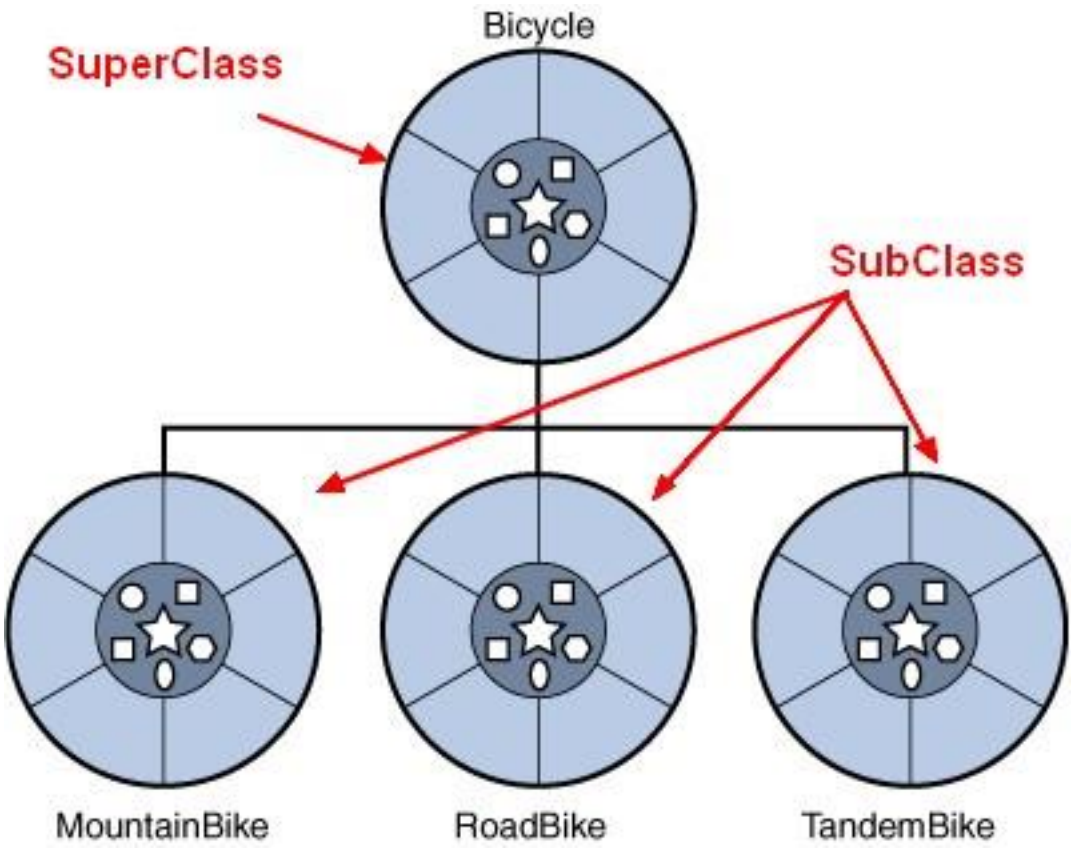
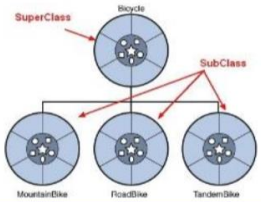
Java Class 23



9 Class 22 	10	11 Class 23 
16 Class 24 	17	18 Class 25 
23 Class 27 Quiz Review	24	25 Class 28 Quiz #3

Group Programming Opportunity (EC)

Party

30 Class 29 	1 PGM #3 due midnight	2 Class 30 PGM #3 Reviews	3	4 Quiz #4 (optional) 1:00 – 2:30 pm
7 Quiz #4 (optional) 10:30 am - noon	8	9	10	11

Introduction to Inheritance

Rather than writing a class that may have similar functionality to another class, it would be nice to be able to "clone" the existing class and make additions or modifications to the class

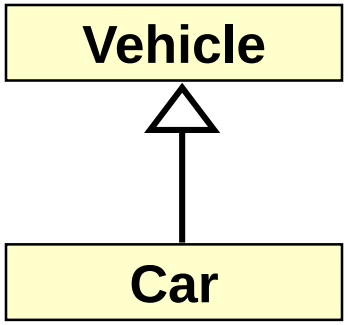
Once a class has be written and tested, it should ideally represent a reusable unit of code

But it takes experience and insight to produce reusable classes

Inheritance

Inheritance is the capability of a class to use the properties and methods of another class while adding its own functionality

