Software Development (CS2500)

Lecture 25: Inheritance (Continued)

M. R. C. van Dongen

November 20, 2013

Outline

- We continue studying inheritance.
- Study some class-design tests that help:
  - Determining sub- and superclass relationship;
  - Determining what attributes to use.
- Continue our study of abstract classes.
- Decide class membership with `instanceof`.
- Learn how to override some common methods:
  - `toString( );`
  - `equals( );`
- Learn how to *control* inheritance.

A Test for Inheritance

- Designing a class hierarchy is an art, more than a science.
- It may be difficult to get things right from the start.
  - What classes should you use?
  - Which class should go to top, middle, and bottom?
- The *is-a test* provides some help to catch early mistakes.
- If 'every *A* *is-a* *B*' then *A* can be a subclass of *B*.

Examples

- Every `Dog` is-an `Animal`?

Examples

- Every `Dog` is-an `Animal`? ($\surd$)
  - Yes, so `Dog` can be a subclass of `Animal`.

Examples

- Every `Dog` is-an `Animal`? ($\surd$)
  - Yes, so `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`?

Examples

- Every `Dog` is-an `Animal`? ($\surd$)
  - Yes, so `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`?
  - No, so `Animal` cannot be a subclass of `Dog`.

Examples

- Every `Dog` is-an `Animal`? ($\surd$)
  - Yes, so `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`?
  - No, so `Animal` cannot be a subclass of `Dog`.
- Every `Apple` is-a `Pear`?

Examples

- Every `Dog` is-an `Animal`? ($\surd$)
  - Yes, so `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`?
  - No, so `Animal` cannot be a subclass of `Dog`.
- Every `Apple` is-a `Pear`?
  - No, so `Apple` cannot be a subclass of `Pear`.

Examples

- Every `Dog` is-an `Animal`? ($\surd$)
  - Yes, so `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`?
  - No, so `Animal` cannot be a subclass of `Dog`.
- Every `Apple` is-a `Pear`?
  - No, so `Apple` cannot be a subclass of `Pear`.
- Every `Pear` is-an `Apple`?

Examples

- Every `Dog` is-an `Animal`? ($\surd$)
  - Yes, so `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`?
  - No, so `Animal` cannot be a subclass of `Dog`.
- Every `Apple` is-a `Pear`?
  - No, so `Apple` cannot be a subclass of `Pear`.
- Every `Pear` is-an `Apple`?
  - No, so `Pear` also cannot be a subclass of `Apple`.

Examples

- Every `Dog` is-an `Animal`? ($\surd$)
  - Yes, so `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`?
  - No, so `Animal` cannot be a subclass of `Dog`.
- Every `Apple` is-a `Pear`?
  - No, so `Apple` cannot be a subclass of `Pear`.
- Every `Pear` is-an `Apple`?
  - No, so `Pear` also cannot be a subclass of `Apple`.
- Every `Cat` is-a `Feline`?

Examples

- Every `Dog` is-an `Animal`? ($\surd$)
  - Yes, so `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`?
  - No, so `Animal` cannot be a subclass of `Dog`.
- Every `Apple` is-a `Pear`?
  - No, so `Apple` cannot be a subclass of `Pear`.
- Every `Pear` is-an `Apple`?
  - No, so `Pear` also cannot be a subclass of `Apple`.
- Every `Cat` is-a `Feline`? ($\surd$)
  - Yes, so `Cat` can be a subclass of `Feline`.

Examples

- Every `Dog` is-an `Animal`? ($\surd$)
  - Yes, so `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`?
  - No, so `Animal` cannot be a subclass of `Dog`.
- Every `Apple` is-a `Pear`?
  - No, so `Apple` cannot be a subclass of `Pear`.
- Every `Pear` is-an `Apple`?
  - No, so `Pear` also cannot be a subclass of `Apple`.
- Every `Cat` is-a `Feline`? ($\surd$)
  - Yes, so `Cat` can be a subclass of `Feline`.
- Every `Feline` is-a `Cat`?

Examples

- Every `Dog` is-an `Animal`? ($\surd$)
  - Yes, so `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`?
  - No, so `Animal` cannot be a subclass of `Dog`.
- Every `Apple` is-a `Pear`?
  - No, so `Apple` cannot be a subclass of `Pear`.
- Every `Pear` is-an `Apple`?
  - No, so `Pear` also cannot be a subclass of `Apple`.
- Every `Cat` is-a `Feline`? ($\surd$)
  - Yes, so `Cat` can be a subclass of `Feline`.
- Every `Feline` is-a `Cat`?
  - No, so `Feline` cannot be a subclass of `Cat`.

Other ‘Tests’

The ‘extends test’ is not so robust:

- `Cat` extends `Feline`.

Other ‘Tests’

The ‘extends test’ is not so robust:

- `Cat` extends `Feline`.
  - So `Cat` can be a subclass of `Feline`.

Other ‘Tests’

The ‘extends test’ is not so robust:

- `Cat` extends `Feline`.
  - So `Cat` can be a subclass of `Feline`.
- `Feline` extends `Cat`.

Other ‘Tests’

The ‘extends test’ is not so robust:

- `Cat` extends `Feline`.
  - So `Cat` can be a subclass of `Feline`.
- `Feline` extends `Cat`.
  - No, so `Feline` cannot be a subclass of `Cat`.

Other ‘Tests’

The ‘extends test’ is not so robust:

- `Cat` extends `Feline`.
  - So `Cat` can be a subclass of `Feline`.
- `Feline` extends `Cat`.
  - No, so `Feline` cannot be a subclass of `Cat`.
- `Conservatory` (`Sunroom`) extends `House`.

Other 'Tests'

The 'extends test' is not so robust:

- `Cat` extends `Feline`.
  - So `Cat` can be a subclass of `Feline`.
- `Feline` extends `Cat`.
  - No, so `Feline` cannot be a subclass of `Cat`.
- `Conservatory` (`Sunroom`) extends `House`.
  - Yes, but `Conservatory` cannot be a subclass of `House`.

Other 'Tests'

The 'extends test' is not so robust:

- `Cat` extends `Feline`.
  - So `Cat` can be a subclass of `Feline`.
- `Feline` extends `Cat`.
  - No, so `Feline` cannot be a subclass of `Cat`.
- `Conservatory` (`Sunroom`) extends `House`.
  - Yes, but `Conservatory` cannot be a subclass of `House`.
  - For example:
    - If `Conservatory` extends `House` then it inherits all `House` methods:
    - `Conservatory.ringDoorBell( )`????
    - `Conservatory.lightFireplace( )`????

An Association Test

- The `House` uses/requires/has access to the `Conservatory`.
- Still `Conservatory` cannot extend `House`.
- However, it makes sense if `House` class has `Conservatory` attribute.

Java

```
public class House {
    private Bell doorBell;
    private Window[] groundfloorWindows;
    private Window[] firstFloorWindows;
    private Conservatory conservatory;
    …
}
```

An Association Test (Continued)

- `MouseCursor` cannot be a subclass of `Window`.
- But, makes sense if `Window` class has `MouseCursor` attribute.

Java

```
public class Window {
    private Position currentPosition;
    private Point lowerLeft;
    private Point upperRight;
    private MouseCursor cursor;
    …
}
```

An Association Test (Continued)

- If a class *A* has a class-*B* attribute then class *A* *uses* *B*.
  - `Window` uses a `MouseCursor`.
  - `House` uses a `Conservatory`.
- The *has-a* test determines when a class uses another class.
- If '*A* has-a *B*' then *A* can have a class-*B* attribute.

