Inheritance

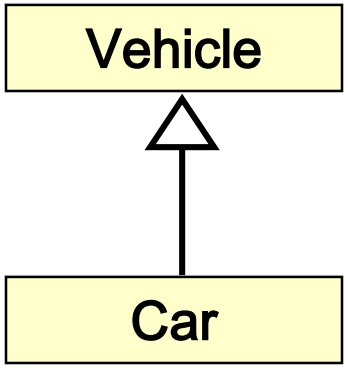
Inheritance is a fundamental object-oriented design technique used to create and organize reusable classes

Inheritance

- Inheritance allows a software developer to derive a new class from an existing one
- The existing class is called the parent class, or superclass, or base class
- The derived class is called the child class or subclass
- As the name implies, the child inherits characteristics of the parent
- That is, the child class inherits the methods and data defined by the parent class

Inheritance

- Inheritance relationships are shown in a UML class diagram using a solid arrow with an unfilled triangular arrowhead pointing to the parent class



- Proper inheritance creates an *is-a* relationship, meaning the child *is a* more specific version of the parent

Inheritance

- A programmer can tailor a derived 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

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

- See `Words.java`
- See `Book.java`
- See `Dictionary.java`

```
//********************************************************************
//  Words.java       Author: Lewis/Loftus
//
//  Demonstrates the use of an inherited method.
//********************************************************************

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

      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
```

```
//********************                                  ********************
//  Words.java
//
//  Demonstrates
//********************                                  ********************

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

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

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

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

```java
//********************************************************************
//  Book.java       Author: Lewis/Loftus
//
//  Represents a book. Used as the parent of a derived class to
//  demonstrate inheritance.
//********************************************************************

public class Book
{
   protected int pages = 1500;

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

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

```
//********************************************************************
//  Dictionary.java       Author: Lewis/Loftus
//
//  Represents a dictionary, which is a book. Used to demonstrate
//  inheritance.
//********************************************************************

public class Dictionary extends Book
{
   private int definitions = 52500;

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

continue

continue

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

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

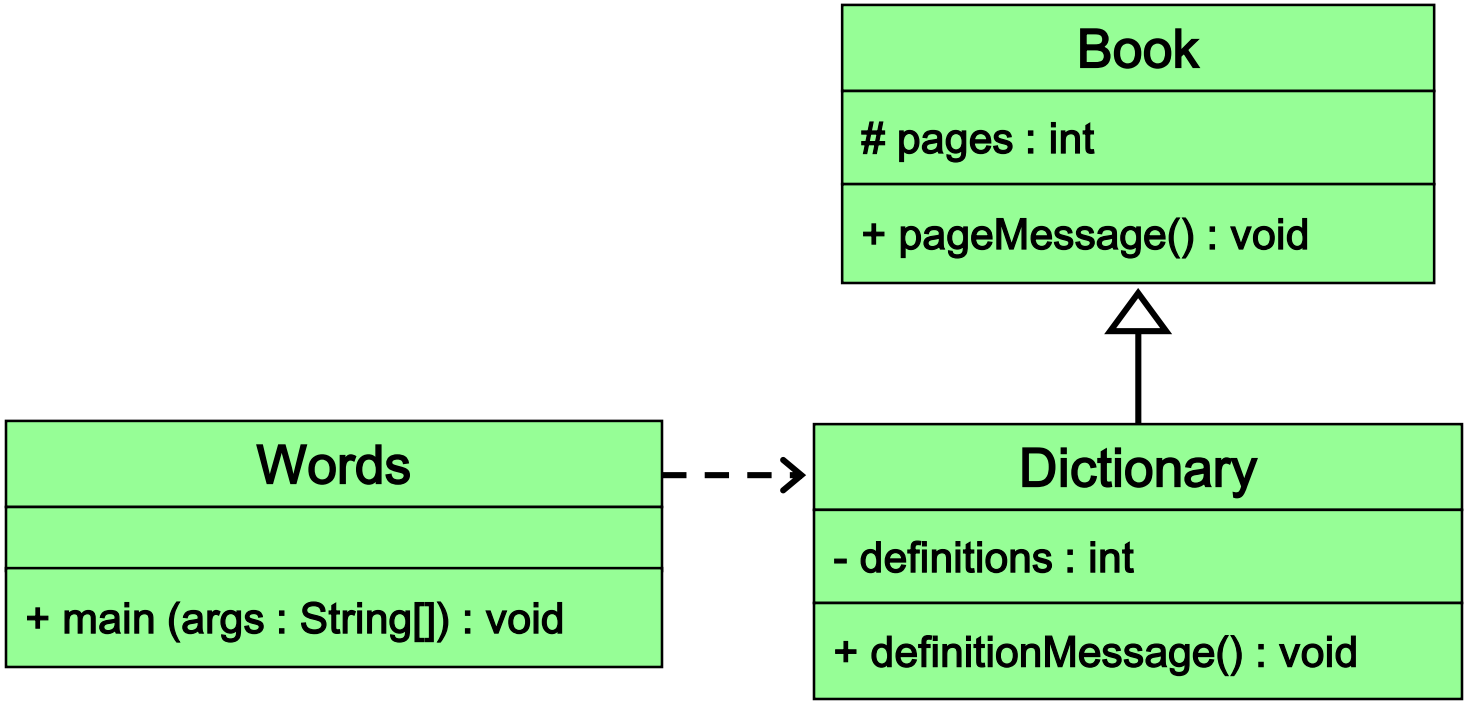
The protected Modifier

- Visibility modifiers affect the way that class members can be used in a child class
- Variables and methods declared with **private visibility cannot be referenced in a child class**
- They can be referenced in the child class if they are declared with public visibility -- but public variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: protected

The protected Modifier

- The protected modifier **allows a child class to reference a variable or method in the child class**
- It provides more encapsulation than public visibility, but is not as tightly encapsulated as private visibility
- A protected variable is also visible to any class in the same package as the parent class
- Protected variables and methods can be shown with a # symbol preceding them in UML diagrams

Class Diagram for Words



The super Reference

- **Constructors are not inherited**, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- 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**

The super Reference

- **The first line of a child's constructor should use the super reference to call the parent's constructor**
- The super reference can also be used to reference other variables and methods defined in the parent's class
- See Words2.java
- See Book2.java
- See Dictionary2.java