Inheritance is a fundamental technique for organizing and creating classes



- Proper inheritance creates an *is-a* relationship, meaning the child class (Car) *is a* more specific version of the parent class (Vehicle)

Inheritance

A programmer can tailor a child class as needed by adding new variables or methods, or by modifying the inherited ones

One benefit of inheritance is *software reuse*

By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software

Deriving Subclasses

In Java, we use the reserved word `extends` to establish an inheritance relationship

When we extend an existing class, we inherit all of the variables and methods of the parent class

```
public class Car extends Vehicle
{
   // class contents
}
```

- A child is always a more specific version of the parent class (also referred to as the superclass or base class)

The `protected` Modifier

Visibility modifiers control access to members of a class

Any `public` method or variable in a parent class can be referenced by name in a child class and through objects of the child class

`public` variables violate the principle of encapsulation

`protected` visibility allows a derived class to reference a method or variable in the parent class

The `super` Reference

Constructors are not inherited, even though they have public visibility

The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor

A child's constructor is responsible for calling the parent's constructor (often automatically)

```
super(numPages);
```

Call parent's constructor with parameter numPages

The first line of a child's constructor should use the `super` reference to call the parent's constructor if parameters need to be passed

Overriding Methods and Shadow Variables

When a child defines a method with the same name and signature as a method in the parent class, we say the child's version overrides the parent's version

The need for overriding occurs in inheritance situations

A method defined with the final modifier cannot be overridden

A shadow variable is a variable in the child class with the same name as a variable in the parent class

Shadowing variables should be avoided

Overloading vs. Overriding

Overloading deals with multiple methods with the same name in the same class but with different signatures

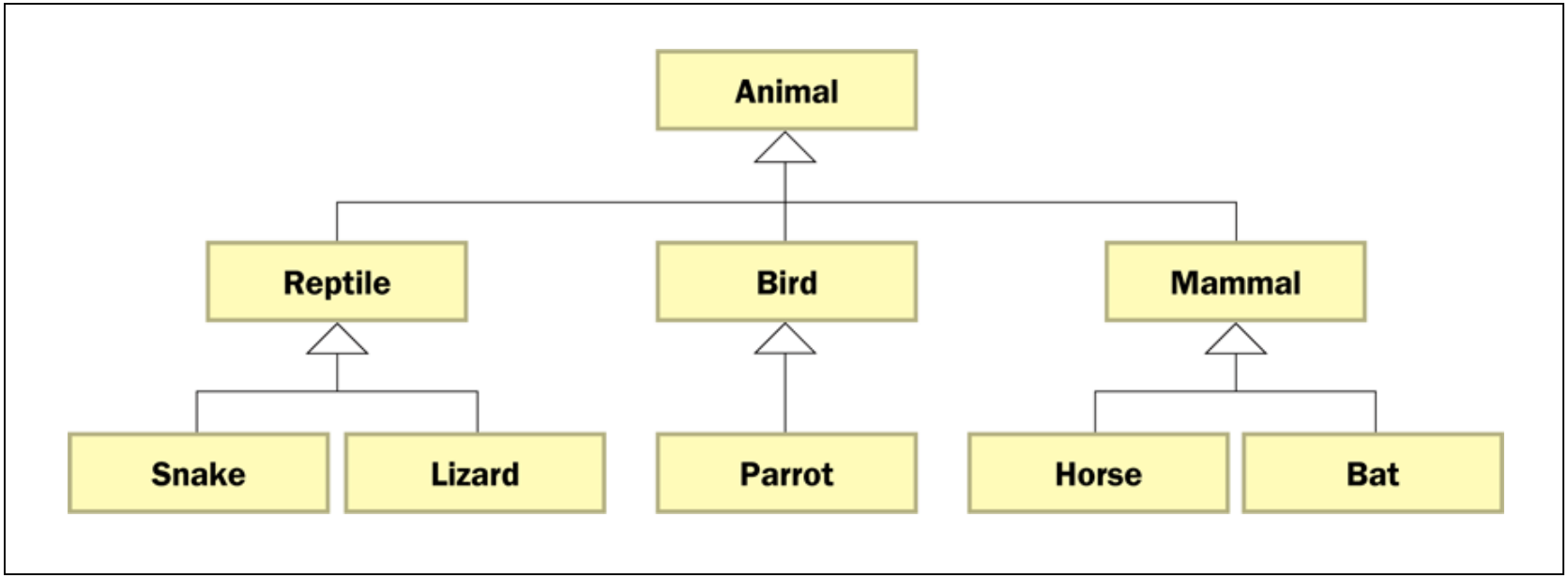
Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature

Overloading lets you define a similar operation in different ways for different parameters