Examples

- Every `House` has-a `Conservatory` (possibly `null`).

Examples

- Every `House` has-a `Conservatory` (possibly `null`). ($\surd$)
  - So `House` should have a `Conservatory` attribute.

Examples

- Every `House` has-a `Conservatory` (possibly `null`). ($\surd$)
  - So `House` should have a `Conservatory` attribute.
- Every `Window` has-a `MouseCursor`.

Examples

- Every `House` has-a `Conservatory` (possibly `null`). ($\surd$)
  - So `House` should have a `Conservatory` attribute.
- Every `Window` has-a `MouseCursor`. ($\surd$)
  - So `Window` should have a `MouseCursor` attribute.

Examples

- Every `House` has-a `Conservatory` (possibly `null`). ($\surd$)
  - So `House` should have a `Conservatory` attribute.
- Every `Window` has-a `MouseCursor`. ($\surd$)
  - So `Window` should have a `MouseCursor` attribute.
- Every `Animal` has-a `Cat`.

Examples

- Every `House` has-a `Conservatory` (possibly `null`). ($\surd$)
  - So `House` should have a `Conservatory` attribute.
- Every `Window` has-a `MouseCursor`. ($\surd$)
  - So `Window` should have a `MouseCursor` attribute.
- Every `Animal` has-a `Cat`.
  - No, so `Animal` shouldn't have a `Cat` attribute.

Examples

- Every `House` has-a `Conservatory` (possibly `null`). ($\surd$)
  - So `House` should have a `Conservatory` attribute.
- Every `Window` has-a `MouseCursor`. ($\surd$)
  - So `Window` should have a `MouseCursor` attribute.
- Every `Animal` has-a `Cat`.
  - No, so `Animal` shouldn't have a `Cat` attribute.
- Every `Cat` has-an `Animal`.

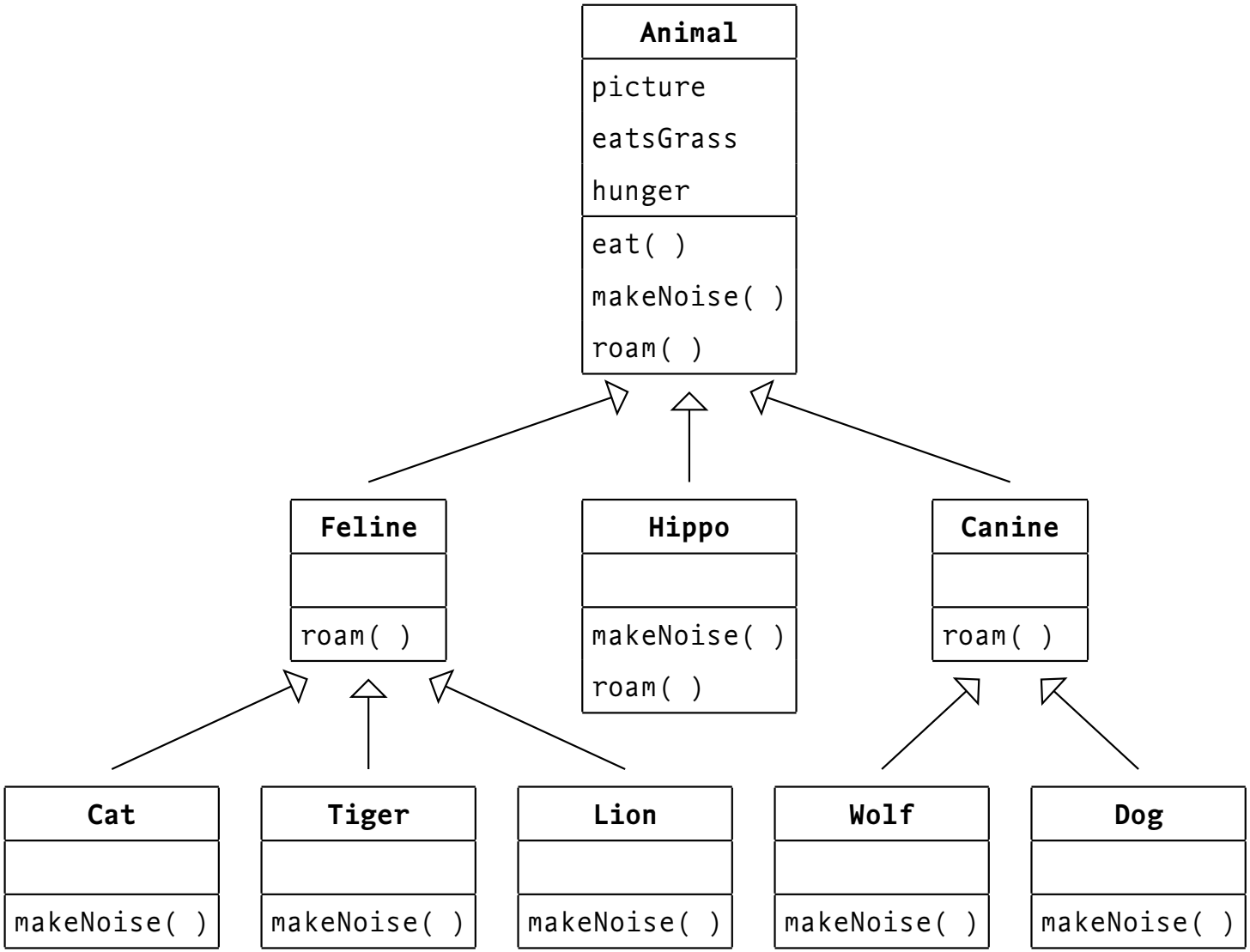
Examples

- Every `House` has-a `Conservatory` (possibly `null`). ($\surd$)
  - So `House` should have a `Conservatory` attribute.
- Every `Window` has-a `MouseCursor`. ($\surd$)
  - So `Window` should have a `MouseCursor` attribute.
- Every `Animal` has-a `Cat`.
  - No, so `Animal` shouldn't have a `Cat` attribute.
- Every `Cat` has-an `Animal`.
  - No, so `Cat` shouldn't have an `Animal` attribute.

