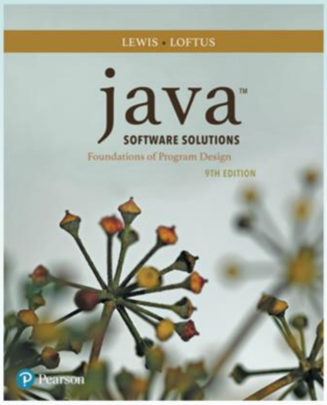
Chapter 9
Inheritance

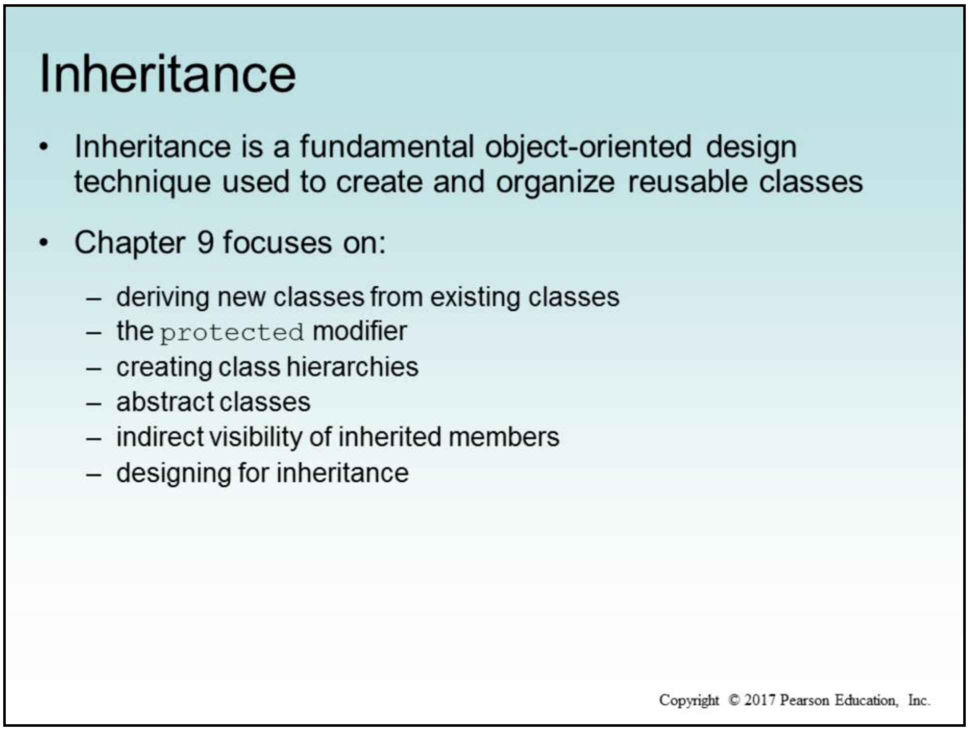


Java Software Solutions

Foundations of Program Design

9th Edition

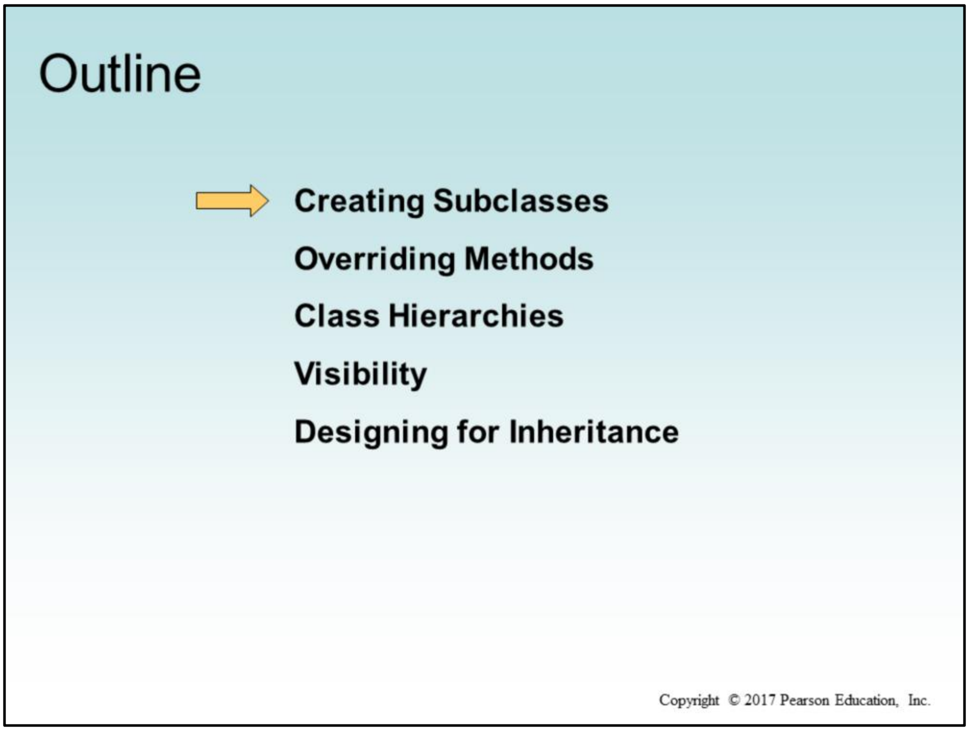
John Lewis
William Loftus



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

-One of the key advantages of inheritance is the ability to **reuse** existing classes to create new ones



Outline

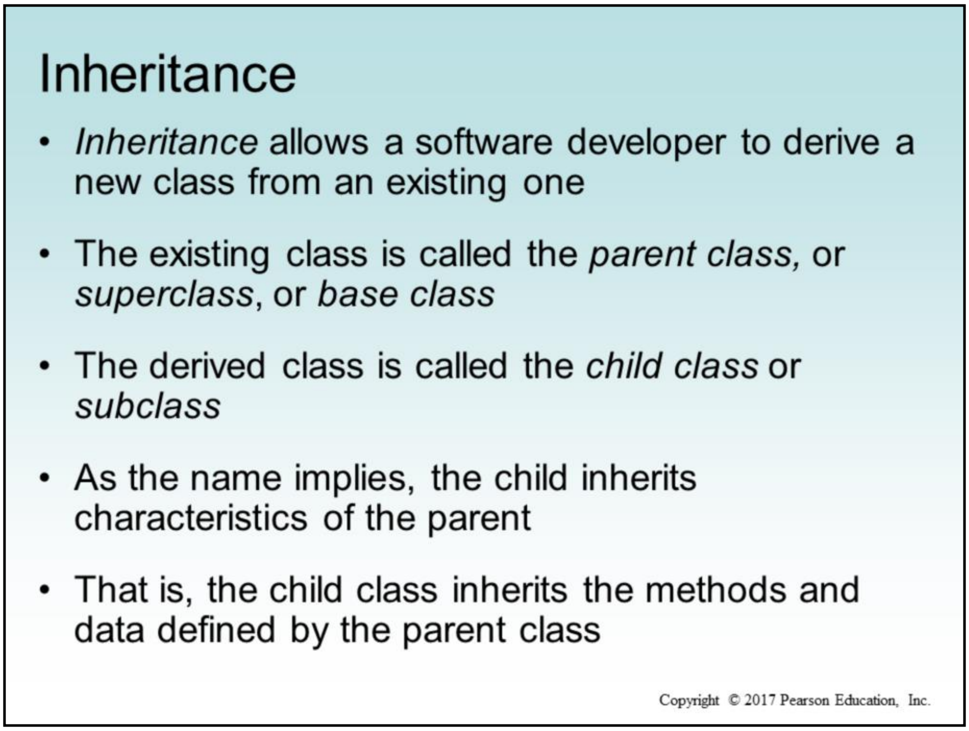
➡ Creating Subclasses

Overriding Methods

Class Hierarchies

Visibility

Designing for Inheritance



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 models the relationships we see in the real world

-Consider, for example, how children inherit properties and behaviors from their parent(s)

-They **inherit** these characteristics from the parent(s) and then **add** their own unique characteristics

-In a similar way, a class can be created by inheriting from another class, then adding its own uniqueness

