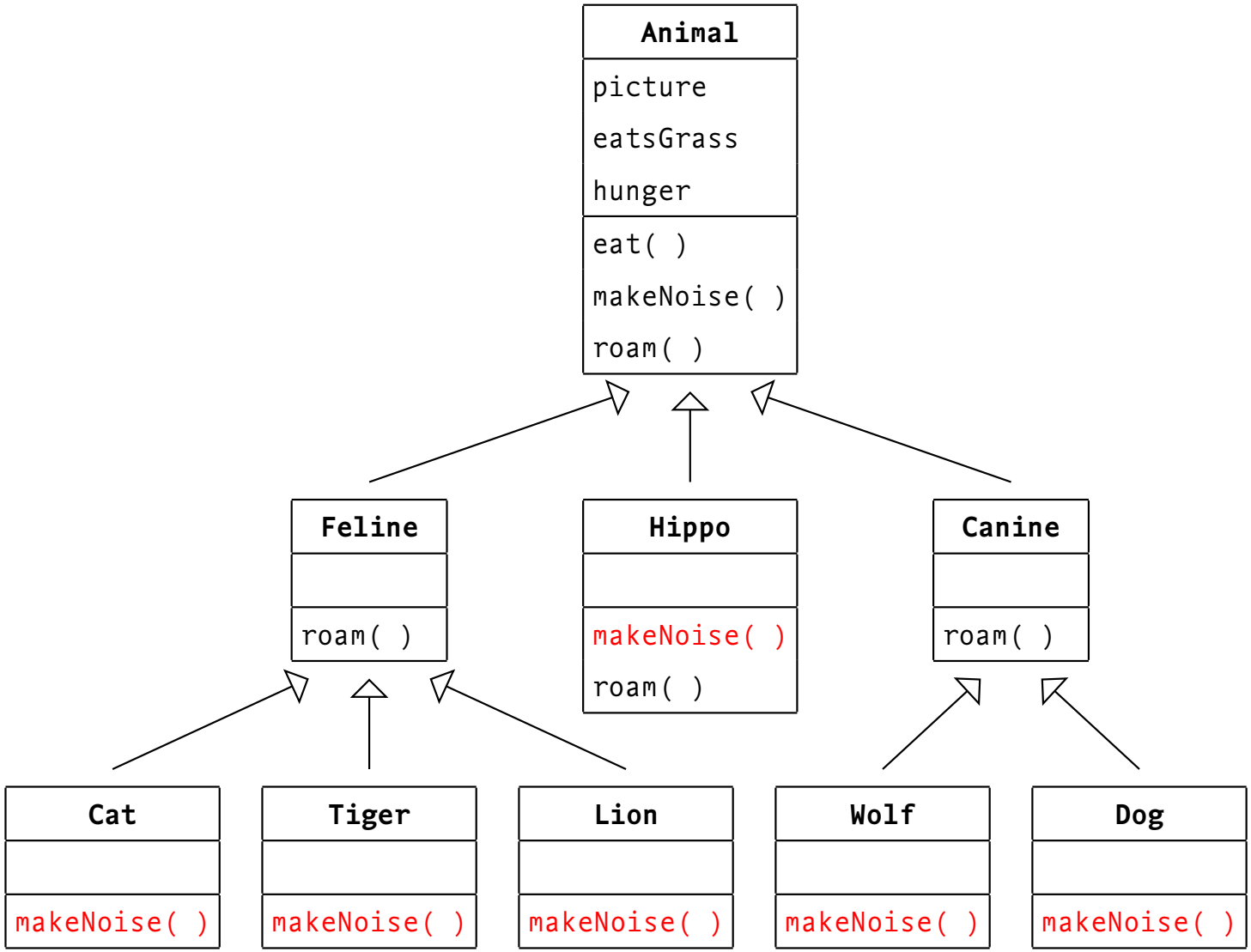
How does this Work?

Implementing the Method for all Relevant Classes



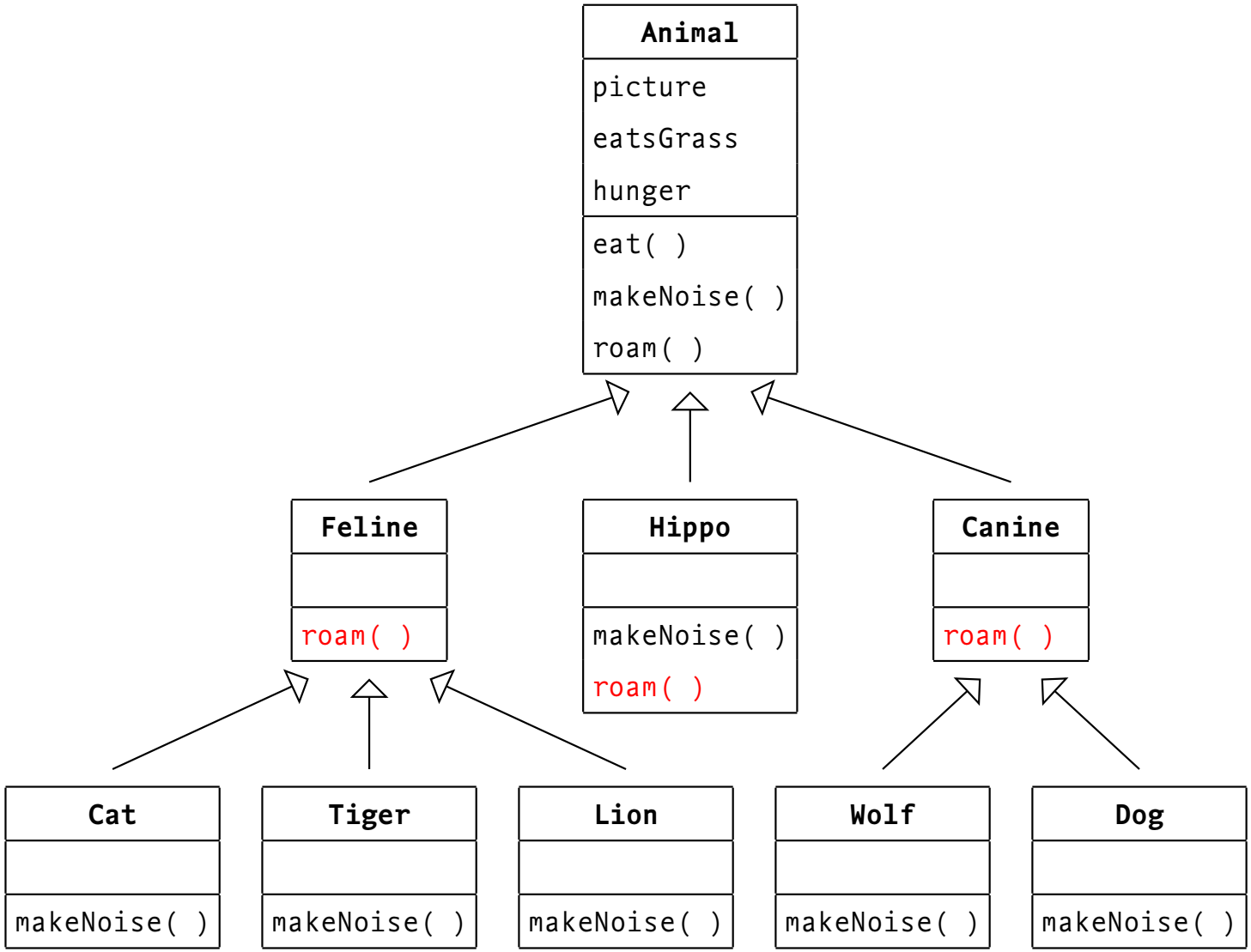
How does this Work?

Implementing the Method for all Relevant Classes



How does this Work?

Implementing the Method for all Relevant Classes



The Barber Paradox?

- A barber shaves people who don't shave themselves.
- There's a small town with only one barber.

The Barber Paradox?

- A barber shaves people who don't shave themselves.
- There's a small town with only one barber.
- Who shaves the barber?

Deciding Class Membership

- Sometimes you need to decide class/interface membership.
- For example, when a polymorphic variable's type is too loose.

Java

```
public class Person {
    public static void main( String[] args ) {
        final Barber barber = Barber.orderBarber( );
        final Person person = new Person( );
        person.shave( );
        barber.shave( );
    }

    public void shave( ) {
        final Person person = this;
        if (/* person is a Barber */) {
            final Barber barber = (Barber)person;
            barber.shaveYourself( );
        } else {
            final Barber barber = Barber.orderBarber( );
            barber.shave( person );
        }
    }
}
```

- More important applications a few slides further on.