```
//********************************************************************
//  Words2.java       Author: Lewis/Loftus
//
//  Demonstrates the use of the super reference.
//********************************************************************

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());
   }
}
```

```
//*****************                                    *****************
//  Words2.java
//
//  Demonstrates t
//*****************                                    *****************

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
```

```
//********************************************************************
//  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;
   }
```

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;
   }
}
```

Multiple Inheritance

- **Java supports single inheritance**, meaning that a derived class can have only one parent class
- Multiple inheritance allows a class to be derived from two or more classes, inheriting the members of all parents
- Collisions, such as the same variable name in two parents, have to be resolved
- Multiple inheritance is generally not needed, and Java does not support it

Overriding Methods

- A child class can override the definition of an inherited method in favor of its own
- The new method must have the same signature as the parent's method, but can have a different body
- The type of the object executing the method determines which version of the method is invoked
- See Messages.java
- See Thought.java
- See Advice.java

```java
//********************************************************************
//  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
   }
}
```

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.
```

```
//-----------------------------------------------------------------
//  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
}
}
```

Overriding

- A method in the parent class can be invoked explicitly using the super reference
- **If a method is declared with the final modifier, it cannot be overridden**
- The concept of overriding can be applied to data and is called **shadowing variables**
- Shadowing variables should be avoided because it tends to cause unnecessarily confusing code

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

Quick Check

True or False?

A child class may define a method with the same name as a method in the parent.

A child class can override the constructor of the parent class.

A child class cannot override a `final` method of the parent class.

It is considered poor design when a child class overrides a method from the parent.

A child class may define a variable with the same name as a variable in the parent.

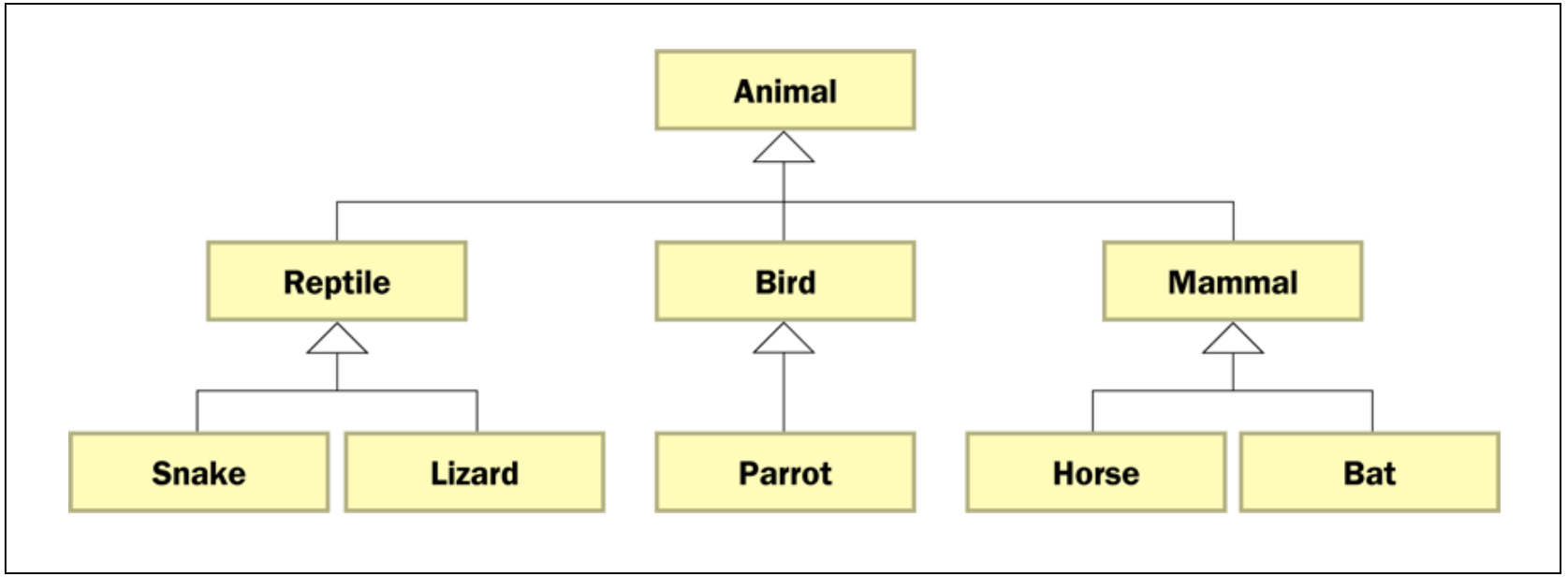
Quick Check

True or False?

A child class may define a method with the same name as a method in the parent.	True
A child class can override the constructor of the parent class.	False
A child class cannot override a `final` method of the parent class.	True
It is considered poor design when a child class overrides a method from the parent.	False
A child class may define a variable with the same name as a variable in the parent.	True, but shouldn't

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**

The Object Class

- The Object class contains a few useful methods, which are inherited by all classes
- For example, the **toString method** is defined in the Object class
- Every time we define the toString method, we are actually overriding an inherited definition
- The toString method in the Object class is defined to return a string that contains the name of the object's class along with a hash code

The Object Class

- The **equals method** of the Object class returns true if two references are aliases
- We can override equals in any class to define equality in some more appropriate way
- As we've seen, the String class defines the equals method to return true if two String objects contain the same characters
- The designers of the String class have overridden the equals method inherited from Object in favor of a more useful version

Visibility Revisited

- It's important to understand one subtle issue related to inheritance and visibility
- All variables and methods of a parent class, even private members, are inherited by its children
- As we've mentioned, private members cannot be referenced by name in the child class
- However**, private members inherited by child classes exist and can be referenced indirectly**

Visibility Revisited

- Because the parent can refer to the private member, the child can reference it indirectly using its parent's methods
- The super reference can be used to refer to the parent class, even if no object of the parent exists
- See FoodAnalyzer.java
- See FoodItem.java
- See Pizza.java

