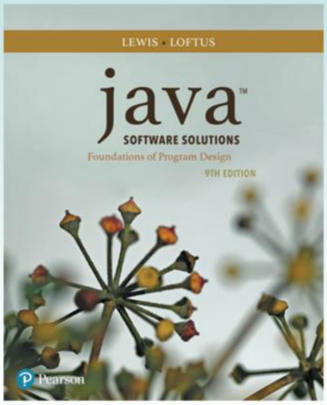
Chapter 9
Inheritance

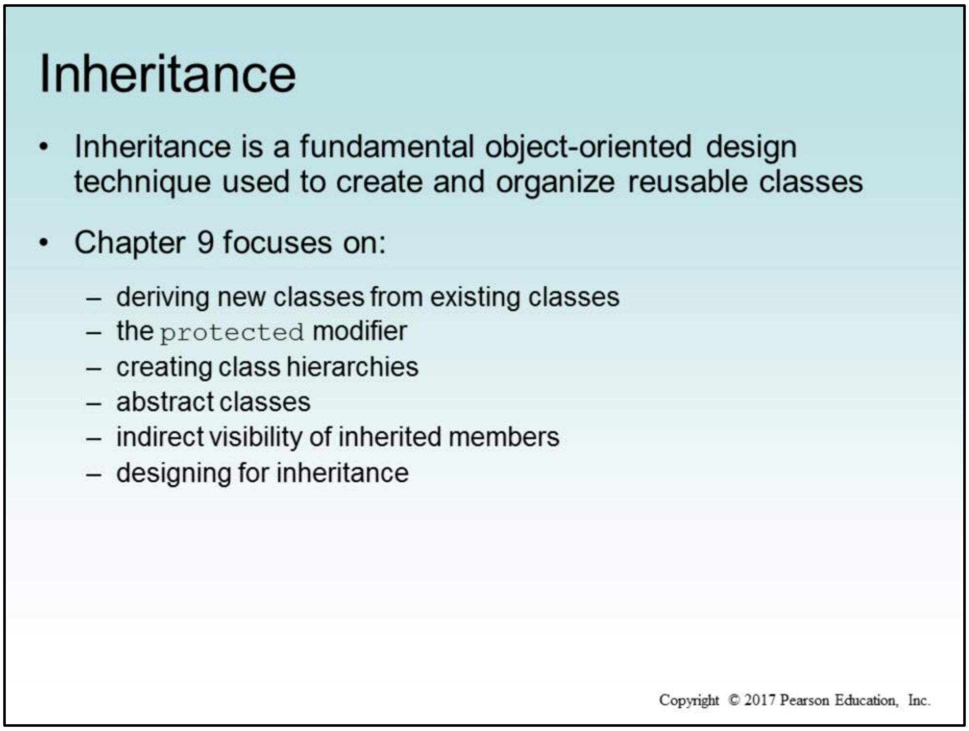


Java Software Solutions

Foundations of Program Design

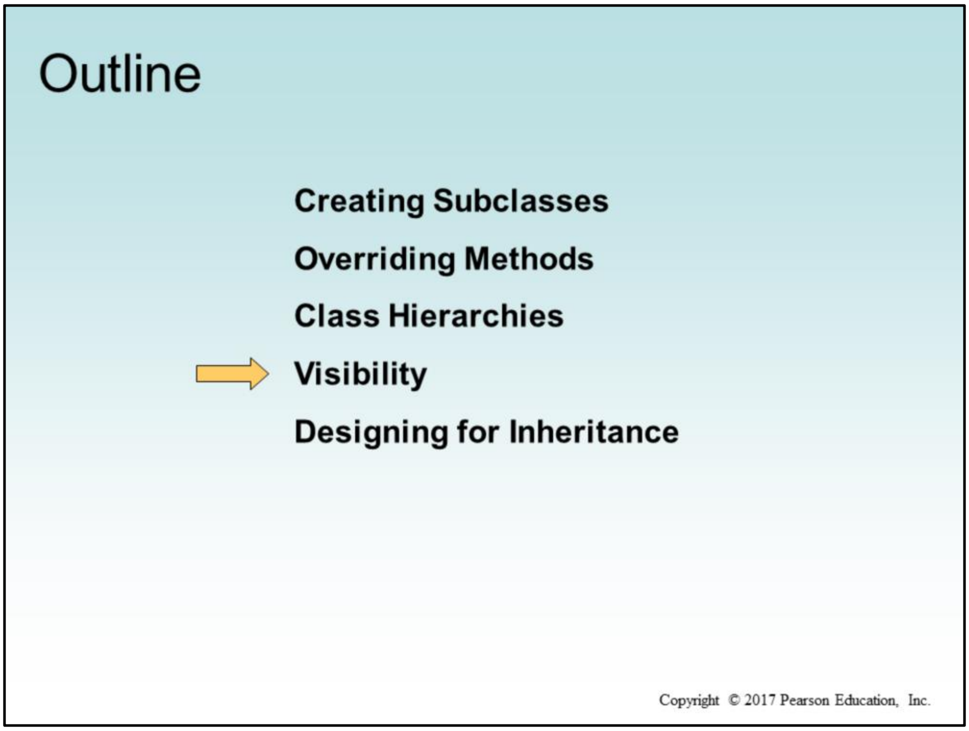
9th Edition

John Lewis
William Loftus



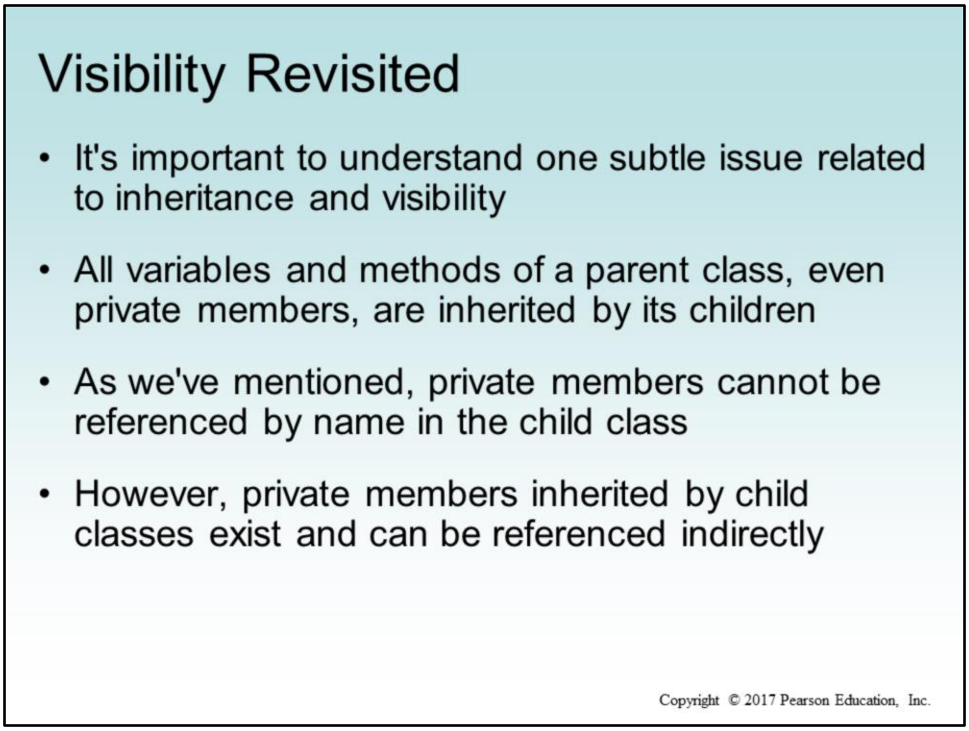
Inheritance

- Inheritance is a fundamental object-oriented design technique used to create and organize reusable classes
- Chapter 9 focuses on:
  - deriving new classes from existing classes
  - the `protected` modifier
  - creating class hierarchies
  - abstract classes
  - indirect visibility of inherited members
  - designing for inheritance