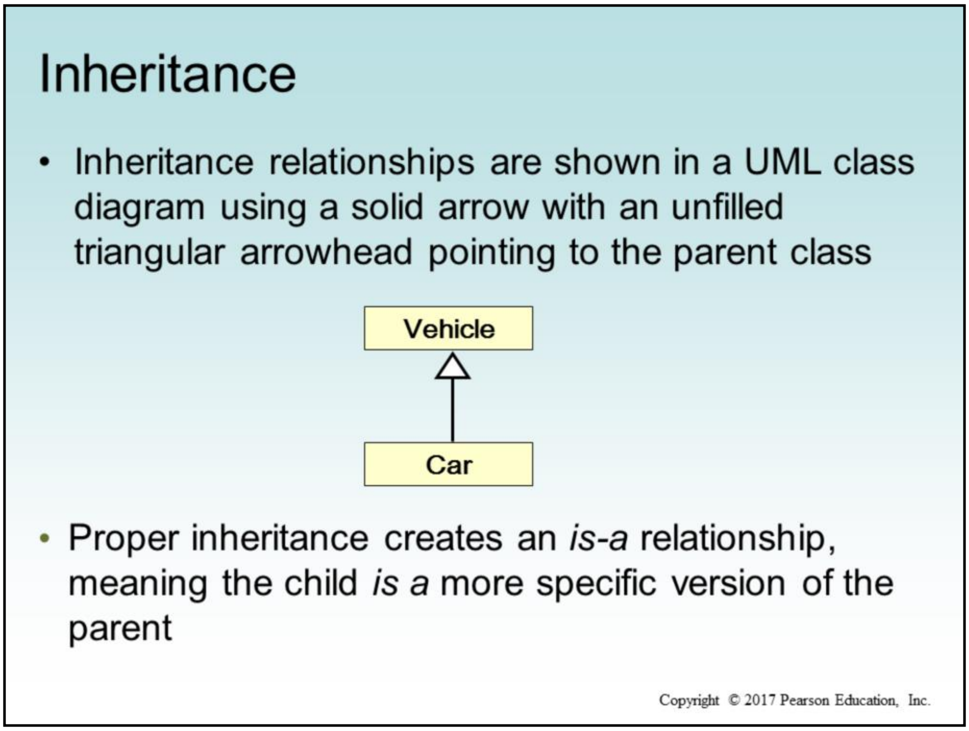
-The new class that is inheriting from another is called the **subclass**, or **child**, or **derived** class

-The existing class that the new class is created from is called the **superclass**, or **parent**, or **base** class

-When a derived class is created it actually contains all the data (instance variables) and methods from the parent

-These are all included automatically and become part of the derived class

-They are included even though we don’t actually “see” them listed in the derived class definition!



Inheritance

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

Vehicle
Car

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

-As we mentioned, inheriting classes from existing ones models what we see in the real world

-In the real world, we see many examples of similarly related models (e.g. animals, vehicles, people)

-By looking at similar traits and behaviors, we can classify models into classification systems

-As the textbook mentions, the word **class** actually comes from this classification idea

-In object-oriented design, inheritance models this classification system we see in the real world

-The text describes an excellent example of such a classification system involving mammals

-We can group **general** attributes and behaviors we see in all mammals and describe them in a *Mammal* class