Deciding Class Membership

- Sometimes you need to decide class/interface membership.
- For example, when a polymorphic variable's type is too loose.

Java

```
public class Barber extends Person {
    private Barber( ) { }

    public static Barber orderBarber( ) {
        return new Barber( );
    }

    public void shaveYourself( ) {
        System.out.println( "Shaving myself" );
    }

    public void shave( final Person person ) {
        System.out.println( "Shaving person" );
    }
}
```

- More important applications a few slides further on.

Deciding Class Membership

- Sometimes you need to decide class/interface membership.
- For example, when a polymorphic variable's type is too loose.

Java

```
public class Person {
    public static void main( String[] args ) {
        final Barber barber = Barber.orderBarber( );
        final Person person = new Person( );
        person.shave( );
        barber.shave( );
    }

    public void shave( ) {
        final Person person = this;
        if (person instanceof Barber) {
            final Barber barber = (Barber)person;
            barber.shaveYourself( );
        } else {
            final Barber barber = Barber.orderBarber( );
            barber.shave( person );
        }
    }
}
```

- More important applications a few slides further on.

Deciding Subclass Membership

- The spell `reference instanceof Clazz` tests for class membership of `Clazz`.
- It returns:
  - `true` if `reference` references an instance of `Clazz`;
  - `true` if `reference` references an instance of a subclass of `Clazz`;
  - `false` otherwise (including when `reference == null`).
- The test also works for interfaces.

Java

```
final String bomb = "blast";

if (bomb instanceof Comparable) {
    System.out.println( bomb );
}
```

Overriding `toString( )`

- The `Object` class defines `public String toString( );`
  - It's an instance method.
- It should return a "meaningful" representation of its instance.
- Arguably most classes should override the method.
- It's especially useful when testing.

How?

Depends on the Class

Java

```
public class Person {
    private final String firstName;
    private final String surname;

    ...

    @Override
    public String toString( ) {
        return firstName + " " + surname;
    }
}
```

How?

Depends on the Class

Java

```
public class Die {
    private final Random generator; // not printed
    private int faceValue;
    ...

    @Override
    public String toString( ) {
        return Integer.toString( faceValue );
    }
}
```

How?

Depends on the Class

Java

```
public class DataBaseConnection {
    private final Database db;
    private final long id;
    private final Port port;
    ...

    @Override
    public String toString( ) {
        return "DatabaseConnection[ id = " + id
                                + ", db = " + db
                                + ", port = " + port
                                + ... // all attributes
                                + " ]";
    }
}
```

How?

Depends on the Class

Java

```
public class DataBaseConnection {
    private final Database db;
    private final long id;
    private final Port port;
    ...

    @Override
    public String toString( ) {
        return "DatabaseConnection[ id = " + id
                                + ", db = " + db // ????
                                + ", port = " + port
                                + ... // all attributes
                                + " ]";
    }
}
```

How?

Depends on the Class

Java

```
public interface Testable {
    public String getTestOutput( );
}

public class Port implements Testable { ... }
public class Database implements Testable { ... }

public class DataBaseConnection implements Testable {
    private final Database db;
    private final long id;
    private final Port port;
    ...

    // Better!
    @Override
    public String getTestOutput( ) {
        return "DatabaseConnection[ id = " + id
                                + ", db = " + db.getTestOutput( )
                                + ", port = " + port.getTestOutput( )
                                + ... // all attributes
                                + " ]"l
    }
}
```

Overriding `equals( )`

- `public boolean equals( Object object ):`
  - Also defined in the `Object` class.
- Method is supposed to test for deep equality.
- Easy if you know the base class of `object`:

Java

```
public class Person {
    private final String firstName;
    private final String surname;

    ...

    @Override
    public boolean equals( Object object ) {
        final Person that = (Person)object;
        return this.firstName.equals( that.firstName )
                  && this.surname.equals( that.surname );
    }
}
```

Overriding `equals( )`

- `public boolean equals( Object object ):`
  - Also defined in the `Object` class.
- Method is supposed to test for deep equality.
- Easy if you know the base class of `object`:

Java

```
public class Person {
    private final String firstName;
    private final String surname;

    ...

    @Override
    public boolean equals( Object object ) {
        final Person that = (Person)object;
        return this.firstName.equals( that.firstName )
                  && this.surname.equals( that.surname );
    }
}
```

- But what if you don't know the base class?

Overriding equals( )

If You're Not Sure About Base Class

Java

```
public class Person {
    private final String firstName;
    private final String surname;

    ...

    @Override
    public boolean equals( Object object ) {
        final boolean result;

        if (object instanceof Person) {
            final Person that = (Person)object;
            result = this.firstName.equals( that.firstName )
                        && this.surname.equals( that.surname );
        } else {
            result = false;
        }

        return result;
    }
}
```

Controlling Inheritance

- In `Java` a subclass inherits all `public` methods and attributes.
- This is useful but public methods may lead to problems.
  - E.g. what if a malicious subclass overrides a method?
- It's clear that more control is needed.
- In `Java` you can restrict inheritance and method overriding:
  1. If you make a class `final` you can't extend it.
  2. If you make a method `final` you can't override it.

Making the Class Final

Java

```
public class Word {
    public void word( ) {
        System.out.println( "It is." );
    }
}
```

Making the Class Final

Java

```
public class Word {
    public void word( ) {
        System.out.println( "It is." );
    }
}

public class Rebuttal extends Word {
    // You can extend this class.
    public void word( ) {
        System.out.println( "Oh no it isn't." );
    }
}
```

Making the Class Final

Java

```
public class Word {
    public void word( ) {
        System.out.println( "It is." );
    }
}

public class Rebuttal extends Word {
    // You can extend this class.
    public void word( ) {
        System.out.println( "Oh no it isn't." );
    }
}

public final class LastWord extends Rebuttal {
    // You cannot extend this class.
    …
    @Override
    public void word( ) {
        System.out.println( "Oh yes it is." );
    }
}
```

Why Make a Class Final?

Why Make a Class Final?

Inheritance Violates Encapsulation

- With method overriding, client classes may change behaviour.
- Almost as bad as providing them direct attribute access.
- Here methods, not attributes, are exposed to modification.

Security: Makes sure the class does what it should do.

- An overridden method may misbehave.
  - Makes it impossible to enforce invariants.
- A `String` should behave as a `String`.
  - Should be impossible to override this method.

Maintenance: Clients may start to rely on overridden behaviour.

Making the Method Final

Java

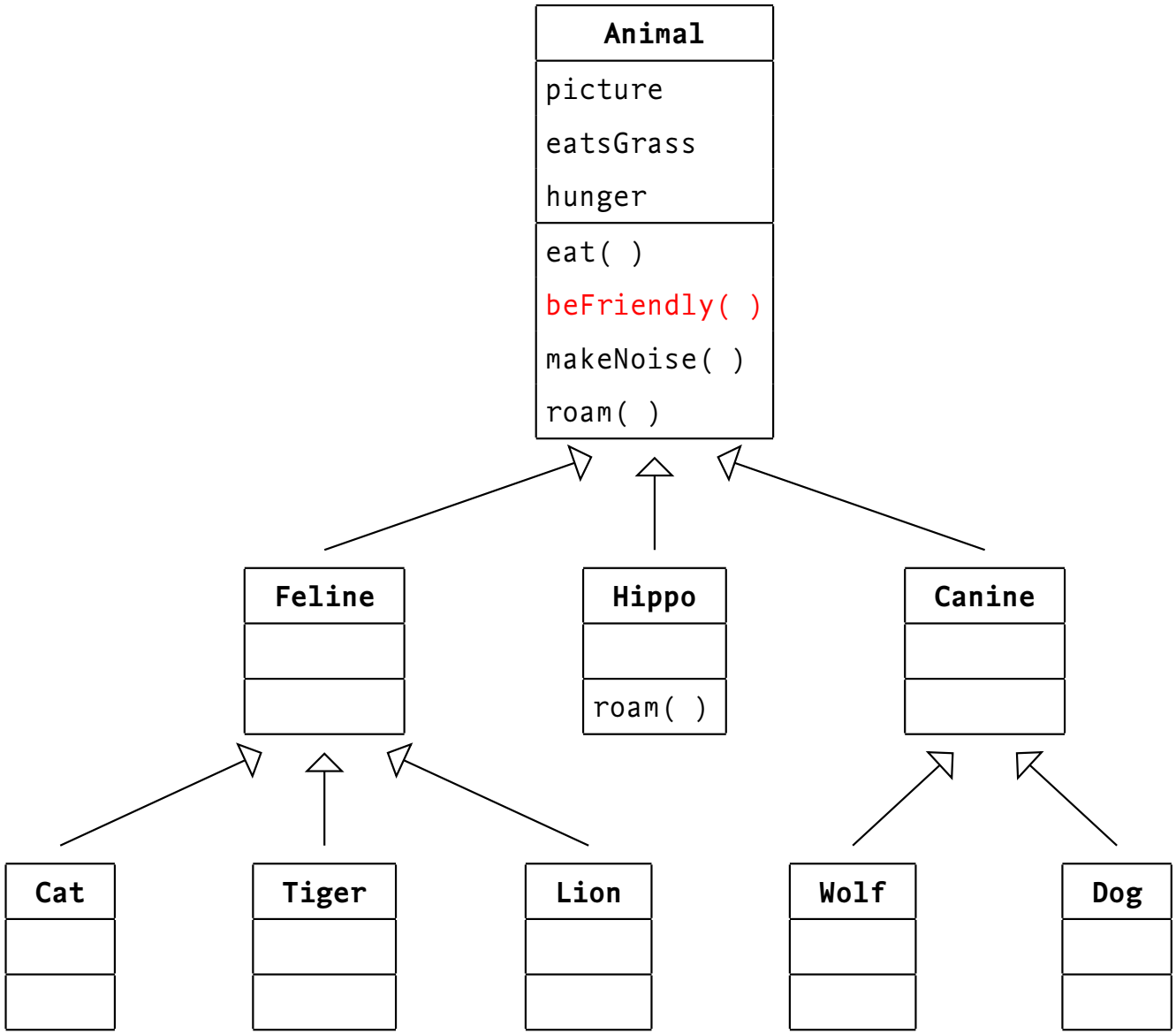
```
public class Example {
    // You may not override this method.
    public final void finalMethod( ) { … }
    // You may override this method.
    public void overridableMethod( ) { … }
}
```