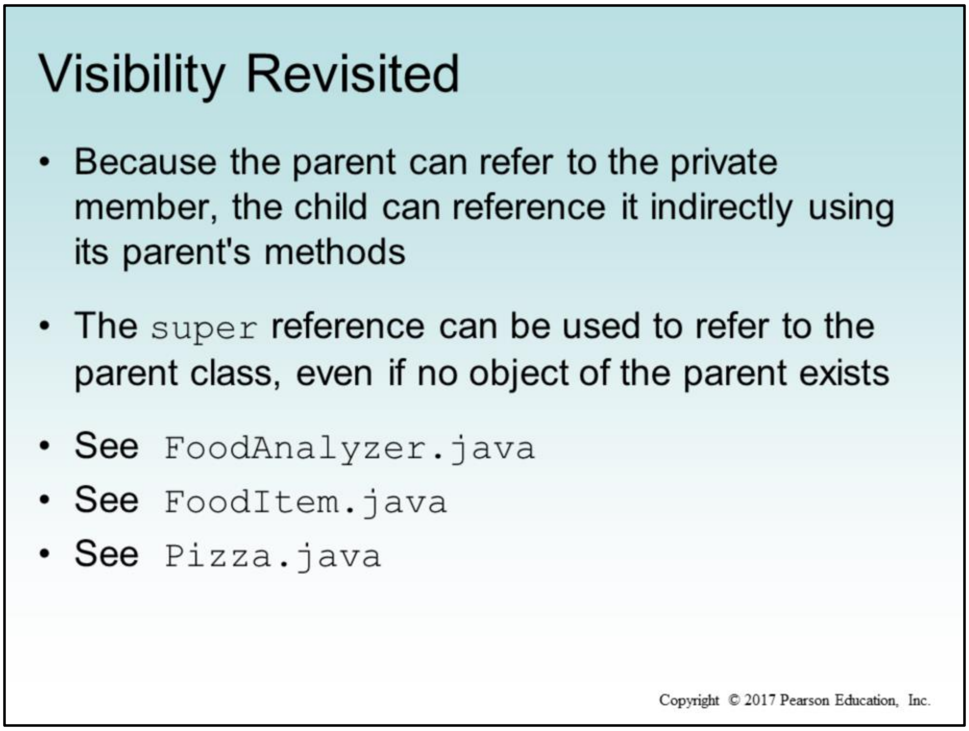
Outline

- Creating Subclasses
- Overriding Methods
- Class Hierarchies
- Visibility
- Designing for Inheritance