Examples

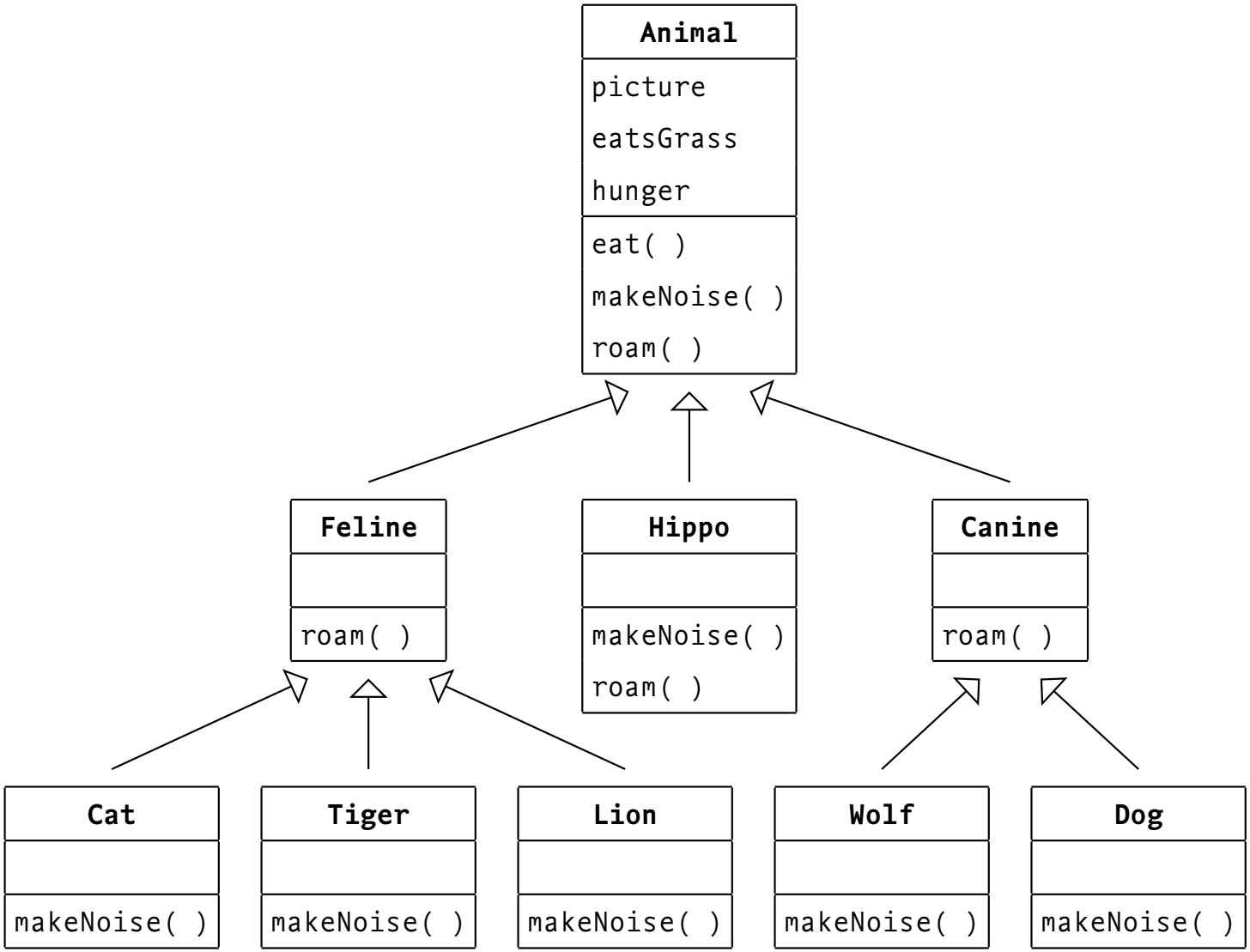
- Every `House` has-a `Conservatory` (possibly `null`). ($\surd$)
  - So `House` should have a `Conservatory` attribute.
- Every `Window` has-a `MouseCursor`. ($\surd$)
  - So `Window` should have a `MouseCursor` attribute.
- Every `Animal` has-a `Cat`.
  - No, so `Animal` shouldn't have a `Cat` attribute.
- Every `Cat` has-an `Animal`.
  - No, so `Cat` shouldn't have an `Animal` attribute.