Overriding lets you define a similar operation in different ways for different object types

Class Hierarchies

A child class of one parent can be the parent of another child, forming a *class hierarchy*



Class Hierarchies

Two children of the same parent are called *siblings*

Common features should be put as high in the hierarchy as is reasonable

An inherited member is passed continually down the line

Therefore, a child class inherits from all its ancestor classes

There is no single class hierarchy that is appropriate for all situations

The Object Class

A class called `Object` is defined in the `java.lang` package of the Java standard class library

All classes are derived from the `Object` class

If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class

Therefore, the `Object` class is the ultimate root of all class hierarchies

For example, the `toString` method is defined in the `Object` class

Abstract Classes

An *abstract class* is a placeholder in a class hierarchy that represents a generic concept

An abstract class cannot be instantiated and usually has one or more abstract methods

An abstract class typically contains a partial description that is inherited by all of its descendants in the class hierarchy; its children, which are more specific, fill in the gaps

Any class that contains one or more abstract methods must be declared as abstract

We use the modifier `abstract` on the class header to declare a class as abstract:

```
public abstract class Product
```

The `protected` Modifier

Visibility modifiers affect the way that class members in a superclass can be inherited in a subclass

`private` members (instance variables and methods) are not inherited (a mistake in the textbook!)

`public` members are inherited -- but `public` instance variables violate the principle of encapsulation (they are accessible by using the dot operator)

`protected` visibility allows a subclass to inherit a member in the superclass and still provides some encapsulation properties

A member declared with `protected` can be accessed (via dot operator) by any class in the same package, in addition to being accessible directly in a subclass

Each inherited member retains the original visibility modifier from the superclass

How About Constructor?

Unlike members (instance variables and methods), constructors are not inherited, even though they have public visibility

However, when a subclass constructor runs, it immediately calls its superclass constructor (for an object to be fully-formed, all the superclass parts must be fully-formed first)

```
public class Animal {
   public Animal() {
      System.out.println("Making an Animal");
   }
}
```

```
public class Hippo extends Animal {
   public Hippo() {
      System.out.println("Making a Hippo");
   }
}
```

```
public class TestHippo {
   public static void main (String[] args) {
      System.out.println("Starting...");
      Hippo h = new Hippo();
   }
}
```

Output

```
Starting…
Making an Animal
Making a Hippo
```

The Hippo constructor is invoked first, but it is the Animal constructor finishes first.

The `super` Reference

We often want to use the parent's constructor to set up the "parent's part" of the object (e.g., `pages` in `Book` class)

The `super` reference in the subclass can be used to refer to the superclass, and often is used to invoke the parent's constructor if you want to pass parameters to the parent's constructor.

If you want to do this, the first line of a child's constructor should use the `super` reference to call the parent's constructor (the parent must exist before the child exists)

If no such call exists, Java will automatically insert `super()` at the beginning of the child's constructor

```
//********************************************************************
//  Book2.java       Author: Lewis/Loftus
//
//  Represents a book. Used as the parent of a derived class to
//  demonstrate inheritance and the use of the super reference.
//********************************************************************

public class Book2
{
   protected int pages;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the book with the specified number of
   //  pages.
   //-----------------------------------------------------------------
   public Book2(int numPages)
   {
      pages = numPages;
   }
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Pages mutator.
    //-----------------------------------------------------------------
    public void setPages(int numPages)
    {
       pages = numPages;
    }

    //-----------------------------------------------------------------
    //  Pages accessor.
    //-----------------------------------------------------------------
    public int getPages()
    {
       return pages;
    }
}
```

```
//********************************************************************
//  Dictionary2.java       Author: Lewis/Loftus
//
//  Represents a dictionary, which is a book. Used to demonstrate
//  the use of the super reference.
//********************************************************************

public class Dictionary2 extends Book2
{
   private int definitions;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the dictionary with the specified number
   //  of pages and definitions.
   //-----------------------------------------------------------------
   public Dictionary2(int numPages, int numDefinitions)
   {
      super(numPages);

      definitions = numDefinitions;
   }
```