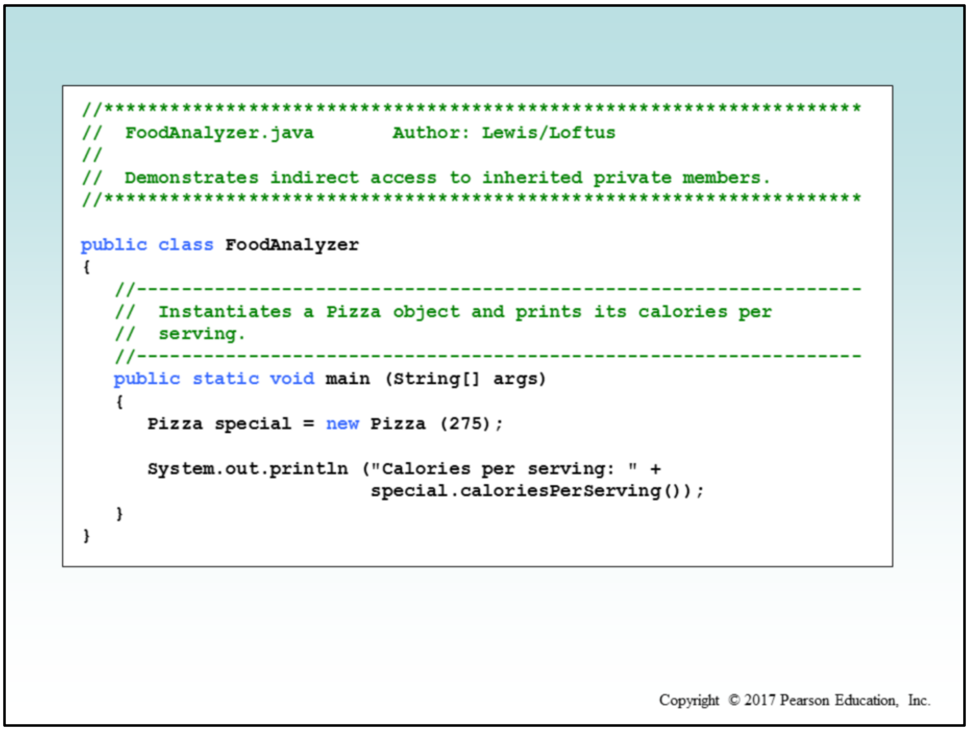
Visibility Revisited

- 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

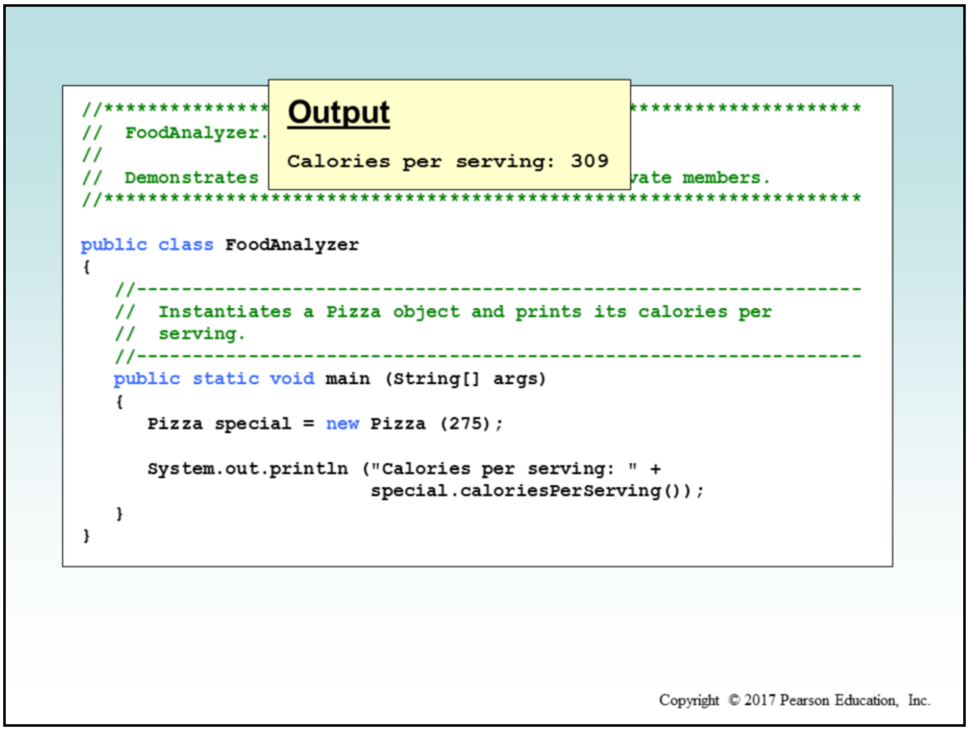


Visibility Revisited

- 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());
   }
}
```

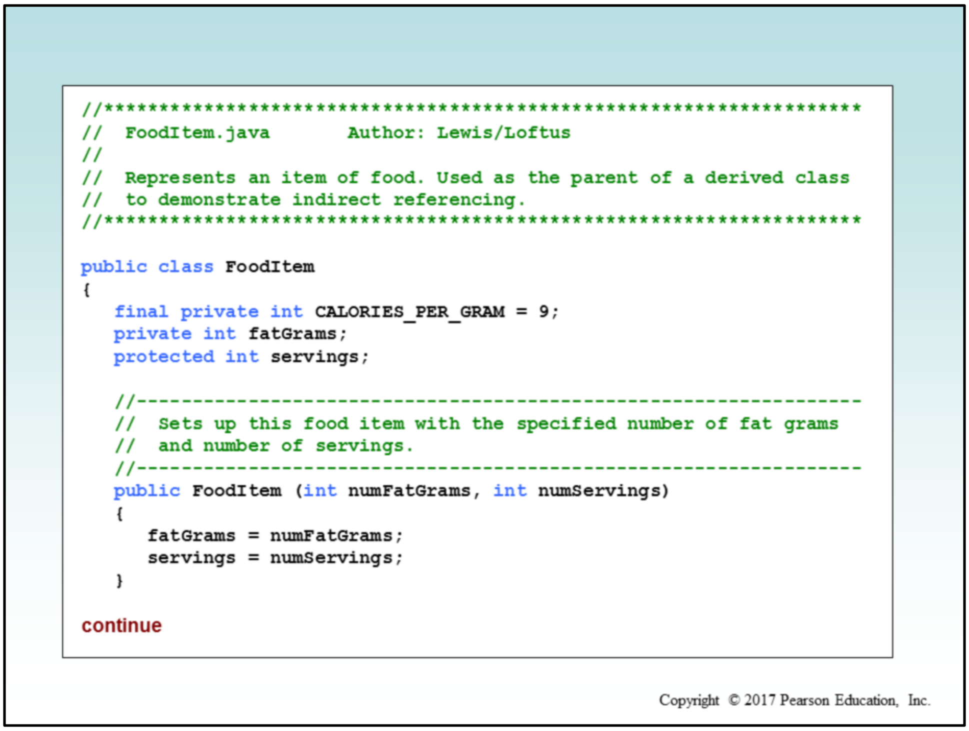
Output