Abstract Classes



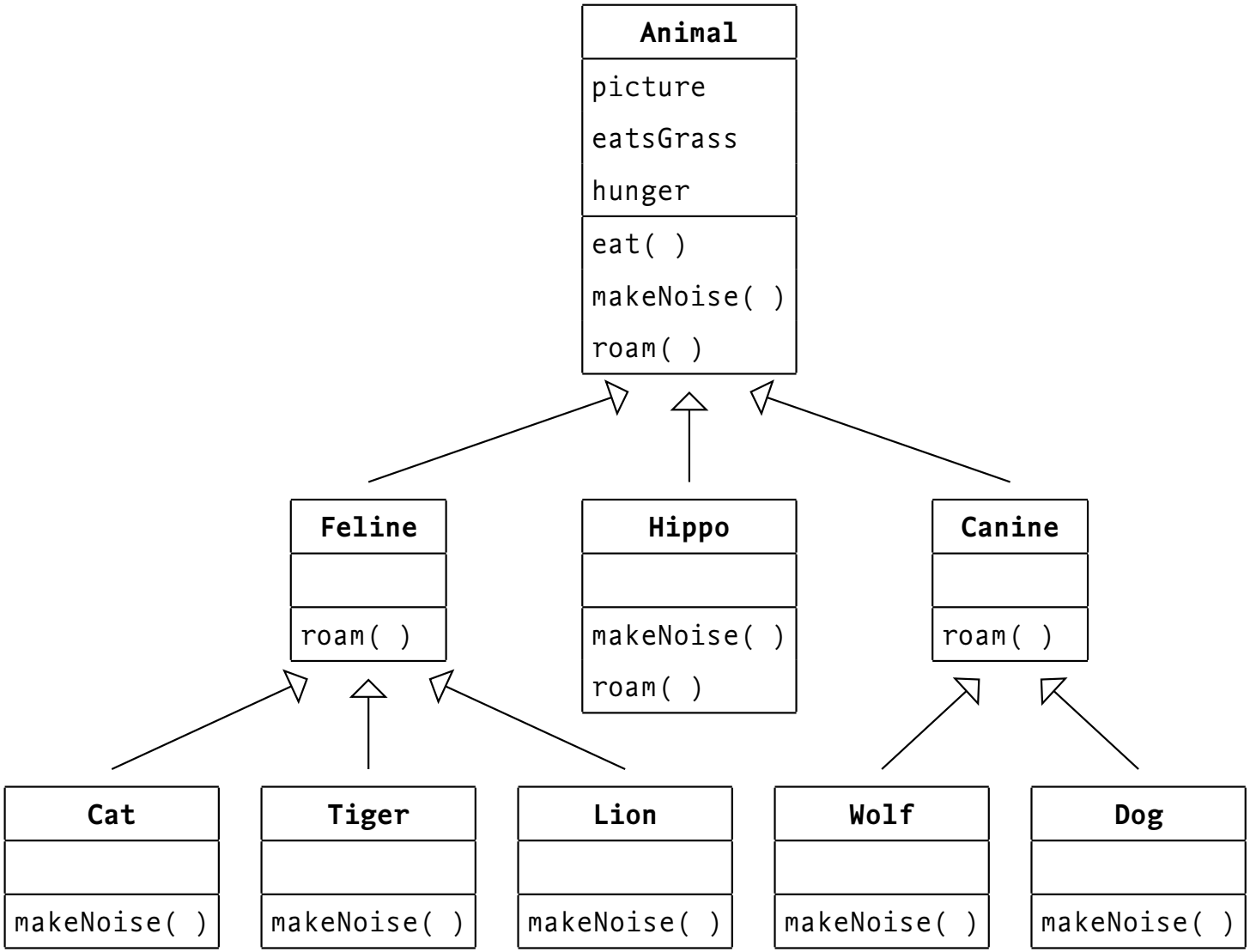
Abstract Classes

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



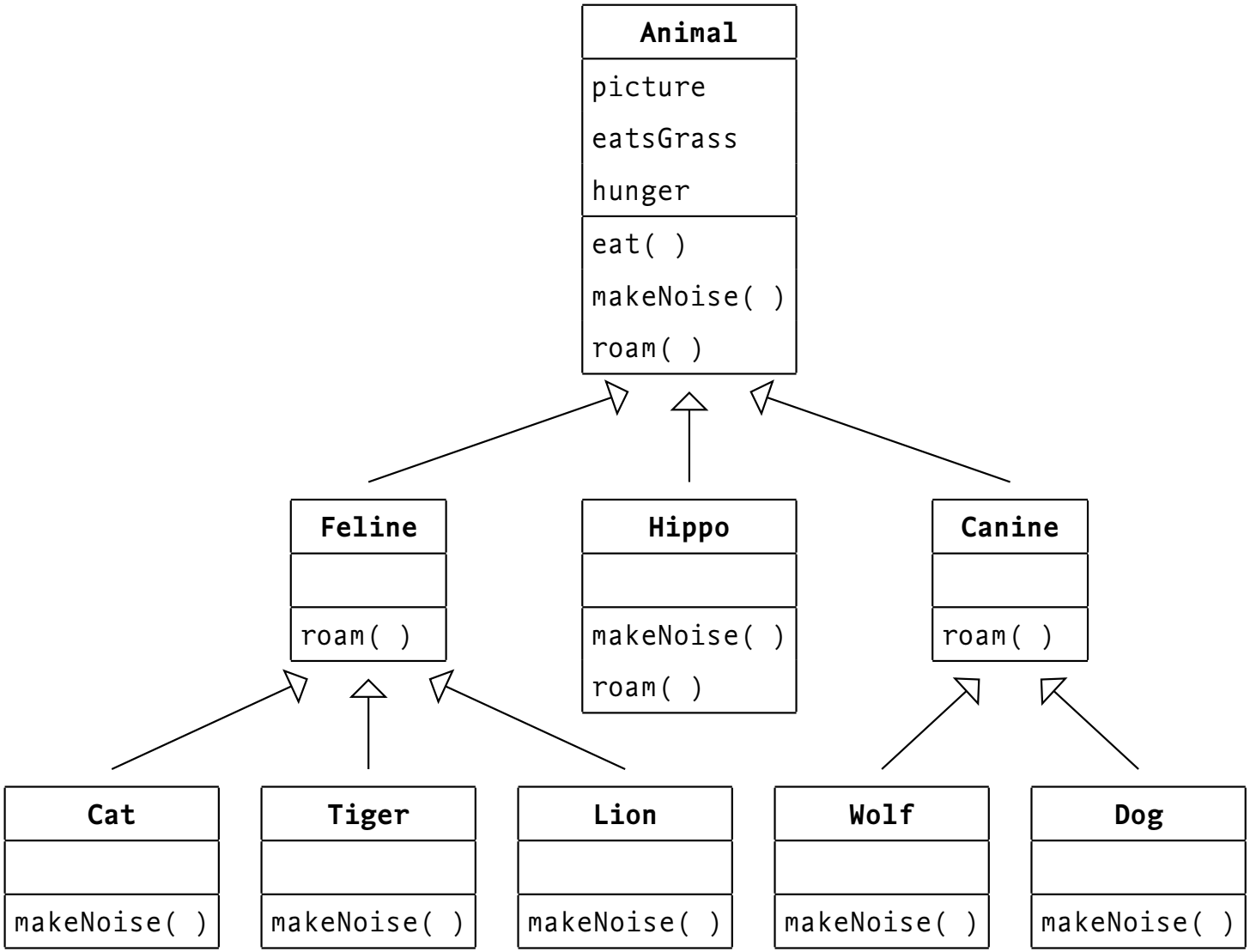
Abstract Classes

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



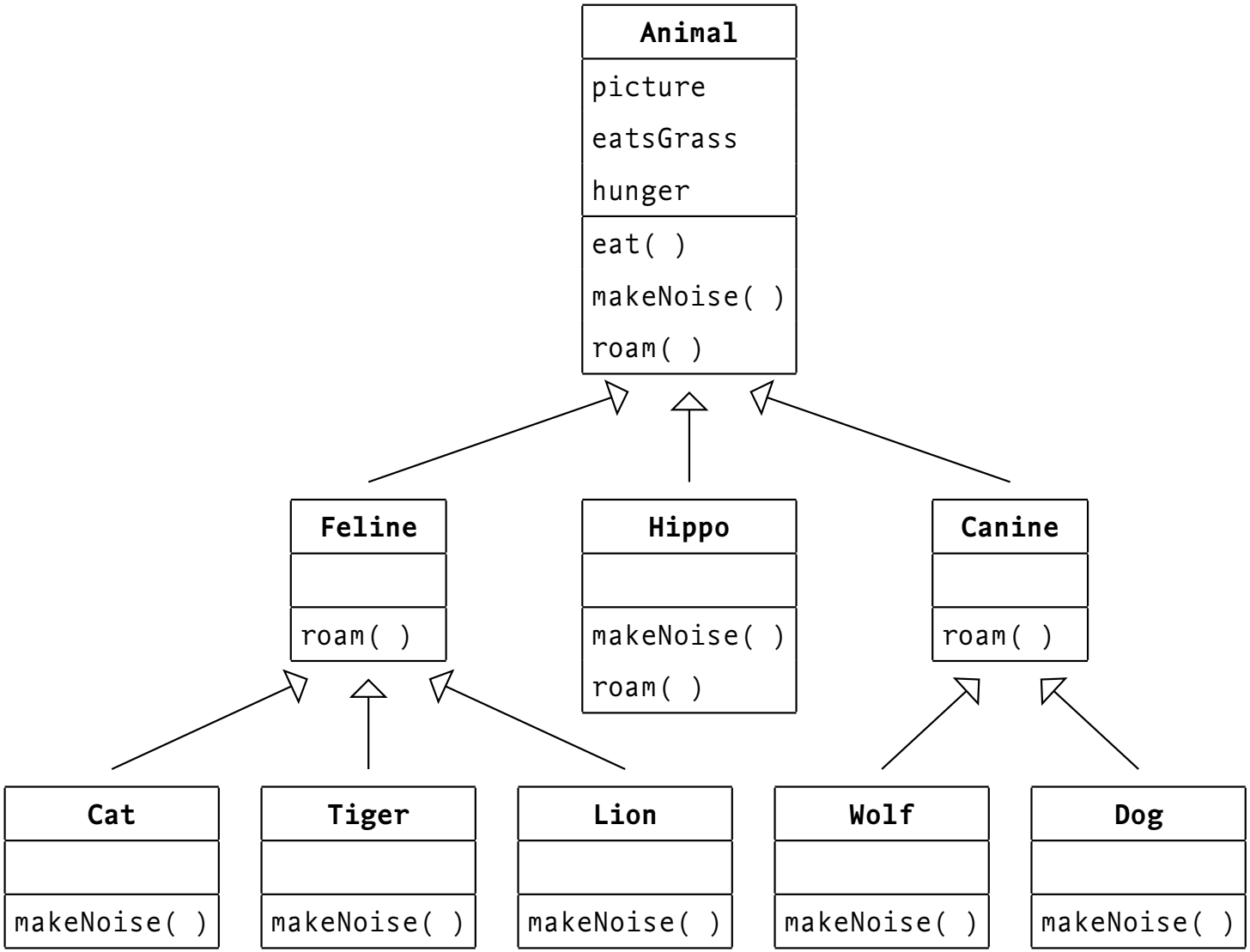
Abstract Classes

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



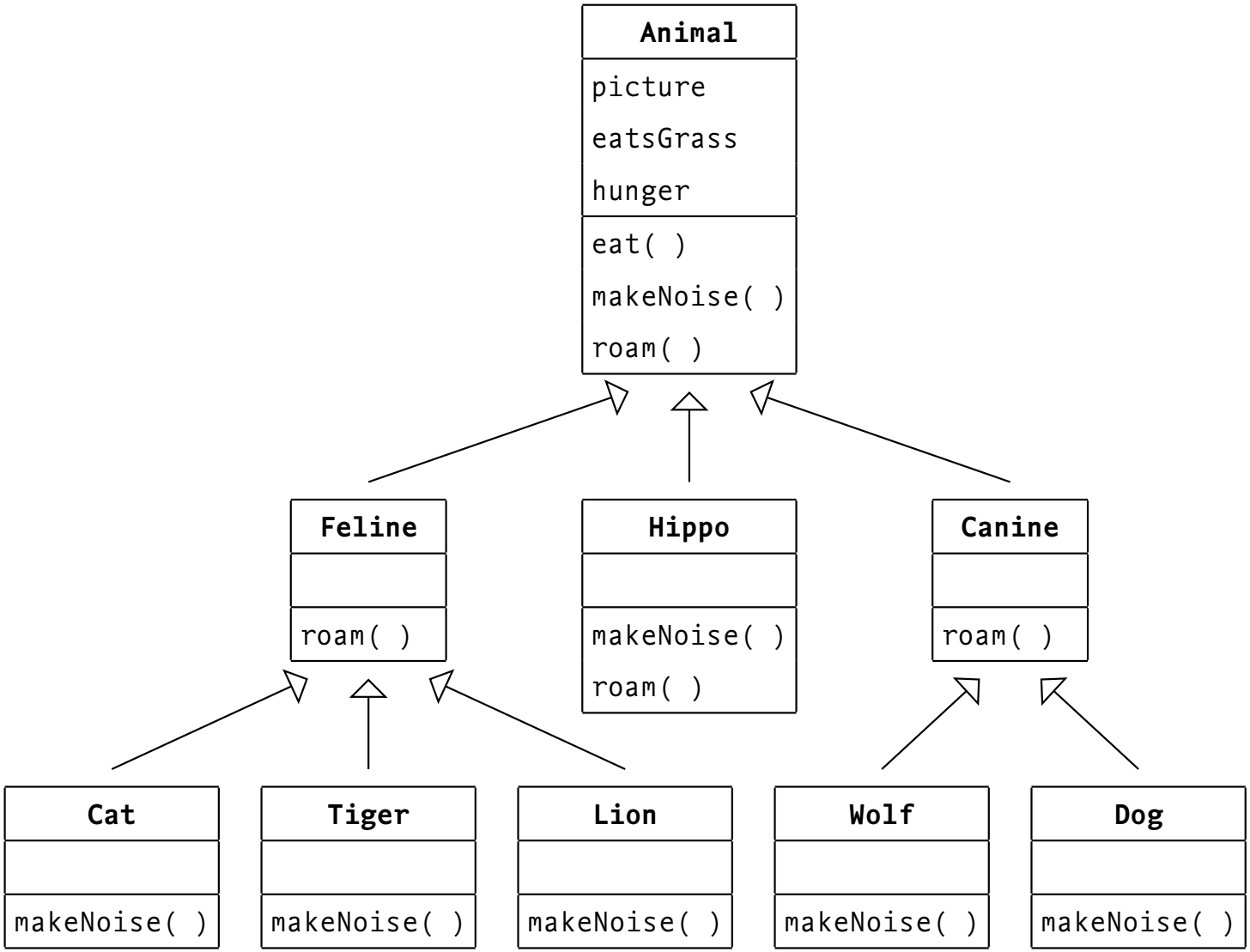
Abstract Classes

```
Cat cat = new Cat( );
```
√