Call parent's constructor with parameter numPages

continue

continue

```
    //-----------------------------------------------------------------
    //  Prints a message using both local and inherited values.
    //-----------------------------------------------------------------
    public double computeRatio()
    {
       return (double) definitions/pages;
    }

    //-----------------------------------------------------------------
    //  Definitions mutator.
    //-----------------------------------------------------------------
    public void setDefinitions(int numDefinitions)
    {
       definitions = numDefinitions;
    }

    //-----------------------------------------------------------------
    //  Definitions accessor.
    //-----------------------------------------------------------------
    public int getDefinitions()
    {
       return definitions;
    }
}
```

```
//********************************
//  Words2.java       Author: Lew
//
//  Demonstrates the use of the su
//********************************

public class Words2
{
   //-----------------------------------------------------------------
   //  Instantiates a derived class and invokes its inherited and
   //  local methods.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Dictionary2 webster = new Dictionary2(1500, 52500);

      System.out.println("Number of pages: " + webster.getPages());

      System.out.println("Number of definitions: " +
                         webster.getDefinitions());

      System.out.println("Definitions per page: " +
                         webster.computeRatio());
   }
}
```

Output

```
Number of pages: 1500
Number of definitions: 52500
Definitions per page: 35.0
```

Overriding Methods

A child class can *override* the definition of an inherited method in favor of its own

Usually, the overridden methods are doing something more specific to the child class

The overridden method must have the same signature (number, type, and order of the parameters) as the parent's method, but can have a different body (it is NOT the same as overloading!)

The type of the object executing the method determines which version of the method is invoked

```
//********************************************************************
//  Thought.java       Author: Lewis/Loftus
//
//  Represents a stray thought. Used as the parent of a derived
//  class to demonstrate the use of an overridden method.
//********************************************************************

public class Thought
{
   //-----------------------------------------------------------------
   //  Prints a message.
   //-----------------------------------------------------------------
   public void message()
   {
      System.out.println("I feel like I'm diagonally parked in a " +
                         "parallel universe.");

      System.out.println();
   }
}
```

```java
//********************************************************************
//  Advice.java       Author: Lewis/Loftus
//
//  Represents some thoughtful advice. Used to demonstrate the use
//  of an overridden method.
//********************************************************************

public class Advice extends Thought
{
   //-----------------------------------------------------------------
   //  Prints a message. This method overrides the parent's version.
   //-----------------------------------------------------------------
   public void message()
   {
      System.out.println("Warning: Dates in calendar are closer " +
                         "than they appear.");

      System.out.println();

      super.message();  // explicitly invokes the parent's version
   }
}
```

```
//********************************************************************
//  Messages.java       Author: Lewis/Loftus
//
//  Demonstrates the use of an overridden method.
//********************************************************************

public class Messages
{
   //-----------------------------------------------------------------
   //  Creates two objects and invokes the message method in each.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Thought parked = new Thought();
      Advice dates = new Advice();

      parked.message();

      dates.message(); // overridden version of method in Advice class
   }
}
```

Output

```
I feel like I'm diagonally parked in a parallel universe.

Warning: Dates in calendar are closer than they appear.

I feel like I'm diagonally parked in a parallel universe.
```

Which Method is Called?

make a new Wolf object

```
Wolf w = new Wolf();
```

calls the version in Wolf

```
w.makeNoise();
```

calls the version in Canine

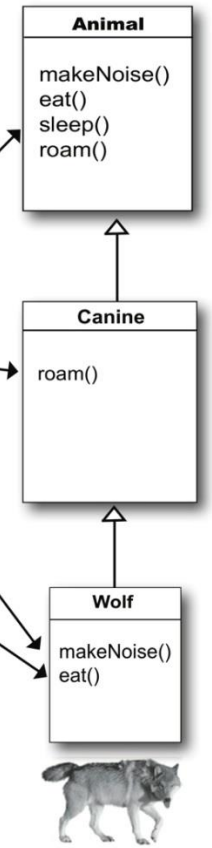
```
w.roam();
```

calls the version in Wolf

```
w.eat();
```

calls the version in Animal

```
w.sleep();
```



When you call a method on an object reference, you're calling the most specific version of the method for that object type.

In other words, ***the lowest one wins!***

"Lowest" meaning lowest on the inheritance tree. Canine is lower than Animal, and Wolf is lower than Canine, so invoking a method on a reference to a Wolf object means the JVM starts

Overriding Methods

A method defined with the `final` modifier cannot be overridden

A shadow variable is a variable in the child class with the same name as a variable in the parent class

Shadowing variables should be avoided (since an inherited variable is already available to the child class)

Two common usages of `super` reference:

- Call a parent's constructor in child's constructor: `super(parameter)`
- Call a parent's method in overridden methods: `super.message()`

Overloading vs. Overriding

Overloading deals with multiple methods with the same name in the same class but with different signatures

Overriding deals with two methods, one in a parent class and one in a child class, that have the same signature

Overloading lets you define a similar operation in different ways for different parameters

Overriding lets you define a similar operation in different ways for different object types

Assignment for Class 24

Read 10.1, 10.2, 10.3