```
//********************************************************************
//  FoodAnalyzer.java       Author: Lewis/Loftus
//
//  Demonstrates indirect access to inherited private members.
//********************************************************************

public class FoodAnalyzer
{
   //-----------------------------------------------------------------
   //  Instantiates a Pizza object and prints its calories per
   //  serving.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Pizza special = new Pizza (275);

      System.out.println ("Calories per serving: " +
                          special.caloriesPerServing());
   }
}
```

```
//*******************                                    ********************
//  FoodAnalyzer.
//
//  Demonstrates                                        vate members.
//*******************************************************************

public class FoodAnalyzer
{
   //-----------------------------------------------------------------
   //  Instantiates a Pizza object and prints its calories per
   //  serving.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Pizza special = new Pizza (275);

      System.out.println ("Calories per serving: " +
                          special.caloriesPerServing());
   }
}
```

Output

```
Calories per serving: 309
```

```
//********************************************************************
//  FoodItem.java       Author: Lewis/Loftus
//
//  Represents an item of food. Used as the parent of a derived class
//  to demonstrate indirect referencing.
//********************************************************************

public class FoodItem
{
   final private int CALORIES_PER_GRAM = 9;
   private int fatGrams;
   protected int servings;

   //-----------------------------------------------------------------
   //  Sets up this food item with the specified number of fat grams
   //  and number of servings.
   //-----------------------------------------------------------------
   public FoodItem (int numFatGrams, int numServings)
   {
      fatGrams = numFatGrams;
      servings = numServings;
   }
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Computes and returns the number of calories in this food item
    //  due to fat.
    //-----------------------------------------------------------------
    private int calories()
    {
       return fatGrams * CALORIES_PER_GRAM;
    }

    //-----------------------------------------------------------------
    //  Computes and returns the number of fat calories per serving.
    //-----------------------------------------------------------------
    public int caloriesPerServing()
    {
       return (calories() / servings);
    }
}
```

```java
//********************************************************************
//  Pizza.java       Author: Lewis/Loftus
//
//  Represents a pizza, which is a food item. Used to demonstrate
//  indirect referencing through inheritance.
//********************************************************************

public class Pizza extends FoodItem
{
   //-----------------------------------------------------------------
   //  Sets up a pizza with the specified amount of fat (assumes
   //  eight servings).
   //-----------------------------------------------------------------
   public Pizza (int fatGrams)
   {
      super (fatGrams, 8);
   }
}
```

Constructors in Inheritance

- Do you have to provide a constructor for a class?
  - No, a no-argument, default constructor is implicitly provided
- This default constructor calls the no-argument constructor of the super-class
- If super-class does not have no-argument version of the constructor
  - A compilation error occurs
  - Make sure there is such a constructor in the superclass
- What if there is no super-class?
  - Object class is the implicit super-class, and it does have a no-argument constructor

Constructors

- What happens when no constructor is defined for a class?

➔ No-argument constructor will be provided implicitly for that class