Multiple Inheritance

- Let's introduce `Pets` to our `Animal` class hierarchy.
- The `Pets` can `beFriendly( )`.
- Other animals don't have `beFriendly( )` behaviour.
- Our design should allow for polymorphic pet variables.

Adding Pets to our Fota Application

Option I: Adding the `Pet` Method to the `Animal` Class



Adding Pets to our Fota Application

Option I: Adding the `Pet` Method to the `Animal` Class

Pros: The are two main advantages:

1. All `Pets` will inherit `Pet` behaviour, and
2. `Animal` can act as a polymorphic type for `Pets`.

Cons: There are also disadvantages:

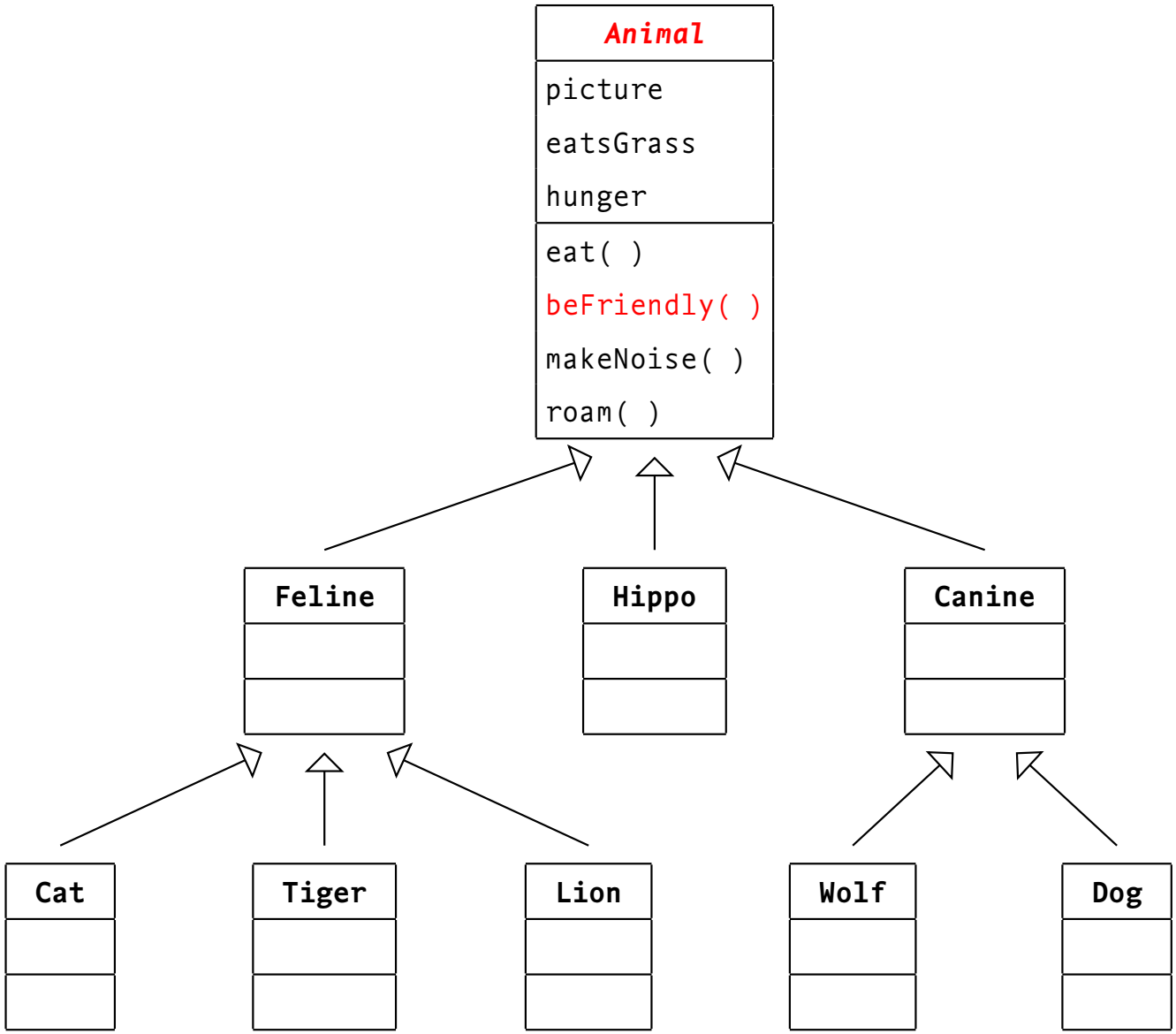
1. We don't have a proper `Pet` type.
2. Non-`Pets` will also get `beFriendly( )` behaviour.
3. Still must override `beFriendly( )` for `Dog` & `Cat`.

Conclusion: Clearly the disadvantages outweigh the advantages.

Cause: The Is-A test fails for non-`Pets`.

Adding Pets to our Fota Application

Option II: As Option I but Make `Animal` Class Abstract



Adding Pets to our Fota Application

Option II: As Option I but Make `Animal` Class Abstract

Pros: The advantages are better than before.