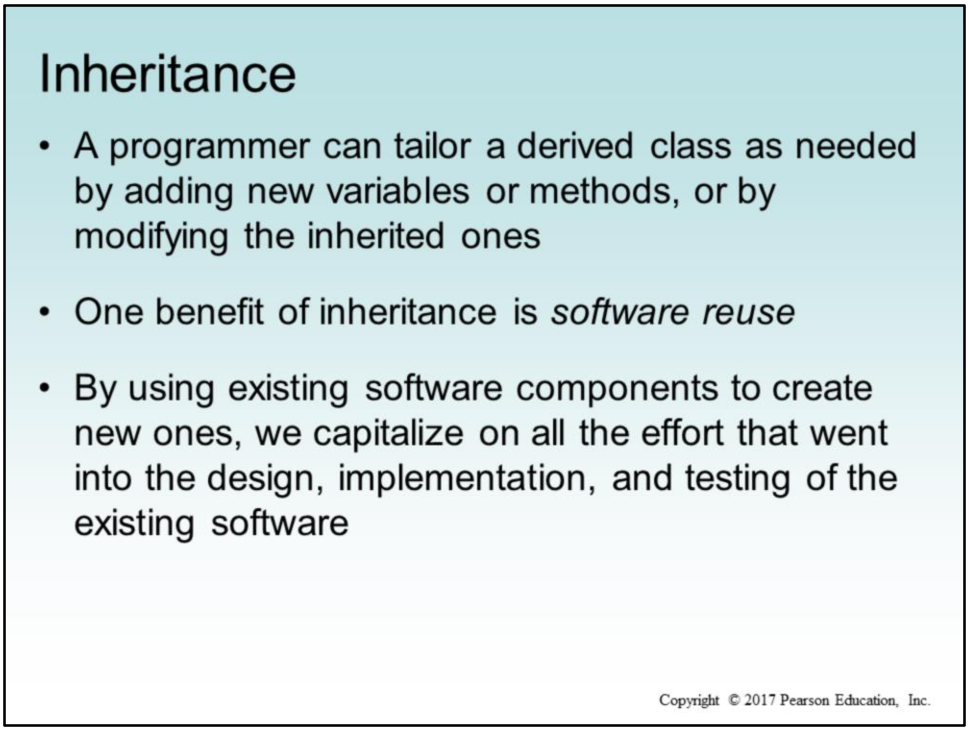
-We can then create a new **type of** mammal, such as a horse, by inheriting from the *Mammal* class

-In our new *Horse* class, we then add to this *Mammal* class what is **specific** to a horse

-Notice the use of the phrase “type of” in our design of the *Horse* class

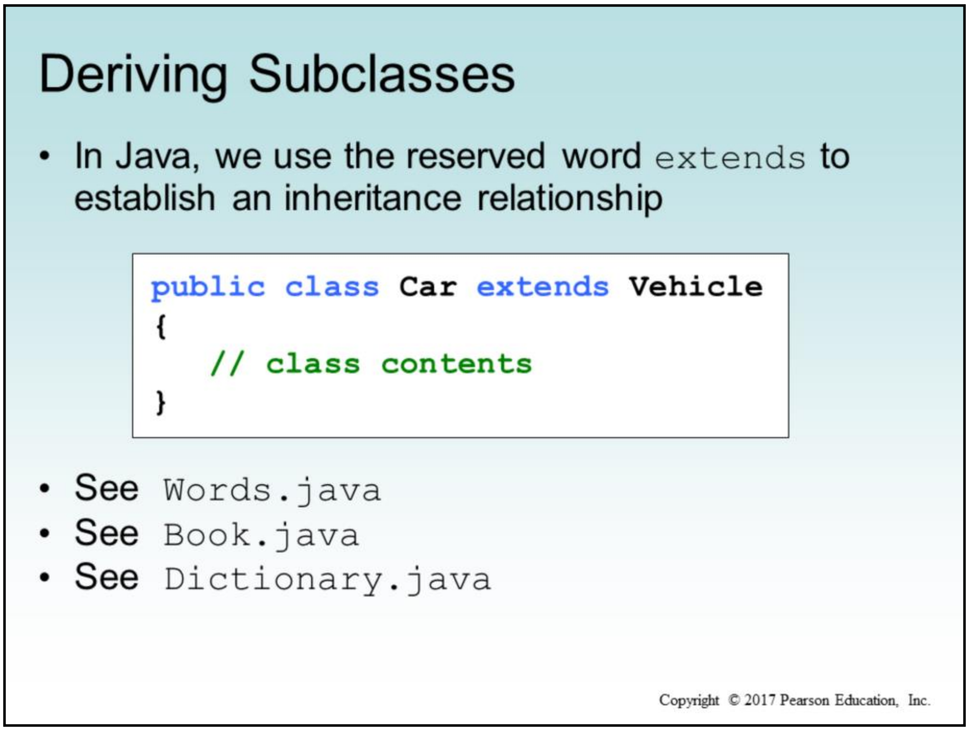
-Inheritance is used anytime we see a relationship where one class is a “type of” another