```
class C2 extends C1 {
        // no constructor is defined
        public C2() { // this constructor is
          super();        // automatically provided by Java.
        }
}
```

- If another constructor of the class is present, no-argument version of the constructor will not be provided automatically.

Constructors of Sub-Classes

- The initialization of the fields of a sub-class consists of two phases:
  - The initialization of the inherited fields
  - The initialization of the fields that are declared in that sub-class.
- **One of the constructors of the super-class must be invoked to initialize the fields inherited from the super-class.**
  - A super-class constructor must be invoked explicitly; **Otherwise the no-argument constructor of the super-class will be automatically invoked**
  - This invocation must be the first statement of the constructor of the sub-class. (Or the one of the constructors of the sub-class must invoke another version of the constructor of that sub-class).

Constructors of Sub-Classes

```
class C1 {
        private int x;
        public C1() { x=0; }
        public C1(int xv) { x=xv; }
}
class C2 extends C1 {
        private int y;
        public C2(int xv, int yv) {
                super(xv);
                y = yv;
        }
        public C2(int yv) {
                this(1,yv);
        }
        public C2() {
                y = 2;
        } }
```

the constructor of the super-class is explicitly invoked

Another version of the constructor of this class is invoked

No-argument version of the constructor of the super-class is implicitly invoked.

super();

If the subclass constructor does not specify which superclass constructor to invoke then the compiler will automatically call the accessible no-args constructor in the superclass.

Order of Initialization Step

- The fields of the super-class are initialized using default values.
- One of the constructors of the super-class is executed.
- The fields of the extended class (sub-class) are initialized using the default values.
- One of the constructors of the extended class (sub-class) is executed.

Order of Initialization Step (cont.)

```
class C1 {
        private int x = 1;        // executed first
        public C1() {
                x = 2;            // executed second
        }
}
class C2 extends C1 {
        private int y = 3;        // executed third
        public C2() {
                super();
                y = 4 ;           // executed fourth
        }
}
```

Inheritance Design Issues

- Every derivation should be an is-a relationship
- Think about the potential future of a class hierarchy, and design classes to be reusable and flexible
- Find common characteristics of classes and push them as high in the class hierarchy as appropriate
- Override methods as appropriate to tailor or change the functionality of a child
- Add new variables to children, but don't redefine (shadow) inherited variables

Inheritance Design Issues

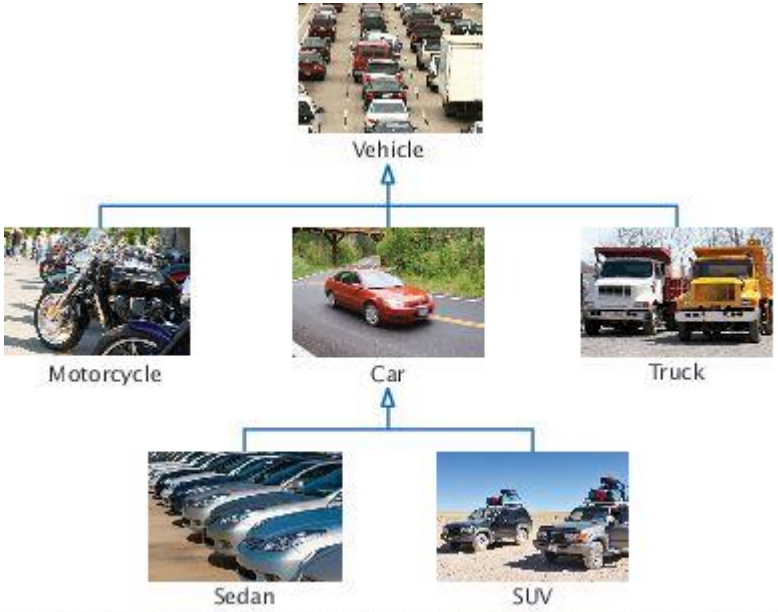
- Allow each class to manage its own data; use the super reference to invoke the parent's constructor to set up its data
- Override general methods such as toString and equals with appropriate definitions
- Use abstract classes to represent general concepts that derived classes have in common
- Use visibility modifiers carefully to provide needed access without violating encapsulation

Restricting Inheritance

- If the **final modifier** is applied to a method, that method cannot be overridden in any derived classes
- If the final modifier is applied to an entire class, then that class cannot be used to derive any children at all
- Therefore, an abstract class cannot be declared as final

Self Check 9.4

Consider the method `doSomething(Car c)`. List all vehicle classes from Figure whose objects *cannot* be passed to this method.



Answer: `Vehicle`, `Truck`, `Motorcycle`

Self Check 9.7

Suppose the class `Employee` is declared as follows:

```
public class Employee
{
   private String name;
   private double baseSalary;
   public void setName(String newName) { . . . }
   public void setBaseSalary(double newSalary) { . . . }
   public String getName() { . . . }
   public double getSalary() { . . . }
}
```

Declare a class `Manager` that inherits from the class `Employee` and adds an instance variable `bonus` for storing a salary bonus. Omit constructors and methods.

Continued

Self Check 9.7

Answer:

```
public class Manager extends Employee
{
   private double bonus;
   // Constructors and methods omitted
}
```

Self Check 9.8

Which instance variables does the `Manager` class have?

Answer: `name`, `baseSalary`, and `bonus`

Self Check 9.9

In the `Manager` class, provide the method header (but not the implementation) for a method that overrides the `getSalary` method from the class `Employee`.

Answer:

```
public class Manager extends Employee
{
   . . .
   public double getSalary() {
      . . .
   }
}
```

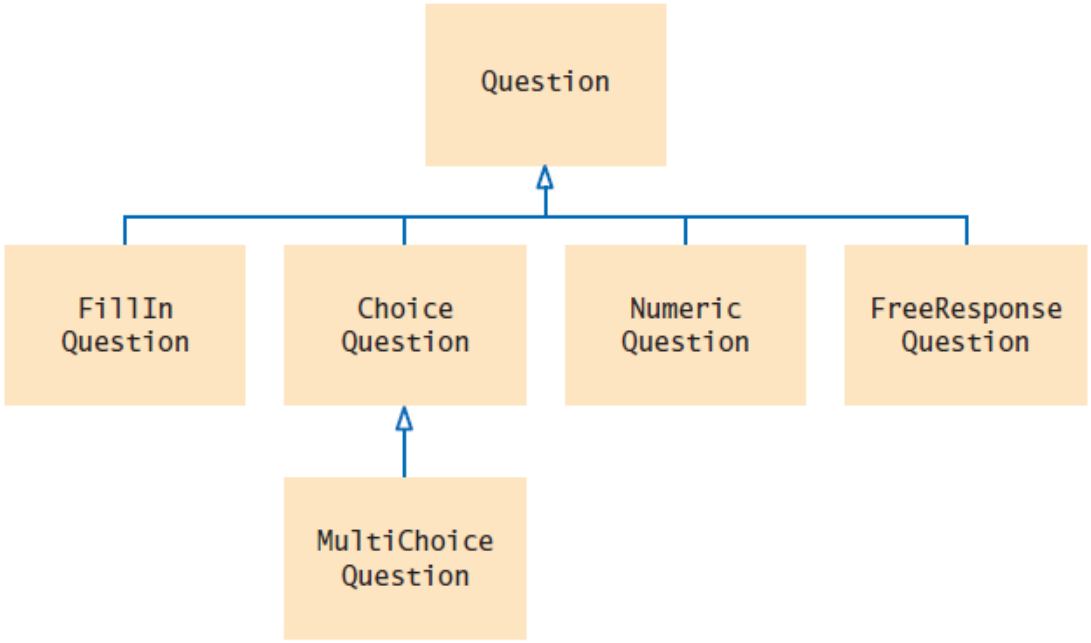
Self Check 9.10

Which methods does the `Manager` class from Self Check 9 inherit?

Answer: `getName`, `setName`, `setBaseSalary`

Example - Questions

Inheritance Hierarchies



Inheritance Hierarchy of Question Types

- Example: Computer-graded quiz
  - There are different kinds of questions
  - A question can display its text, and it can check whether a given response is a correct answer.
  - You can form subclasses of the `Question` class.

section_1/Question.java

```
1  /**
2     A question with a text and an answer.
3  */
4  public class Question
5  {
6     private String text;
7     private String answer;
8
9     /**
10       Constructs a question with empty question and answer.
11    */
12    public Question()
13    {
14       text = "";
15       answer = "";
16    }
17
18    /**
19       Sets the question text.
20       @param questionText the text of this question
21    */
22    public void setText(String questionText)
23    {
24       text = questionText;
25    }
26
```

Continued

section_1/Question.java

```
27      /**
28         Sets the answer for this question.
29         @param correctResponse the answer
30      */
31      public void setAnswer(String correctResponse)
32      {
33         answer = correctResponse;
34      }
35
36      /**
37         Checks a given response for correctness.
38         @param response the response to check
39         @return true if the response was correct, false otherwise
40      */
41      public boolean checkAnswer(String response)
42      {
43         return response.equals(answer);
44      }
45
46      /**
47         Displays this question.
48      */
49      public void display()
50      {
51         System.out.println(text);
52      }
53   }
```

section_1/QuestionDemo1.java

```
 1  import java.util.Scanner;
 2
 3  /**
 4     This program shows a simple quiz with one question.
 5  */
 6  public class QuestionDemo1
 7  {
 8     public static void main(String[] args)
 9     {
10        Scanner in = new Scanner(System.in);
11
12        Question q = new Question();
13        q.setText("Who was the inventor of Java?");
14        q.setAnswer("James Gosling");
15
16        q.display();
17        System.out.print("Your answer: ");
18        String response = in.nextLine();
19        System.out.println(q.checkAnswer(response));
20     }
21  }
22
```

Continued

section_1/QuestionDemo1.java

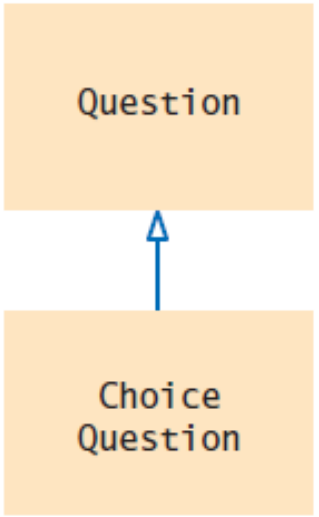
Program Run:

```
Who was the inventor of Java?
Your answer: James Gosling
true
```

Implementing Subclasses

- To get a `ChoiceQuestion` class, implement it as a subclass of `Question`
  - Specify what makes the subclass different from its superclass.
  - Subclass objects automatically have the instance variables that are declared in the superclass.
  - Only declare instance variables that are not part of the superclass objects.

- A subclass inherits all methods that it does not override.



Implementing Subclasses

- The subclass inherits all public methods from the superclass.
- You declare any methods that are new to the subclass.
- You change the implementation of inherited methods if the inherited behavior is not appropriate.
- **Override a method**: supply a new implementation for an inherited method

Implementing Subclasses

A `ChoiceQuestion` object differs from a `Question` object in three ways:

- Its objects store the various choices for the answer.
- There is a method for adding answer choices.
- The display method of the `ChoiceQuestion` class shows these choices so that the respondent can choose one of them.

Implementing Subclasses

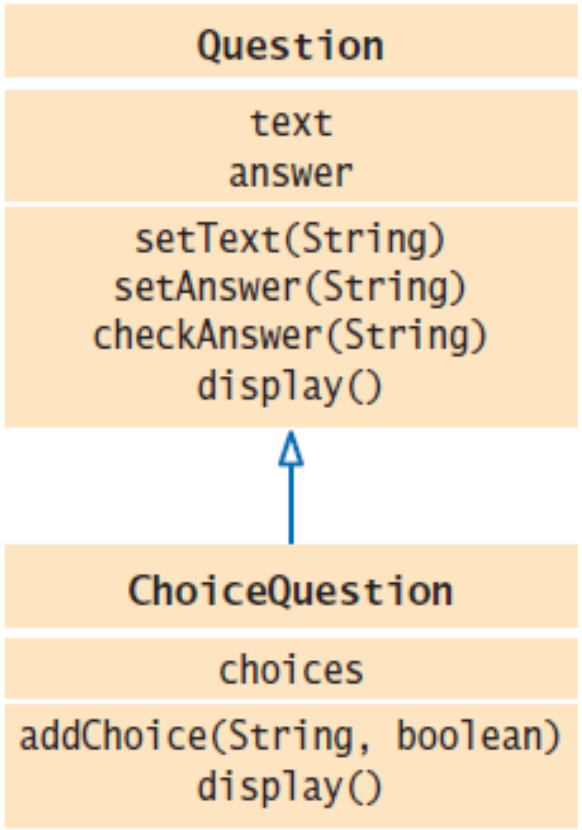
- The `ChoiceQuestion` class needs to spell out the three differences:

```
public class ChoiceQuestion extends Question
{
   // This instance variable is added to the subclass
   private ArrayList<String> choices;

   // This method is added to the subclass
   public void addChoice(String choice, boolean correct) { . . . }