1. We can make all animals behave appropriately.
2. `Animal` can act as a polymorphic type for `Pets`.

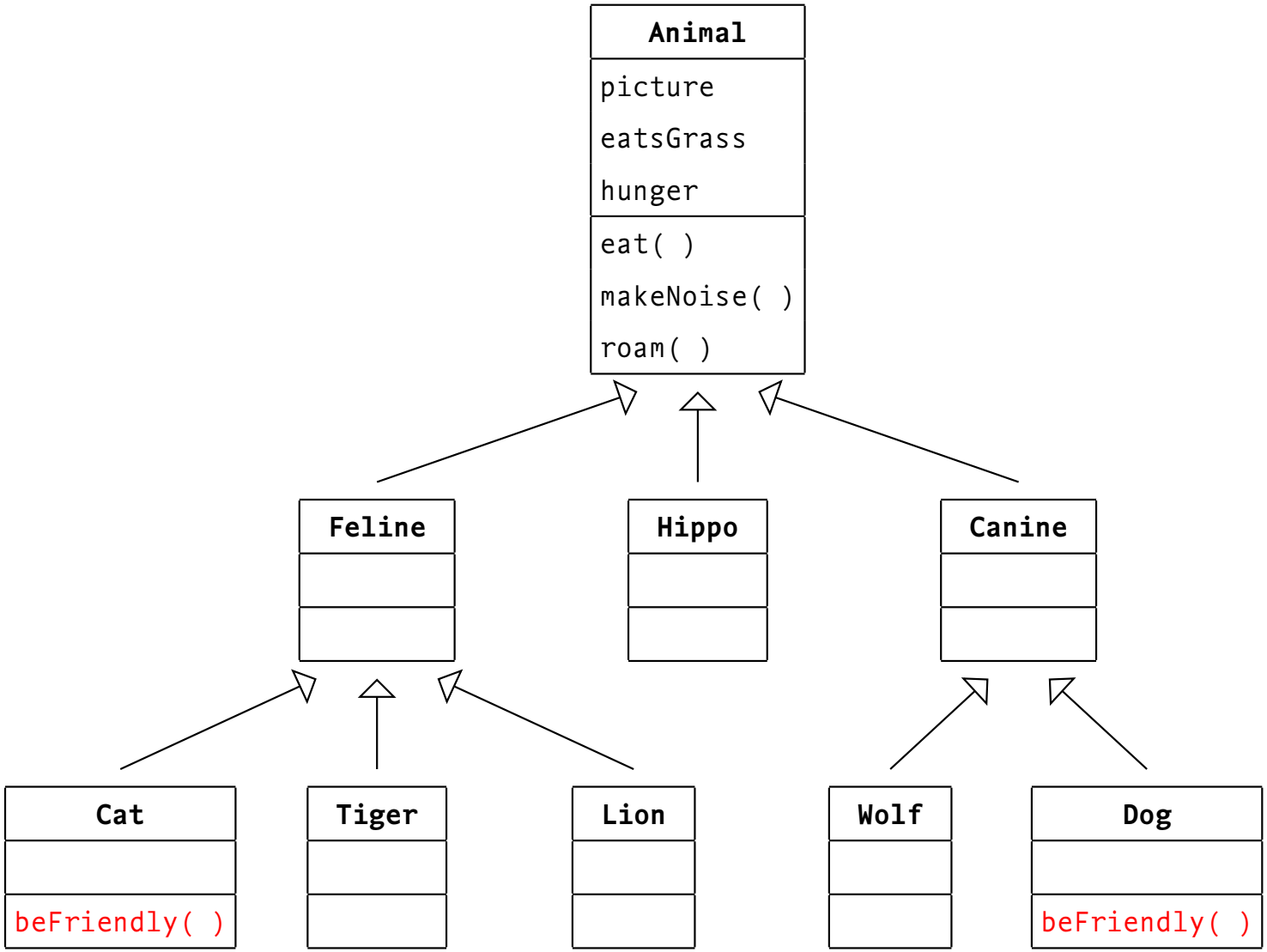
Cons: We still don't have a proper `Pet` type.
Must override `beFriendly( )` in all concrete classes.

Conclusion: This design is worse than Option I.

Cause: The Is-A test fails for non-`Pets`.

Adding Pets to our Fota Application

Option III: Put the `Pet` Method where It Belongs



Adding Pets to our Fota Application

Option III: Put the `Pet` Method where It Belongs

Pros: The following are some advantages.

- Definition of `beFriendly( )` is where it belongs.
- Implementing `beFriendly( )` requires little effort.
- All animals behave appropriately.

Cons: The following are some disadvantages.

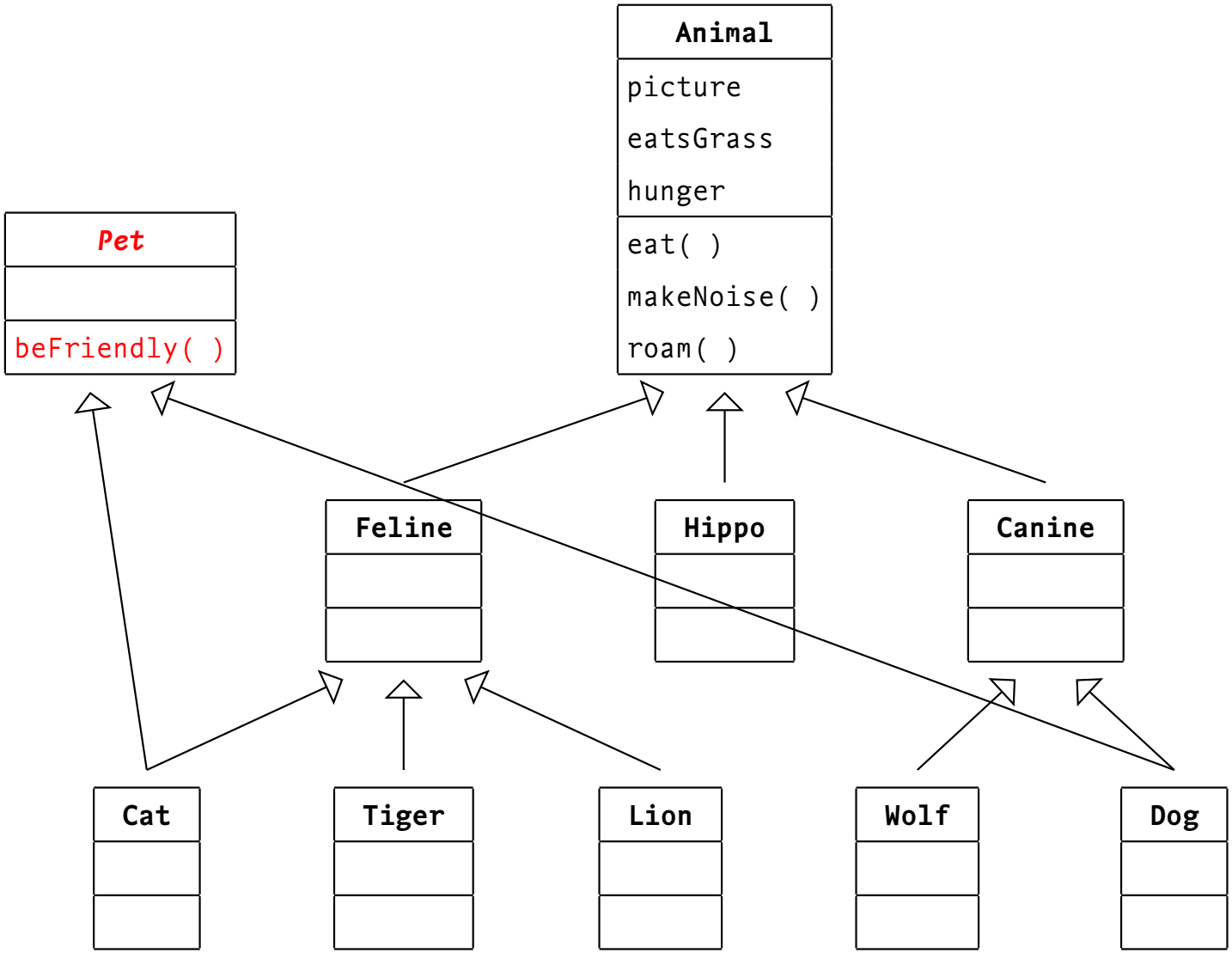
- We still don't have a proper `Pet` type.
- The `befriendly( )` method isn't abstract.
  - We can't guarantee a consistent `beFriendly( )`.
- We lose a proper polymorphic type for `Pets`.

Conclusion: This design makes `Pets` difficult to work with.

Cause: Polymorphism is a requirement for most applications.

Adding Pets to our Fota Application

Option IV: Two Superclasses for Pets



Adding Pets to our Fota Application

Option IV: Two Superclasses for `Pets`

Pros: The following are the advantages.

- The `beFriendly( )` method is where it belongs.
- Implementing `beFriendly( )` requires little effort.
- Guarantees consistent `beFriendly( )` definitions.
- `Pet` can act as a polymorphic type for pets.