-The phrase “is a” is also used to describe this relationship (e.g. a horse “is a” mammal)



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`

-It is a very simple process to create a new class from an existing one

-When we write the new class, we use the keyword **extends** in our class description

-In this example, the new class (*Car*) is inherited from the existing class (*Vehicle*)

-*Car* is the derived class, or subclass, or child class

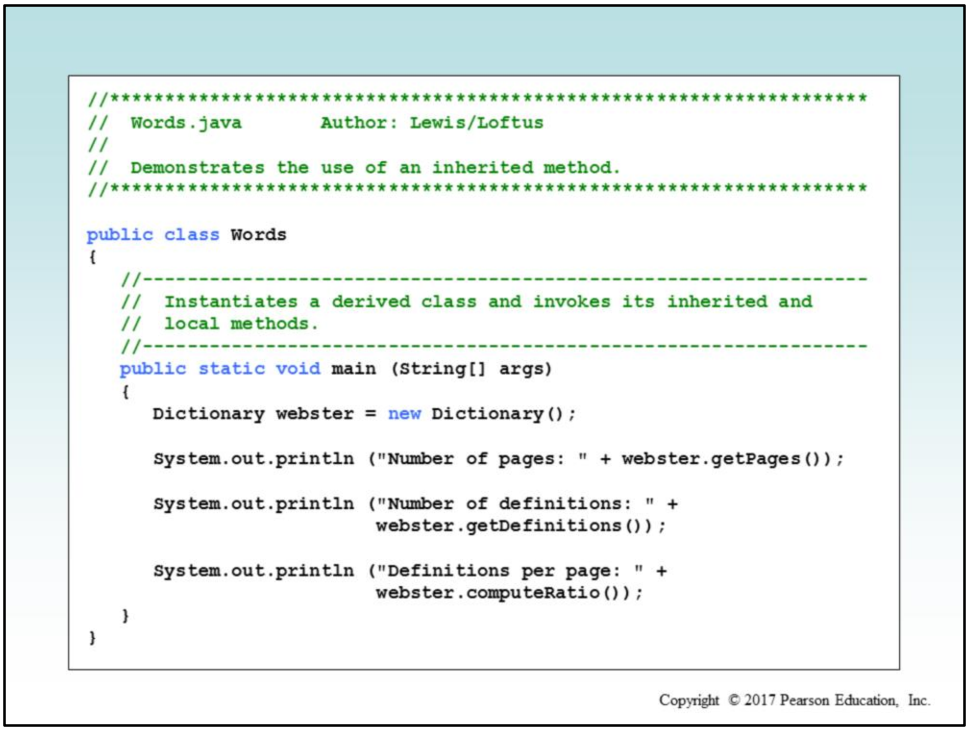
-*Vehicle* is the base class, or superclass, or parent class

-Once derived, the Car class actually has all of the variables and methods from the Vehicle class