```
Calories per serving: 309
```

-This example is simply demonstrating how a private method in a parent class is called **indirectly**

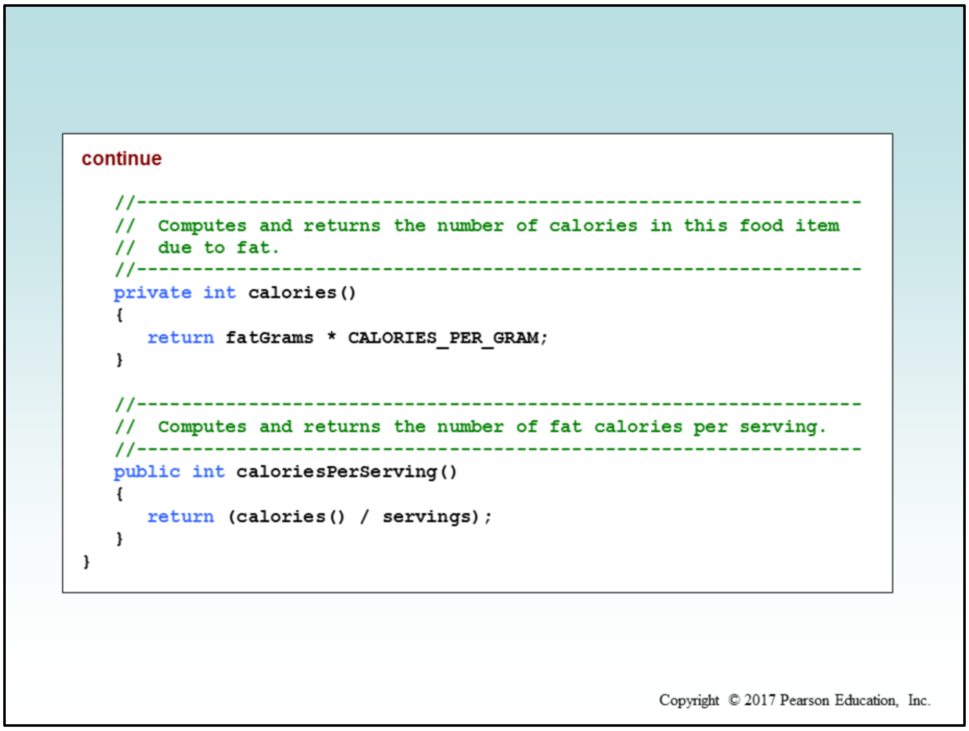
-Specifically, the public caloriesPerServing method calls the private calories method in the parent (FoodItem) class

-Since this public method is called by a child (Pizza) object above, it **indirectly** calls the private calories method



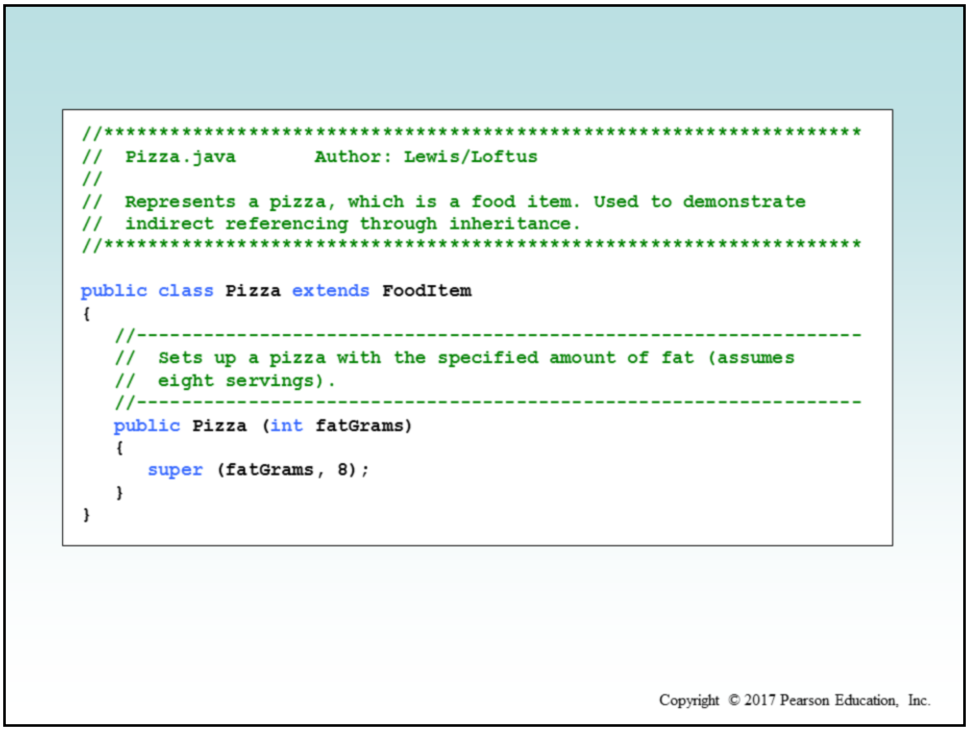
```java
//********************************************************************
//  FoodItem.java       Author: Lewis/Loftus
//
//  Represents an item of food. Used as the parent of a derived class
//  to demonstrate indirect referencing.
//********************************************************************