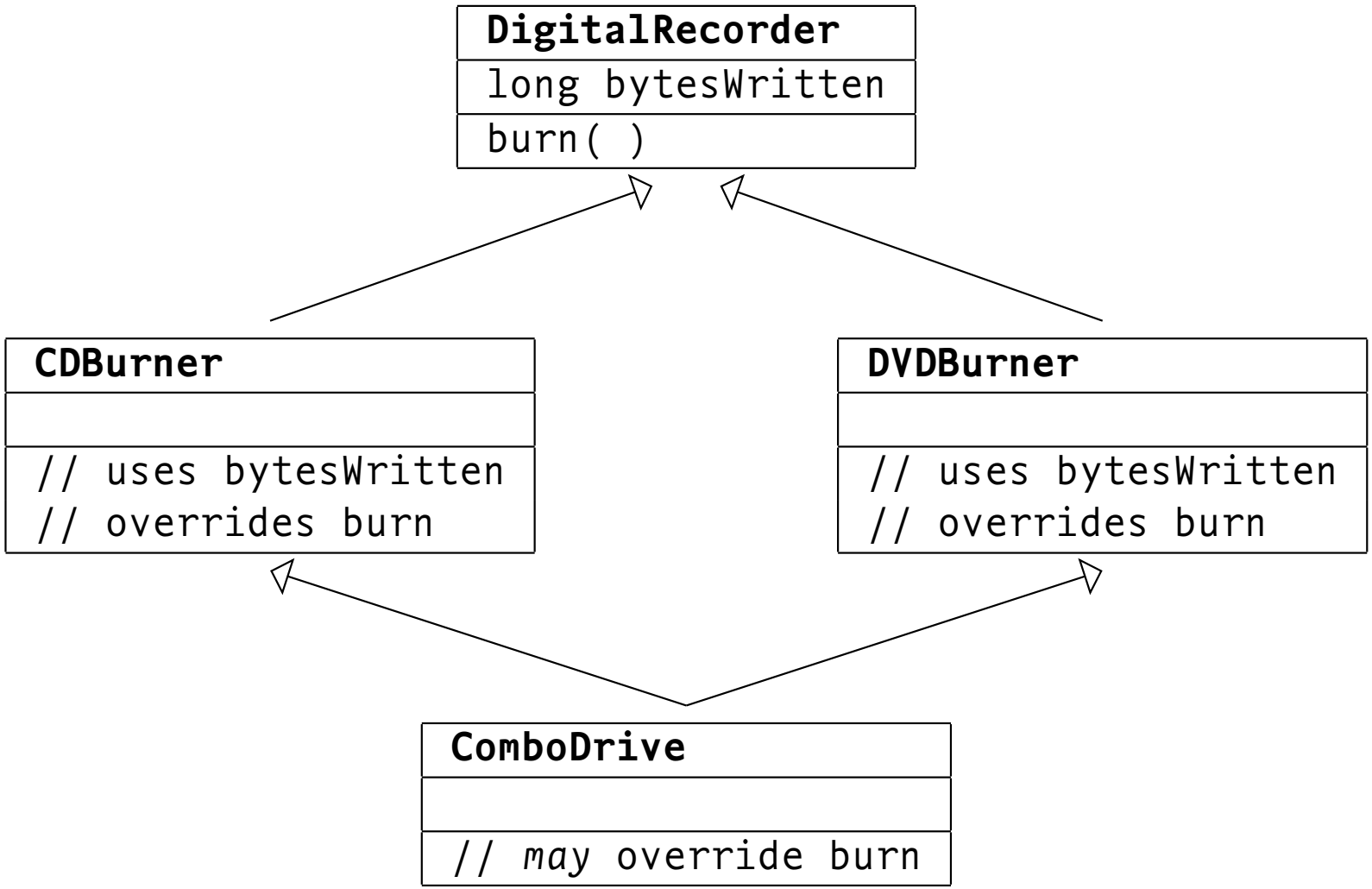
Cons: `Java` doesn't allow *multiple inheritance*.

Conclusion: This design is ideal but impossible.

Cause: A decision by the `Java` language designers.

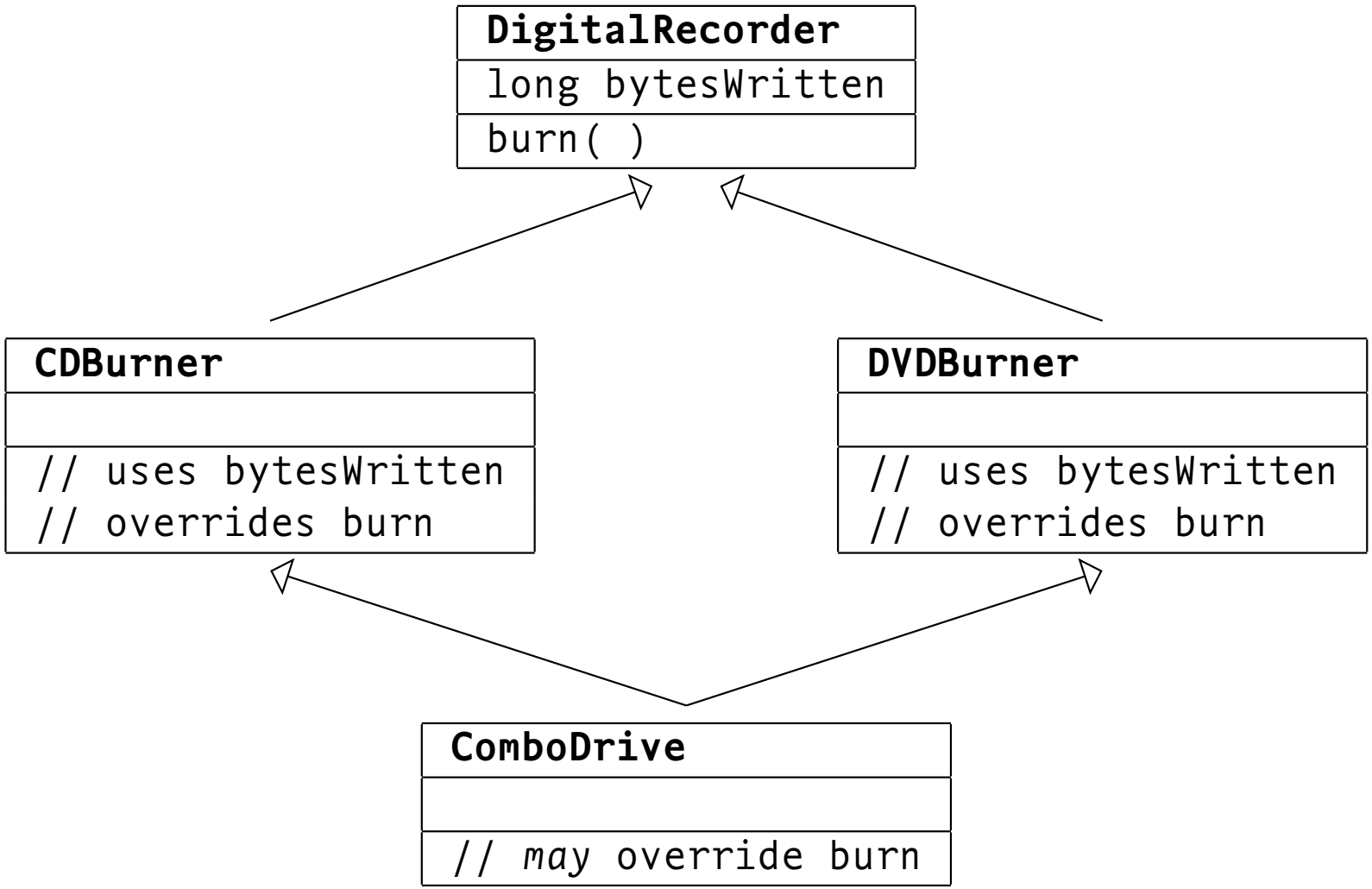
Deadly Diamond of Death

Different Assumptions about Valid Values for `bytesWritten`



Deadly Diamond of Death

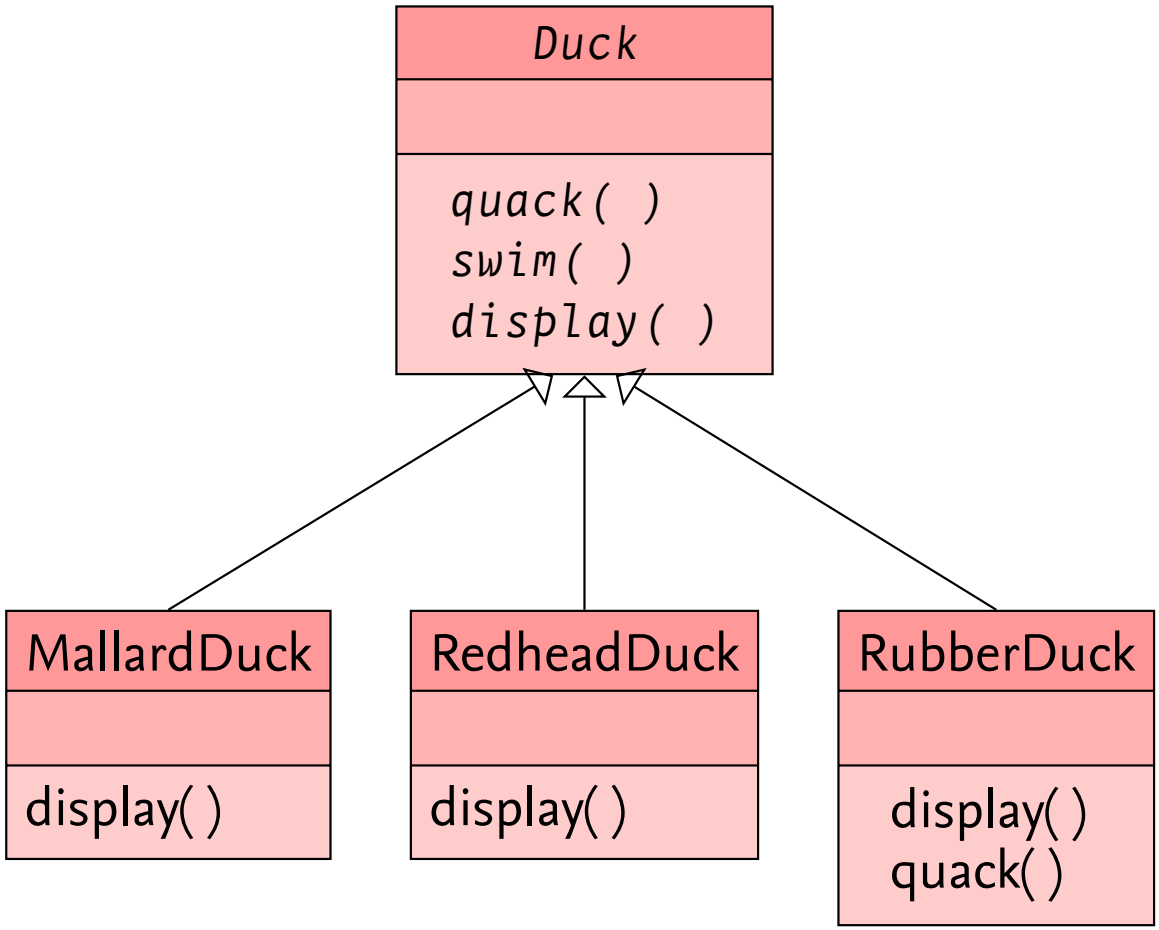
Which `burn( )` is Inherited by `ComboDrive`?