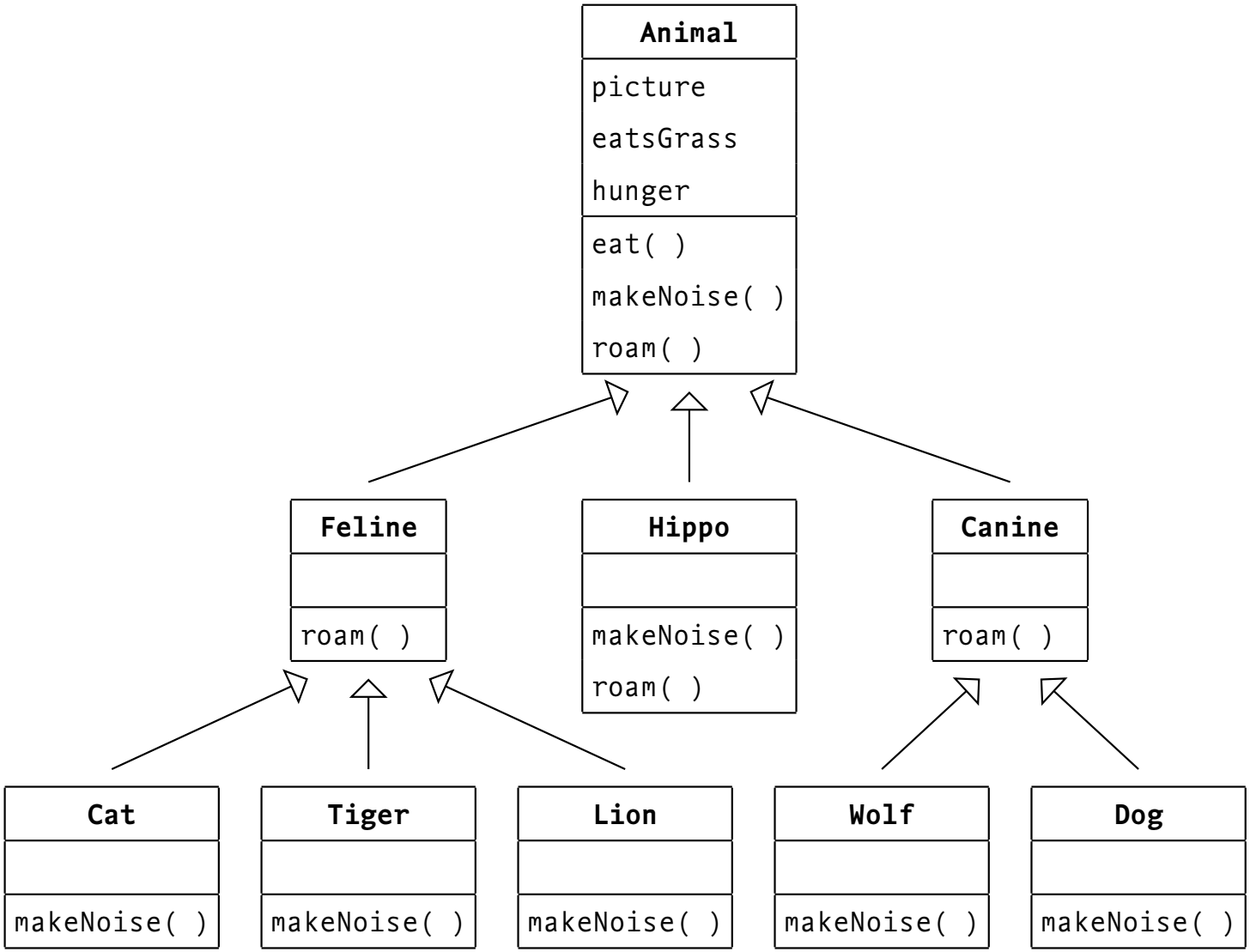
Abstract Classes

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



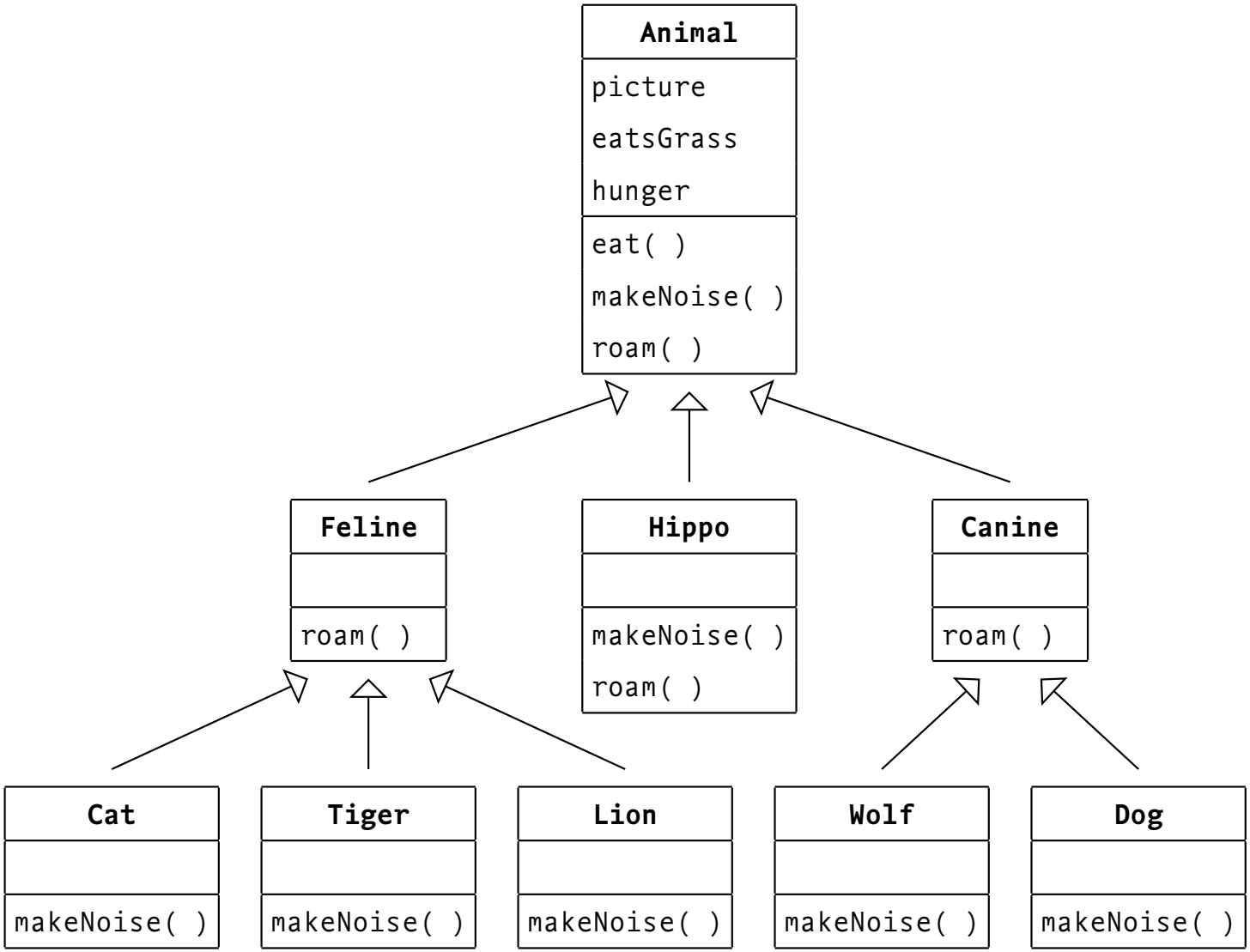
Abstract Classes

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



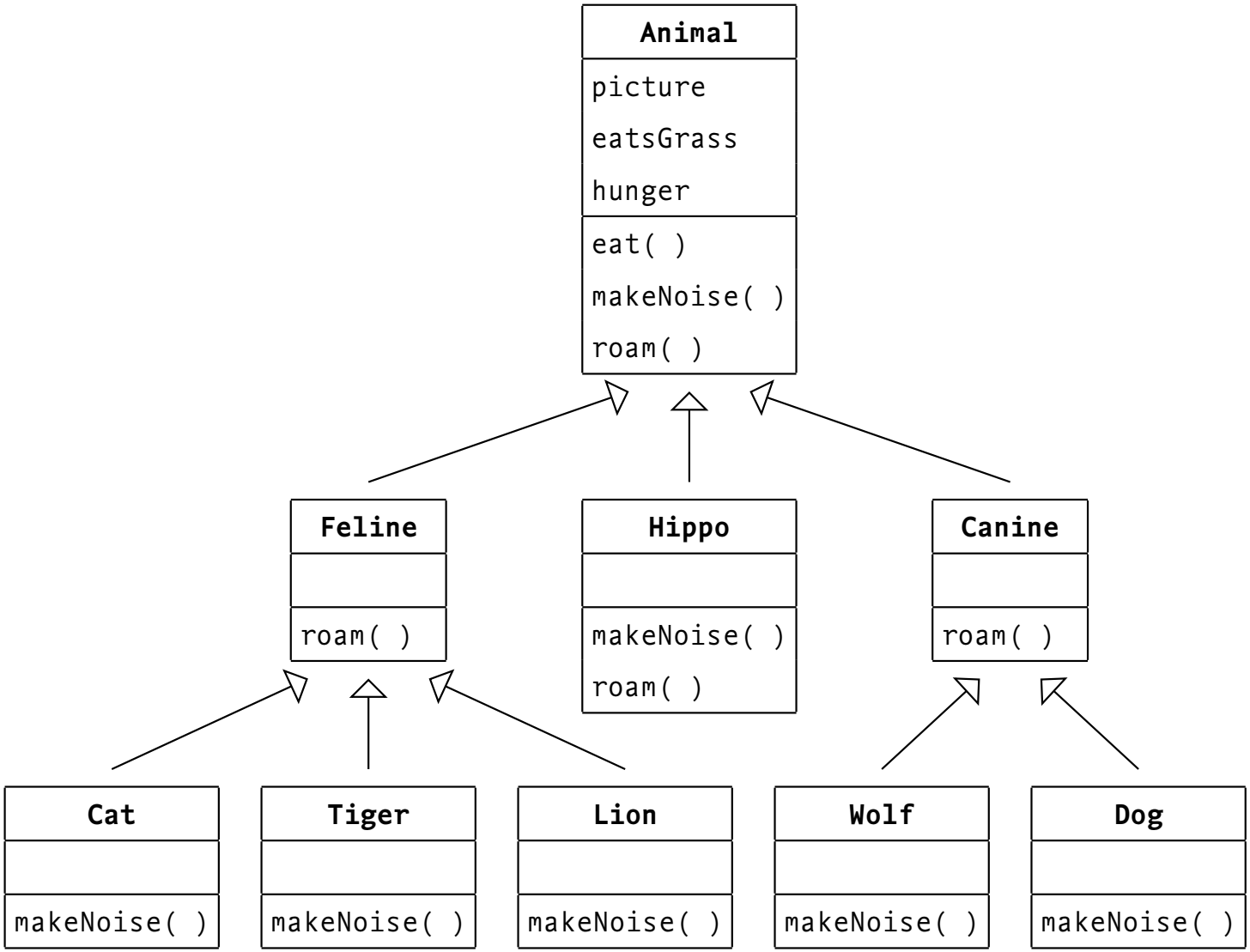
Abstract Classes

```
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 protocol 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 intended 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 fogetting 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 fogetting 