   // This method overrides a method from the superclass
   public void display() { . . . }
}
```

- The `extends` reserved word indicates that a class inherits from a superclass.

Implementing Subclasses

- UML of `ChoiceQuestion` and `Question`

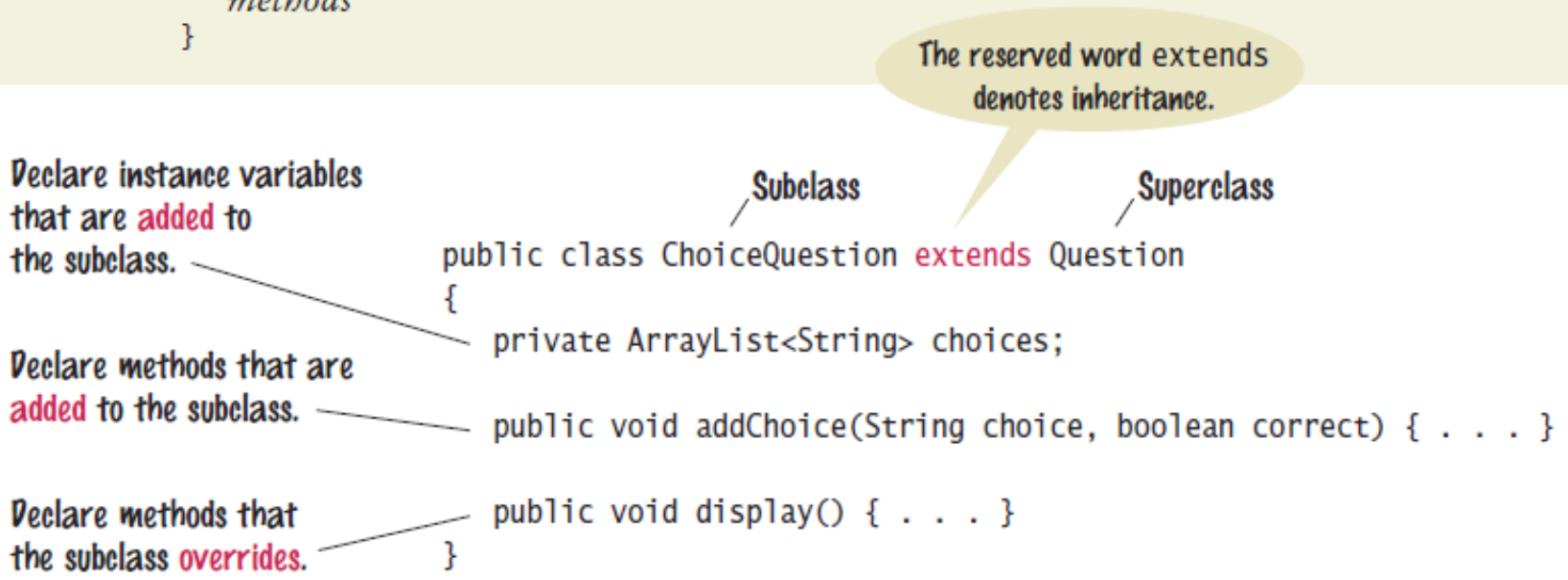


The `ChoiceQuestion` Class Adds an Instance Variable and a Method, and Overrides a Method

Syntax 9.1 Subclass Declaration

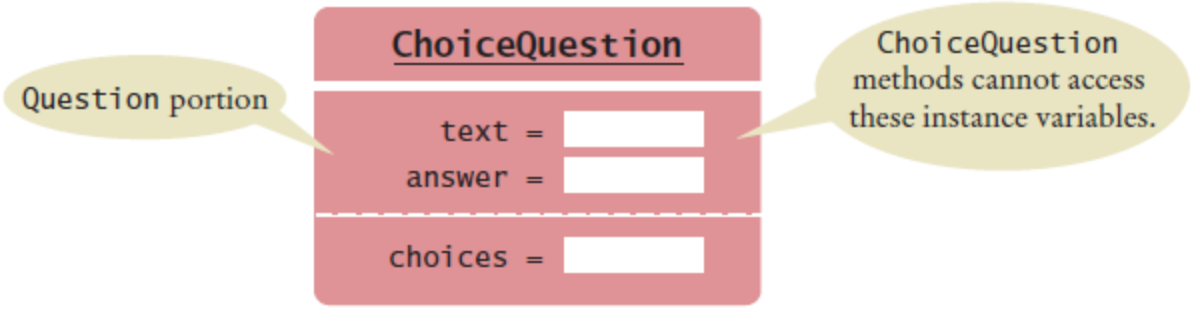
Syntax

```
public class SubclassName extends SuperclassName
{
   instance variables
   methods
}
```



Implementing Subclasses

- A `ChoiceQuestion` object



- You can call the inherited methods on a subclass object:

```
choiceQuestion.setAnswer("2");
```

- The private instance variables of the superclass are inaccessible.
- The `ChoiceQuestion` methods cannot directly access the instance variable `answer`.
- `ChoiceQuestion` methods must use the public interface of the `Question` class to access its private data.

Implementing Subclasses

- Adding a new method: addChoice

```
public void addChoice(String choice, boolean correct)
{
   choices.add(choice);
   if (correct)
   {
      // Convert choices.size() to string
      String choiceString = "" + choices.size();
      setAnswer(choiceString);
   }
}
```

Implementing Subclasses

- `addChoice` method can not just access the `answer` variable in the superclass:
- It must use the `setAnswer` method
- Invoke `setAnswer` on the implicit parameter:

```
setAnswer(choiceString);
```

OR

```
this.setAnswer(choiceString);
```

Overriding Methods

- Problem: `ChoiceQuestion`'s display method can't access the `text` variable of the superclass directly because it is `private`.
- Solution: It can call the `display` method of the superclass, by using the reserved word `super`

```
public void display()
{
   // Display the question text
   super.display(); // OK
   // Display the answer choices
   . . .
}
```

- `super` is a reserved word that forces execution of the superclass method.

section_3/ChoiceQuestion.java

```
1  import java.util.ArrayList;
2
3  /**
4     A question with multiple choices.
5  */
6  public class ChoiceQuestion extends Question
7  {
8     private ArrayList<String> choices;
9
10    /**
11       Constructs a choice question with no choices.
12    */
13    public ChoiceQuestion()
14    {
15       choices = new ArrayList<String>();
16    }
17
```

Continued

section_3/ChoiceQuestion.java

```
18      /**
19         Adds an answer choice to this question.
20         @param choice the choice to add
21         @param correct true if this is the correct choice, false otherwise
22      */
23      public void addChoice(String choice, boolean correct)
24      {
25         choices.add(choice);
26         if (correct)
27         {
28            // Convert choices.size() to string
29            String choiceString = "" + choices.size();
30            setAnswer(choiceString);
31         }
32      }
33
```

Continued

section_3/ChoiceQuestion.java