Implementing Duck Games

- Joe works at SimuDuck™.
- SimuDuck™ specialises in  pond simulation games.
  - These games involves lots of quacking and swimming  s.
- Joe is in charge of SimuDuck™'s most popular game.
- The game is written in Java and is based on inheritance.

The Design



Enters Mr Change

Enters Mr Change

Enters Mr Change

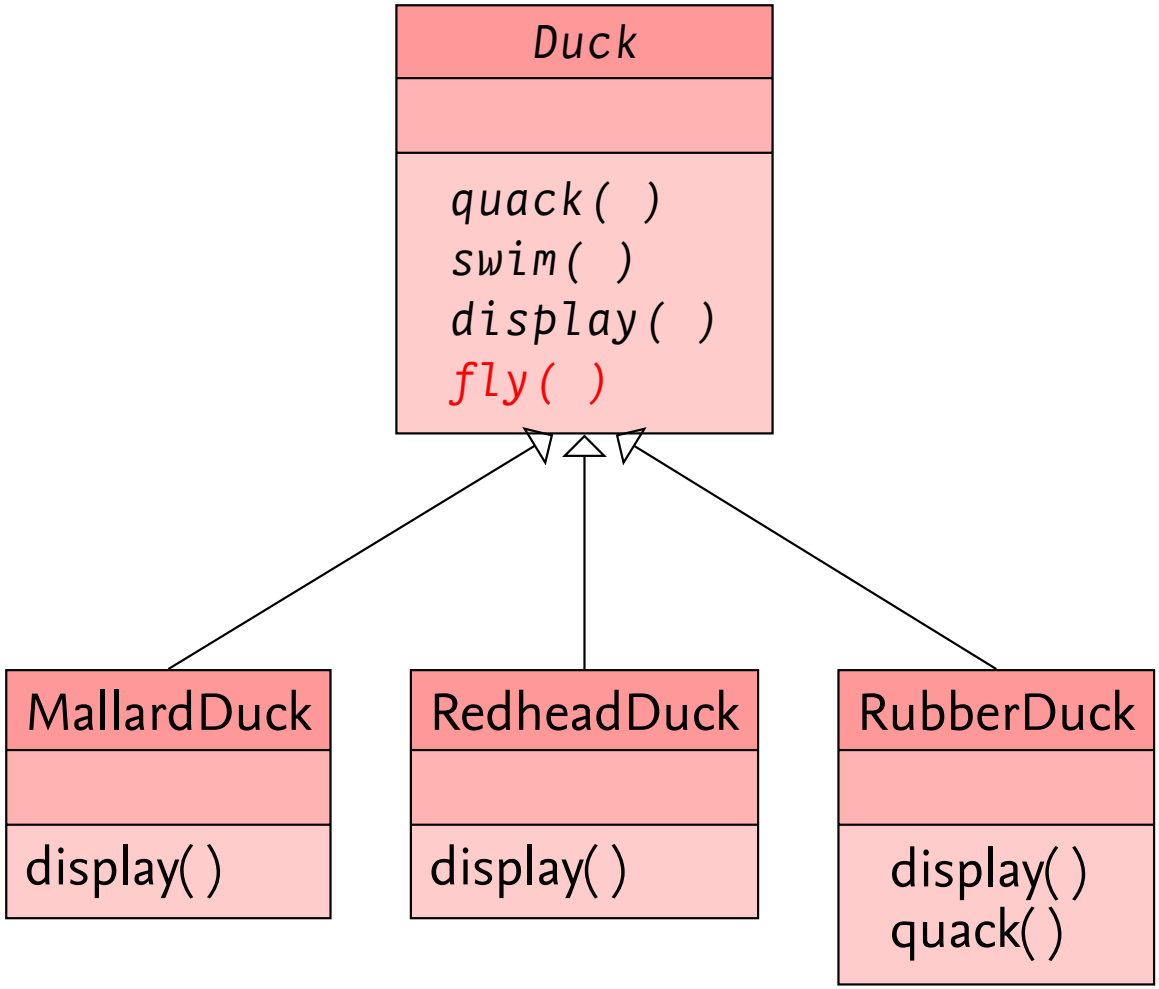
Enters Mr Change

Enters Mr Change

Enters Mr Change

Enters Mr Change

The Design



The Verdict?

The Verdict?

The Verdict?

What Has Gone Wrong?

- At first Joe didn't understand what had gone wrong.

What Has Gone Wrong?

- At first Joe didn't understand what had gone wrong.
- It was inheritance that was causing the problem.
  - The `Duck` class defined the default `fly( )` behaviour.
  - This was inherited by *all* `Duck` subclasses.
  - None of the subclasses overrode the behaviour.
  - Therefore all  s had the default `fly( )` behaviour.
  - Including `RubberDucks`.

What Should Joe Do?

What Should Joe Do?

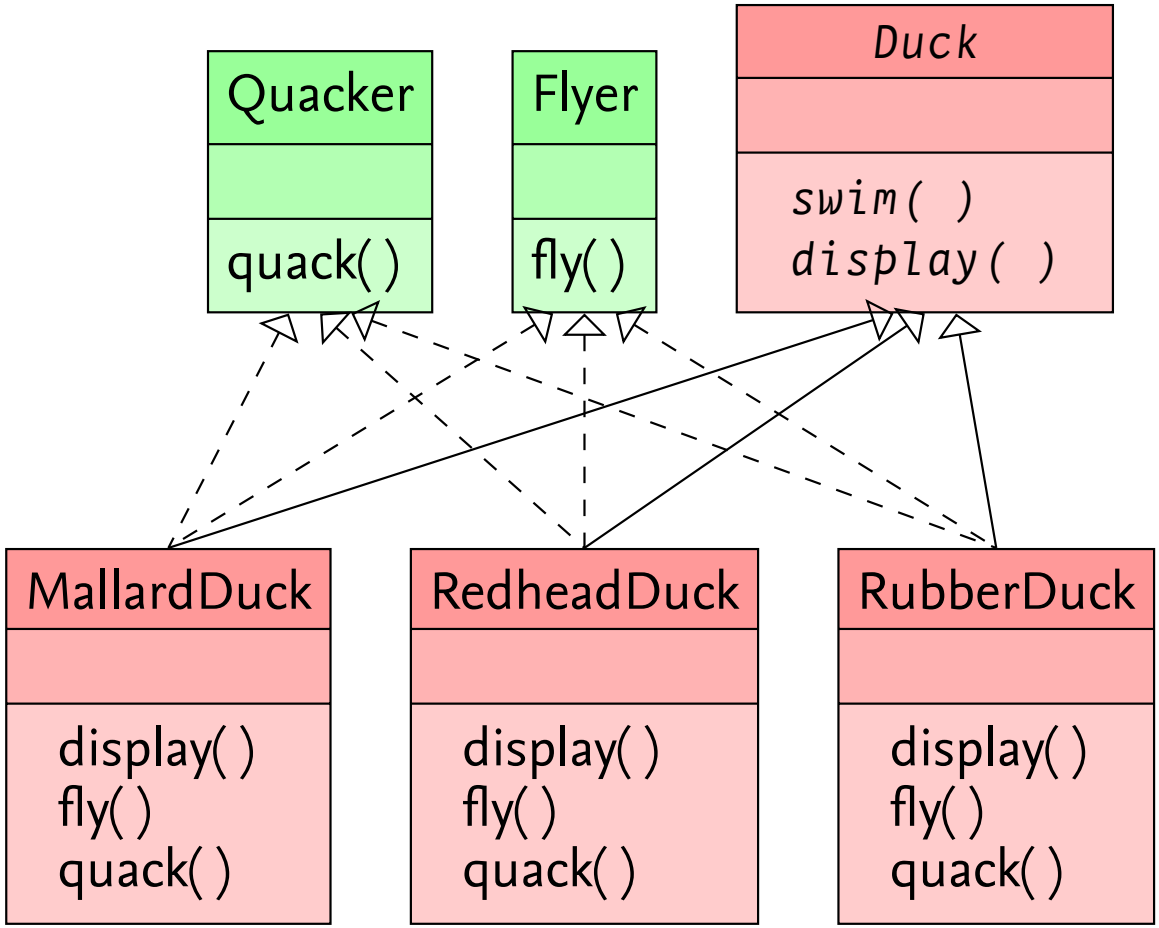
Should he override `fly( )` in the `RubberDuck` Class?