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);
   }
}
```



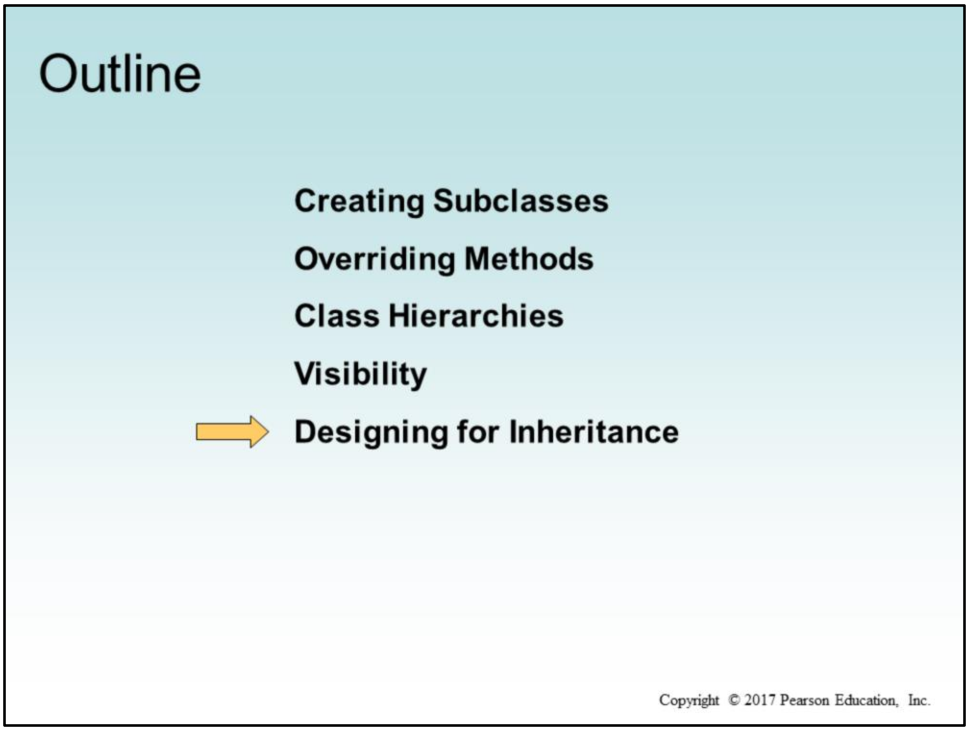
```
//********************************************************************
//  Pizza.java       Author: Lewis/Loftus
//
//  Represents a pizza, which is a food item. Used to demonstrate
//  indirect referencing through inheritance.
//********************************************************************