```
34      public void display()
35      {
36         // Display the question text
37         super.display();
38         // Display the answer choices
39         for (int i = 0; i < choices.size(); i++)
40         {
41            int choiceNumber = i + 1;
42            System.out.println(choiceNumber + ": " + choices.get(i));
43         }
44      }
45   }
46
```

section_3/QuestionDemo2.java

```
1  import java.util.Scanner;
2
3  /**
4     This program shows a simple quiz with two choice questions.
5  */
6  public class QuestionDemo2
7  {
8     public static void main(String[] args)
9     {
10       ChoiceQuestion first = new ChoiceQuestion();
11       first.setText("What was the original name of the Java language?");
12       first.addChoice("*7", false);
13       first.addChoice("Duke", false);
14       first.addChoice("Oak", true);
15       first.addChoice("Gosling", false);
16
17       ChoiceQuestion second = new ChoiceQuestion();
18       second.setText("In which country was the inventor of Java born?");
19       second.addChoice("Australia", false);
20       second.addChoice("Canada", true);
21       second.addChoice("Denmark", false);
22       second.addChoice("United States", false);
23
24       presentQuestion(first);
25       presentQuestion(second);
26    }
27
```

Continued

section_3/QuestionDemo2.java

```
28      /**
29         Presents a question to the user and checks the response.
30         @param q the question
31      */
32      public static void presentQuestion(ChoiceQuestion q)
33      {
34         q.display();
35         System.out.print("Your answer: ");
36         Scanner in = new Scanner(System.in);
37         String response = in.nextLine();
38         System.out.println(q.checkAnswer(response));
39      }
40   }
41
```

Continued

section_3/QuestionDemo2.java

Program Run:

```
What was the original name of the Java language?
1: *7
2: Duke
3: Oak
4: Gosling
Your answer: *7
 false
In which country was the inventor of Java born?
1: Australia
2: Canada
3: Denmark
4: United States
Your answer: 2
true
```

Self Check 9.5

Should a class `Quiz` inherit from the class `Question`? Why or why not?

Answer: It shouldn't. A quiz isn't a question; it *has* questions.

Common Error: Replicating Instance Variables from the Superclass

- A subclass has no access to the private instance variables of the superclass:

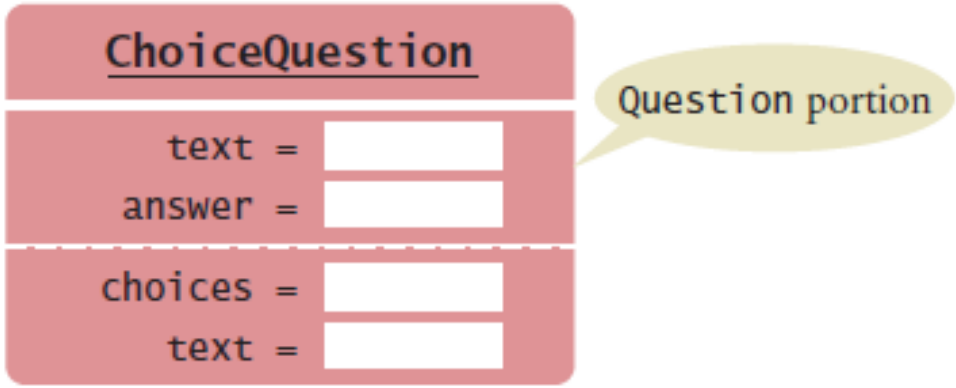
```
public ChoiceQuestion(String questionText)
{
   text = questionText; // Error—tries to access
                        // private superclass variable
}
```

- Beginner's error: “solve” this problem by adding another instance variable with same name
- Error!

```
public class ChoiceQuestion extends Question
{
   private ArrayList<String> choices;
   private String text; // Don’t!
   . . .
}
```

Common Error: Replicating Instance Variables from the Superclass

- The constructor compiles, but it doesn't set the correct text!



- The `ChoiceQuestion` constructor should call the `setText` method of the `Question` class.

Self Check 9.11

What is wrong with the following implementation of the `display` method?

```
public class ChoiceQuestion
{
   . . .
   public void display()
   {
      System.out.println(text);
      for (int i = 0; i < choices.size(); i++)
      {
         int choiceNumber = i + 1;
         System.out.println(choiceNumber + ": " + choices.get(i));
      }
   }
}
```

Answer: The method is not allowed to access the instance variable `text` from the superclass.

Self Check 9.11

What is wrong with the following implementation of the `display` method?

```
public class ChoiceQuestion
{
   . . .
   public void display()
   {
      this.display();
      for (int i = 0; i < choices.size(); i++)
      {
         int choiceNumber = i + 1;
         System.out.println(choiceNumber + ": " +
            choices.get(i));
      }
   }
}
```

Answer: The type of the `this` reference is `ChoiceQuestion`. Therefore, the `display` method of `ChoiceQuestion` is selected, and the method calls itself.

Self Check 9.13

Look again at the implementation of the `addChoice` method that calls the `setAnswer` method of the superclass. Why don't you need to call `super.setAnswer`?

Answer: Because there is no ambiguity. The subclass doesn't have a `setAnswer` method.