What Should Joe Do?

Should he override `fly( )` in the `RubberDuck` Class?

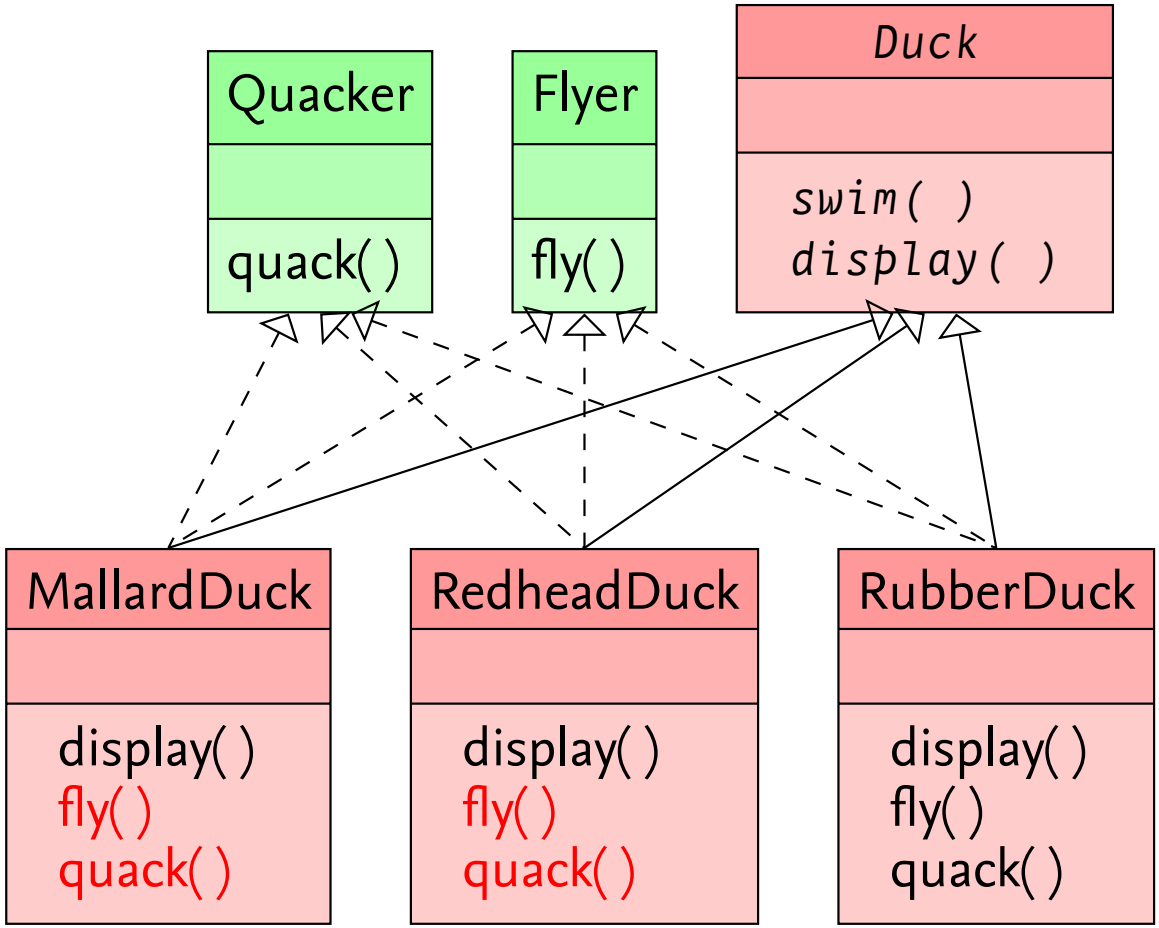
- If he did that he might have to duplicate code later.
  - For example, what if a `WoodenDecoyDuck` was added later?
  - `RubberDuck` and `WoodenDecoyDuck` were almost the same,
    - Yet shared no code....
- Of course he could introduce a common superclass.
  - But that would mean much work.
  - Also there was no guarantee that work would stop there.

Should he Use Interfaces?



Should he Use Interfaces?

Did Somebody say Code Duplication?



What Joe Really Wants

- Joe really wants software that doesn't change.
- He does realise that change is the only constant.
- Code changes should have little impact on existing code.
- That would save much time rewriting existing code.

First Design Principle

Encapsulate what Varies

- We've seen that inheritance didn't work for Joe.
  - When the (`Duck`) superclass changes this affects all subclasses.
- Interfaces cannot change but they have no implementation:
  - No code reuse.
- Encapsulate what Varies: Identify the aspects of your application that vary and separate them from what stays the same.
- We implement each aspect as a behaviour:
  - Implement separate classes for different behaviour.
  - Lets us choose specific behaviour by selecting a specific class.
  - Reusing the implementation comes for free.
  - Separates the implementation: increases flexability.

Encapsulate what Varies

- With interfaces, all classes *implemented* `Flyer` and `Quacker`.
- This is what caused the code duplication.
- We're going to encapsulate what varies:
  - We separate what varies: `fly( )` and `quack( )` behaviour.
  - We define a `Flyer` interface.
    - Encapsulate each different `fly( )` behaviour as separate class.
  - We also define a `Quacker` interface.
    - Encapsulate each different `quack( )` behaviour as separate class.
  - We reuse the behaviour in the actual `Duck` subclasses.
    - This is done using *delegation*.
    - (It involves a design pattern.)

Second Design Principle

- We need to design classes that implement  behaviour.
- Behaviour is *assigned* to specific `Duck` instance attributes.
  - Assiging behaviour can even be done at runtime.
- Program to an interface, not to an implementation.