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);
   }
}
```

-One interesting observation here is that a child class doesn’t have to necessarily add anything when derived

-In other words, the Pizza child class doesn’t add instance variables or methods

-It exists solely to pass the fat grams and number of servings (8) to the parent constructor to create a Pizza object

-Note the use of the **super** keyword to specify the parent constructor above.



Outline

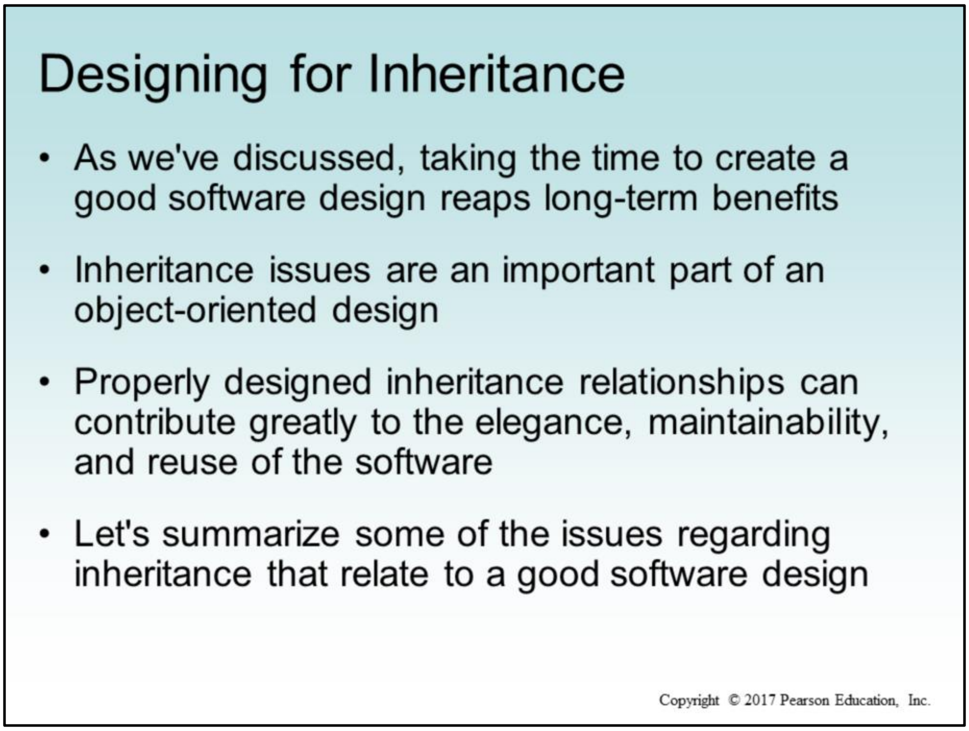
Creating Subclasses

Overriding Methods

Class Hierarchies

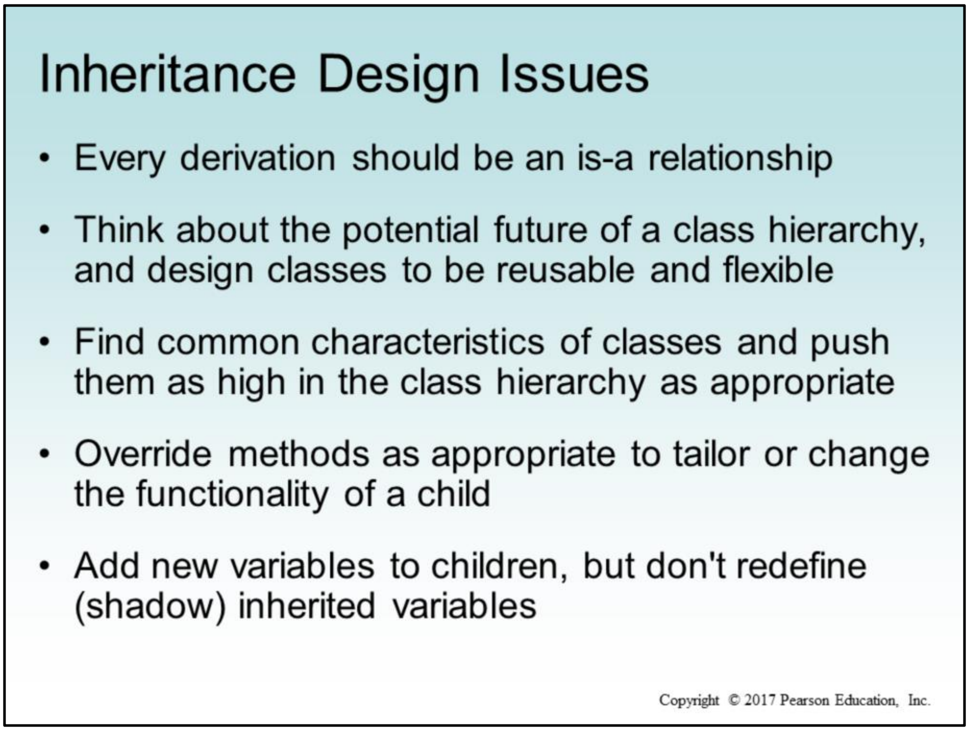
Visibility