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 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

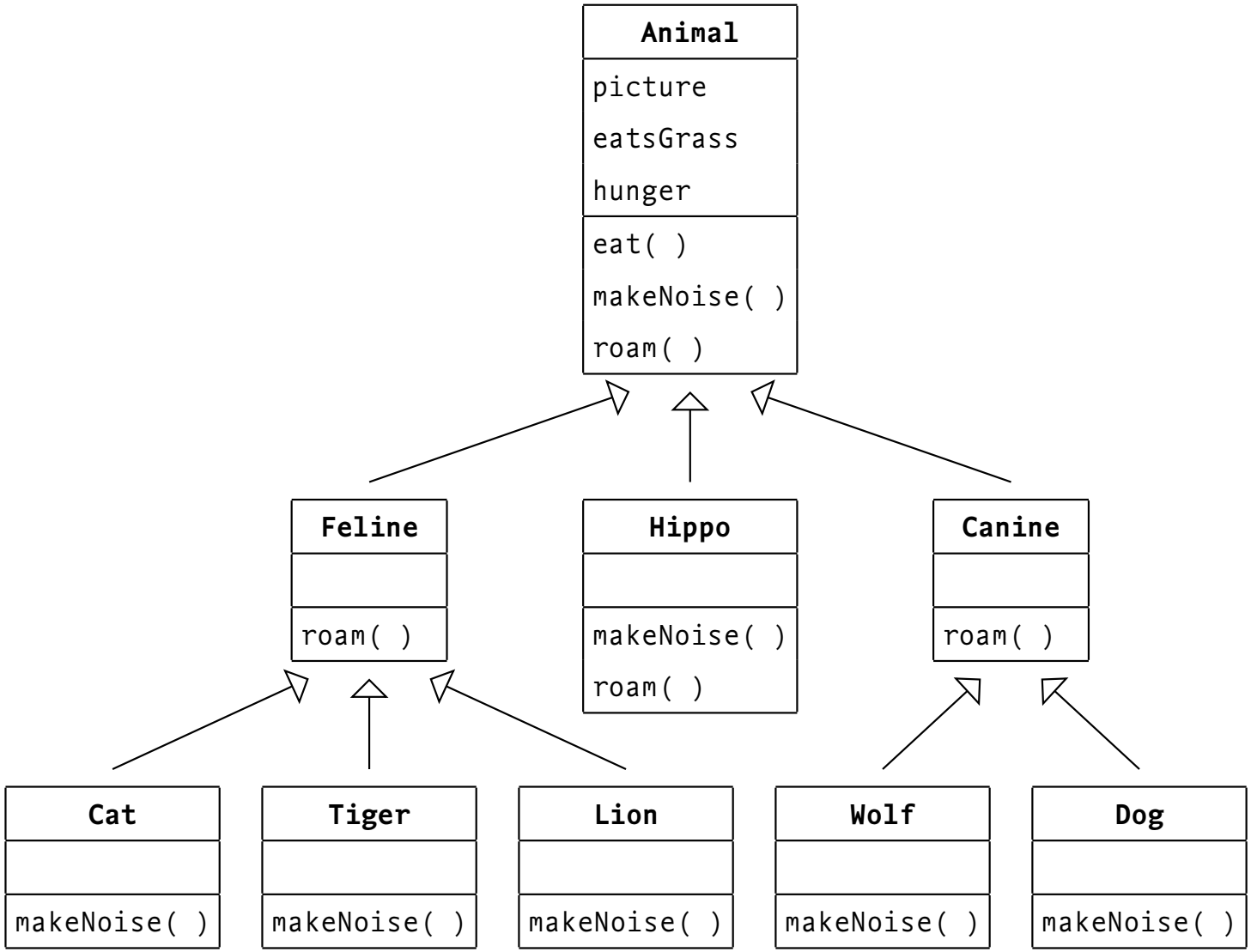
```
public abstract class Animal {
    public abstract void makeNoise( );
}
```

Java

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

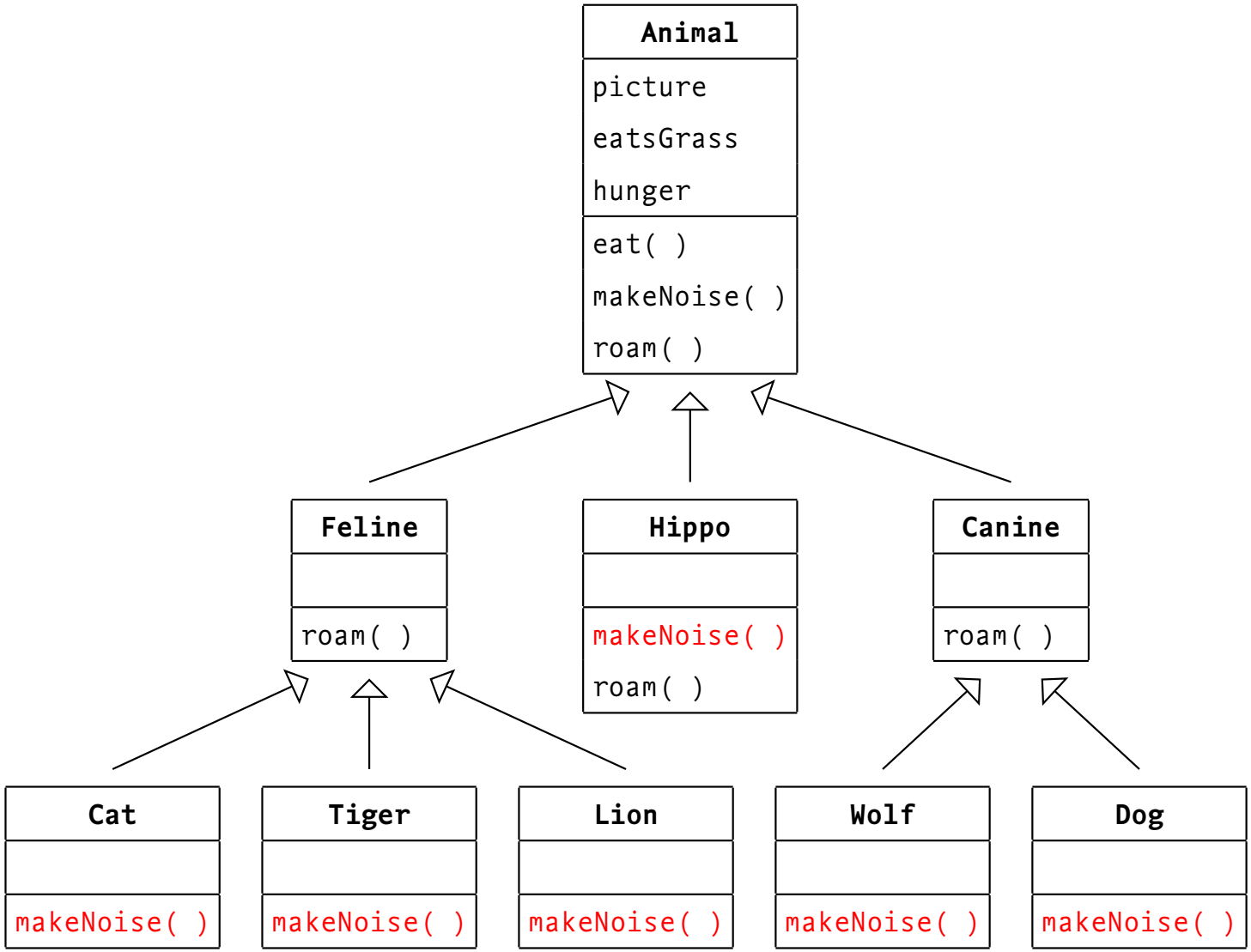
How does this Work?

Implementing the Method for all Relevant Classes



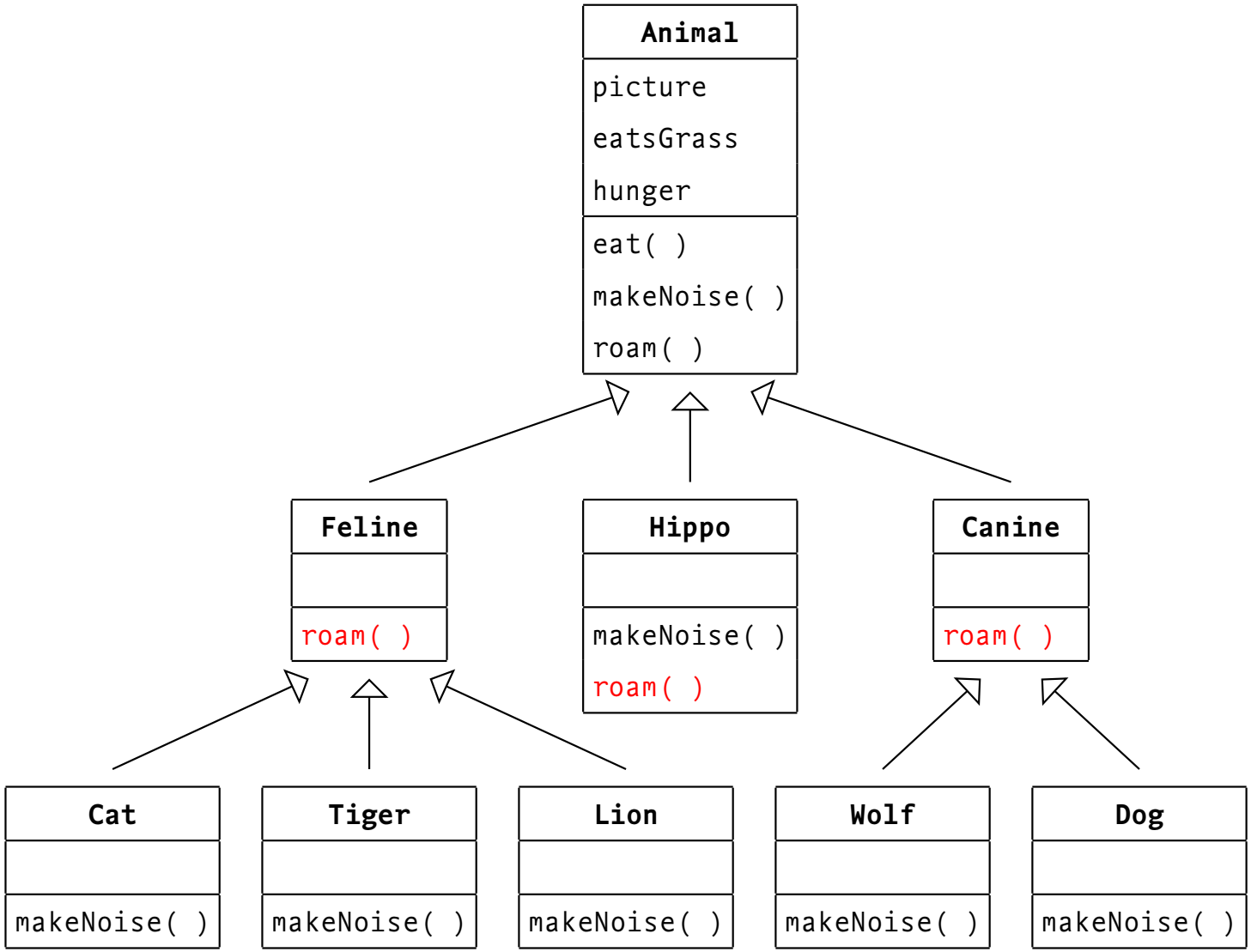
How does this Work?

Implementing the Method for all Relevant Classes



How does this Work?

Implementing the Method for all Relevant Classes



The Barber of Seville

- A barber is somebody that shaves people that don't shave themselves.

The Barber of Seville

- A barber is somebody that shaves people that don't shave themselves.
- 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.shaveMyself( );
        } 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 shaveMyself( ) {
        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.shaveMyself( );
        } else {
            final Barber barber = Barber.orderBarber( );
            barber.shave( person );
        }
    }
}
```