Second Design Principle

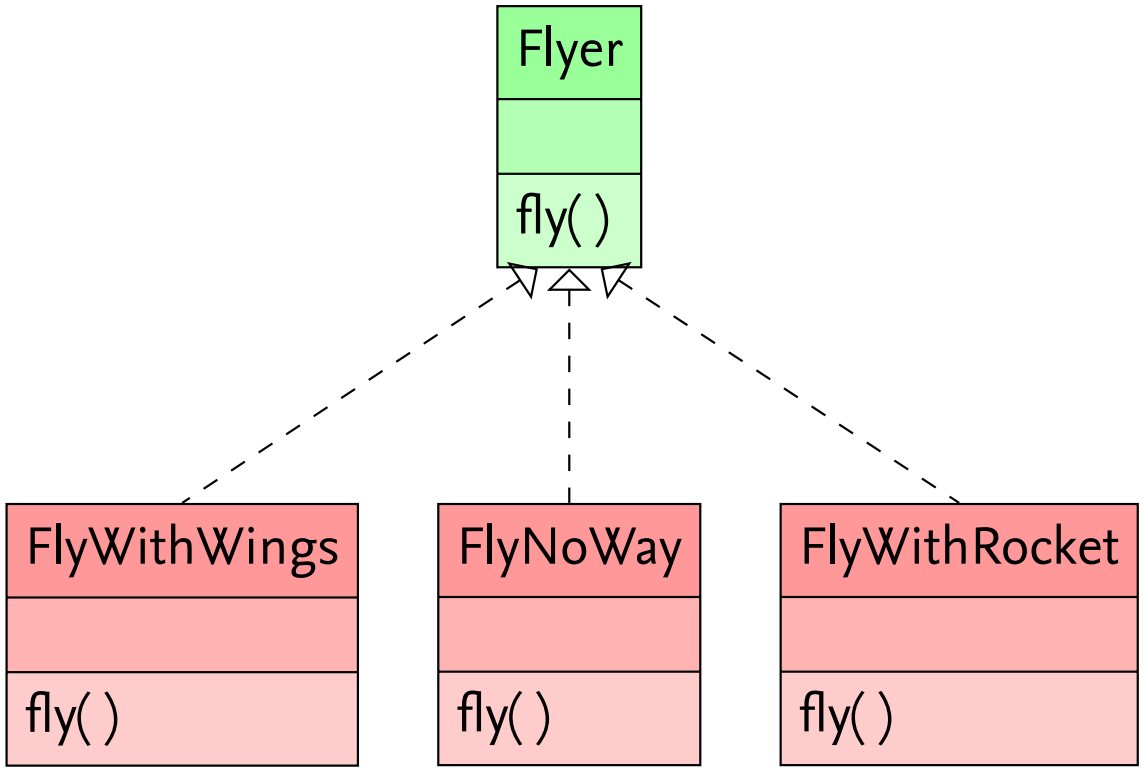
Program to an Interface

- We need to design classes that implement  behaviour.
- Behaviour is *assigned* to specific `Duck` instance attributes.
  - Assiging behaviour can even be done at runtime.
- Program to an interface, not to an implementation.

Programming to an Interface

- We use an interface (a supertype) for each behaviour.
  - `Flyer`, `Quacker`, ...
  - Specific classes implement specific behaviours.
  - We use instances of these classes to use the behaviour.
- Before we depended on an *implementation:*
  - Default or overridden class behaviour.
- Now we depend on an interface:
  - An object with a type.
- Clients are now *unaware* of actual type and class of object.
  - This greatly reduces subsystem dependencies.

Implementing the `fly( )` Behaviour



Integrating the Duck Behaviour

Each Duck Delegates the fly( ) and quack( ) Behaviour

Duck
private Flyer flyer private Quacker quacker
public final fly( ) { flyer.fly( ); } public final quack( ) { quacker.quack( ); } public swim( ) public display( )

Implementing the MallardDuck

Java

```
public class MallardDuck extends Duck {
    public MallardDuck( ) {
        super( new SqueekQuack( ), new FlyWithWings( ) );
    }

    @Override
    public void display( ) {
        System.out.println( "MallardDuck here...." );
    }
}
```

Implementing the MutableDuck

Java

```
public class MutableDuck extends Duck {
    public MutableDuck( ) {
        super( new SqueekQuack( ), new FlyWithWings( ) );
    }

    public void setQuackBehaviour( Quacker quacker ) {
        // Assumes quacker is public/not final now.
        this.quacker = quacker;
    }

    public void setFlyBehaviour( Flyer flyer ) {
        // Assumes flyer is public/not final now.
        this.flyer = flyer;
    }

    @Override
    public void display( ) {
        System.out.println( "MutableDuck here...." );
    }
}
```

Inheritance versus Object Composition

Inheritance: Lets us create subclasses: *white-box reuse*.

- Subclass inherits superclass behaviour.
- Subclasses can override superclass behaviour.
- You get code reuse for free.
- You cannot change behaviour at runtime.
- Violates encapsulation.
  - Subclass may rely on superclass implementation.
  - Subclass may break when superclass is changed.

Composition: Lets you *compose* classes: *black-box reuse*.

- A client class may use an object.
- You get code reuse but it takes more effort.
- Lets you change behaviour at runtime.
- Respects encapsulation.
  - Helps encapsulated classes focus on a single task.

Has-A can be better than Is-A

- In our new design we rely on Has-A (more then on Is-A):
  - Each `Duck` has-a `flyer`, and
  - Each `Duck` has-a `quacker`.
- "Has-A" lets us implement behaviour by *composing* classes.
- The result is a more flexible design:
  - It lets us encapsulate behaviour.
  - We can change behaviour at runtime.

Third Design Principle

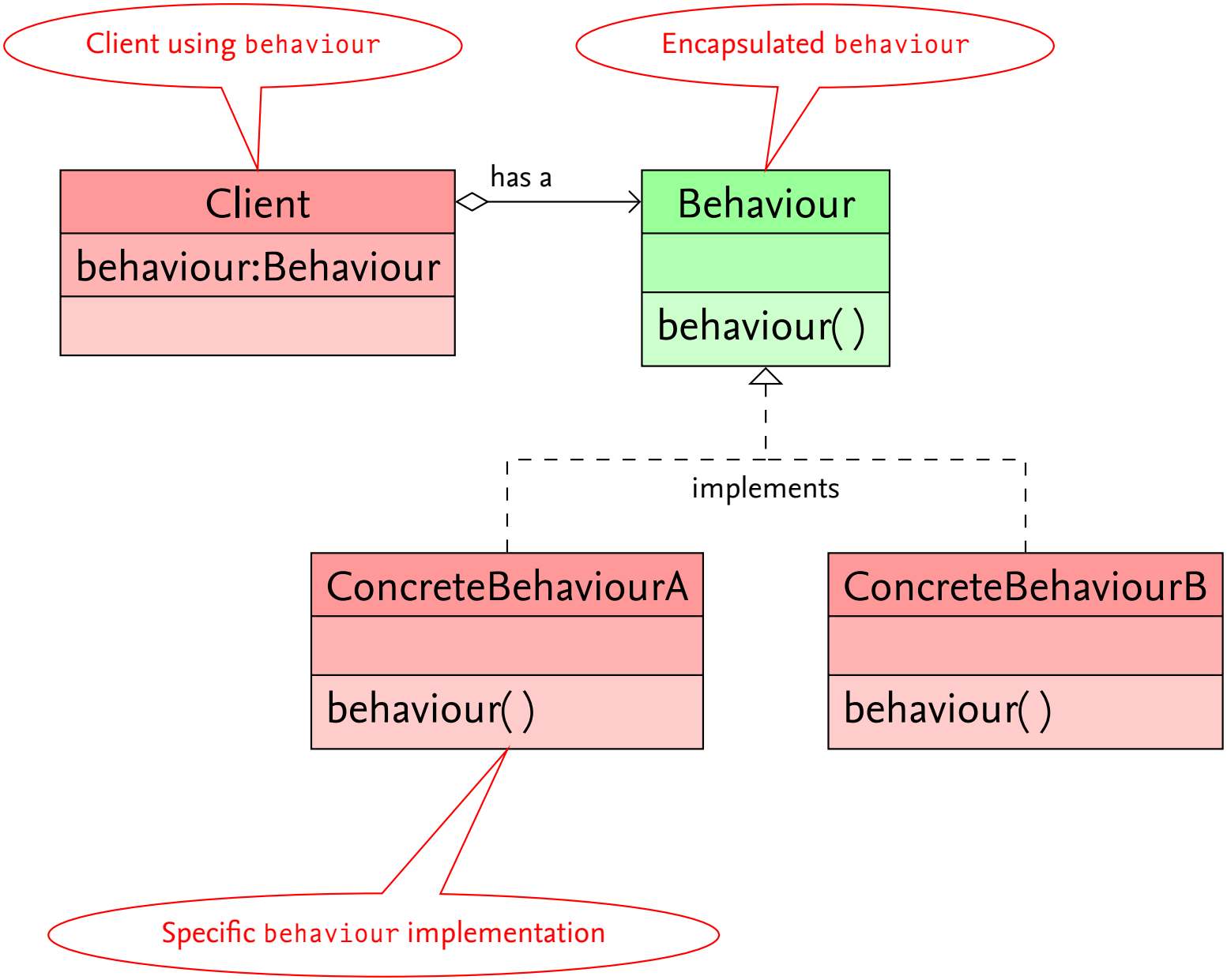
- In our new design we rely on Has-A (more then on Is-A):
  - Each `Duck` has-a `flyer`, and
  - Each `Duck` has-a `quacker`.
- "Has-A" lets us implement behaviour by *composing* classes.
- The result is a more flexible design:
  - It lets us encapsulate behaviour.
  - We can change behaviour at runtime.

Favour Composition over Inheritance.

The Strategy Pattern

- The *Strategy Pattern:*
  - Defines a class of algorithms;
  - Encapsulates each algorithm; and
  - Makes them interchangeable.
- Lets the algorithms vary independently from clients using it [**Gamma:et:al:2008**].

Finally: Strategy Pattern



Question Time

Questions Anybody?

For Friday

- Study the presentation.
- Study Chapter 8 from the book.

Acknowledgements

- This lecture is partially based on
  - [**Head:First:Java**].
  - [**Head:First:Design:Patterns**].

Bibliography I

About this Document

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