➡ Designing for Inheritance



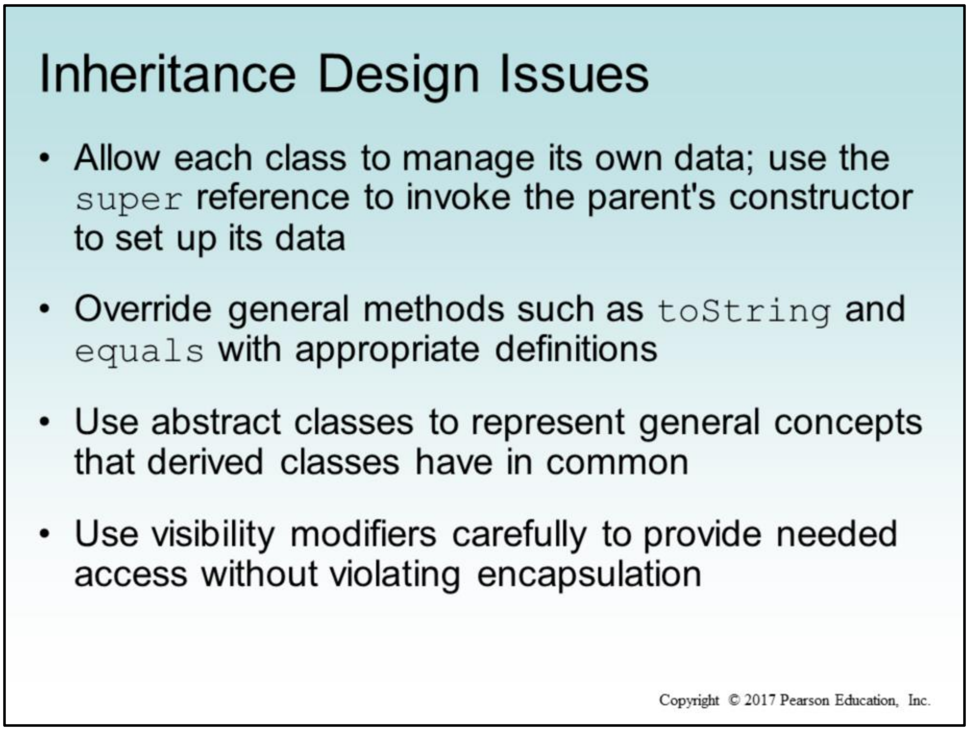
Designing for Inheritance

- As we've discussed, taking the time to create a good software design reaps long-term benefits
- Inheritance issues are an important part of an object-oriented design
- Properly designed inheritance relationships can contribute greatly to the elegance, maintainability, and reuse of the software
- Let's summarize some of the issues regarding inheritance that relate to a good software design



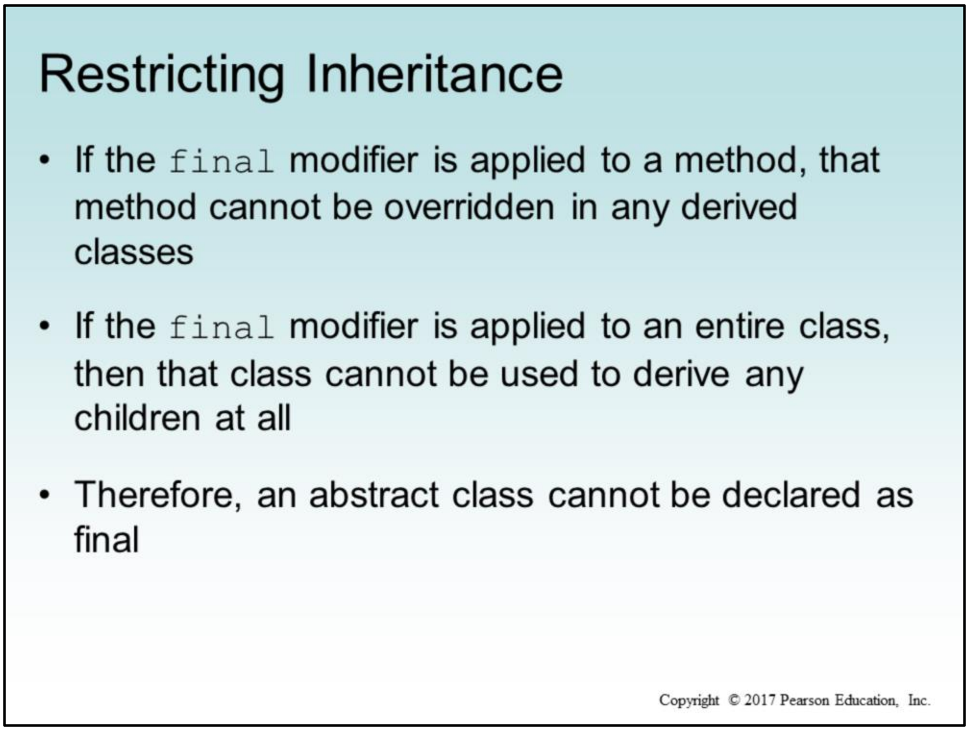
Inheritance Design Issues

- 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