- More important applications a few slides further on.

Deciding Subclass Membership

- The text `reference instanceof Clazz` tests for class membership of `Clazz`.
- It returns:
  - `true` if `reference` is an instance of `Clazz`;
  - `true` if `reference` is an instance of a subclass of `Clazz`;
  - `false` otherwise.
- 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 instance method `public String toString( );`
- 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 Dice {
    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 URL url;
    ...

    // Not sure what to do really.
    @Override
    public String toString( ) {
        return "DatabaseConnection[ id = " + id
                                + ", db = " + db // ????
                                + ", url = " + url
                                + ... // all attributes
                                + " ]";
    }
}
```

How?

Depends on the Class

Java

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

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

public class DataBaseConnection implements Testable {
    private final Database db;
    private final long id;
    private final URL url;
    ...

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

Overriding `equals( )`

- `public boolean equals( Object object ):`
  - Defined in `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 ):`
  - Defined in `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. Make the class `final`:
     - By making a class `final` the class cannot be extended.
  2. Make a method `final`:
     - By making a method `final` the method cannot be overridden.

Making the Class Final

Java

```
public final class LastWord {
    // 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: Make 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`.

Maintenance: Clients may rely on own overridden behaviour.

Making the Method Final

Java

```
public class Example {
    // You aren't allowed to override this method.
    public final void finalMethod( ) { … }
    // You can override this method.
    public void overridableMethod( ) { … }
}
```

For Friday

- Study [Horstmann 2013, Sections 9.3–9.4].

Acknowledgements

- This lecture corresponds to [Horstmann 2013, Sections 9.3–9.4].
- Some material is based on [Sierra, and Bates 2004].

About this Document

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