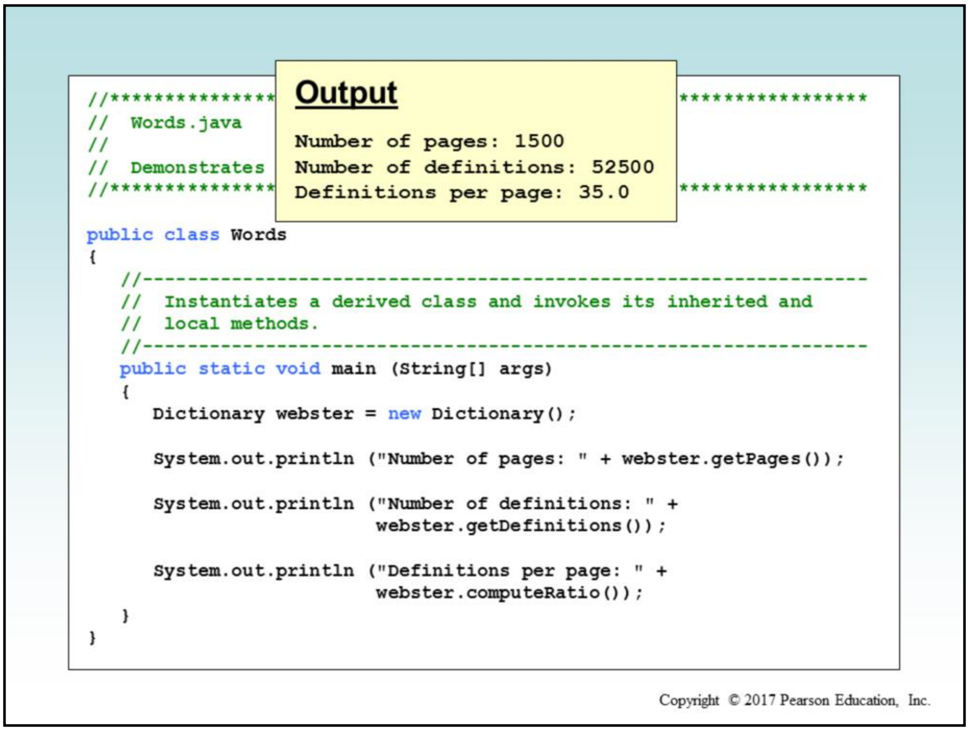
-This can be confusing because we don't actually see them listed in the Car class

-We can refer to the data and methods in the Vehicle class as if they were in the Car class!

-We'll see, however, that visibility modifiers in the parent class can affect what we can use in the Car class



```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());
   }
}
```

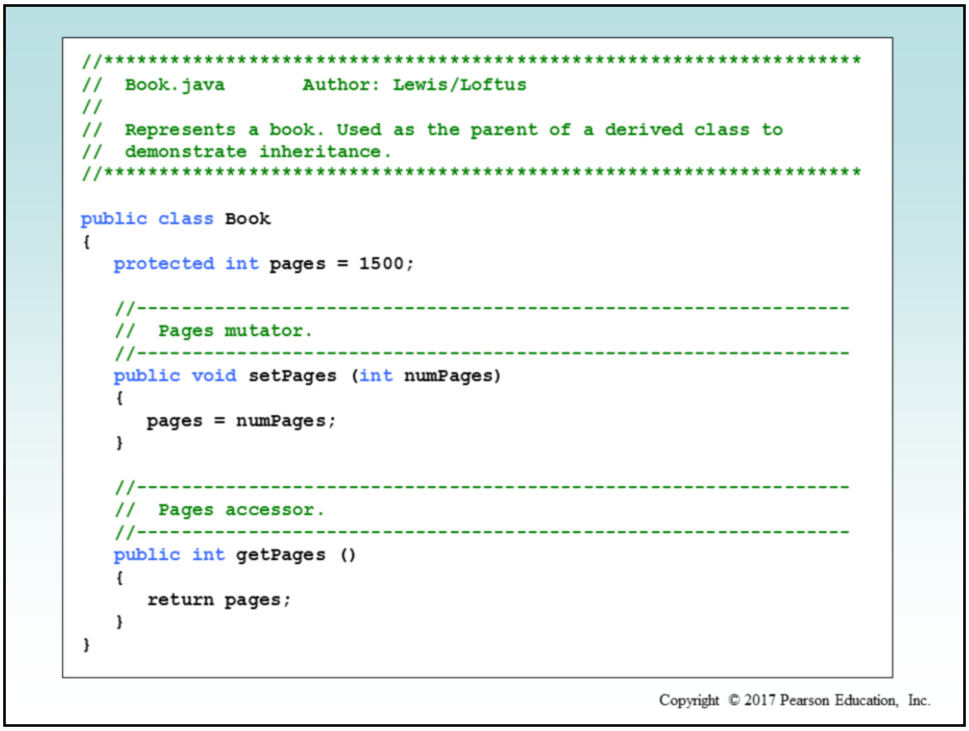


```
//***************                                     *****************
//  Words.java
//
//  Demonstrates
//***************                                     *****************

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());
   }
}
```

Output

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

-Note how the derived Dictionary object can call methods (getPages) from the base Book class



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

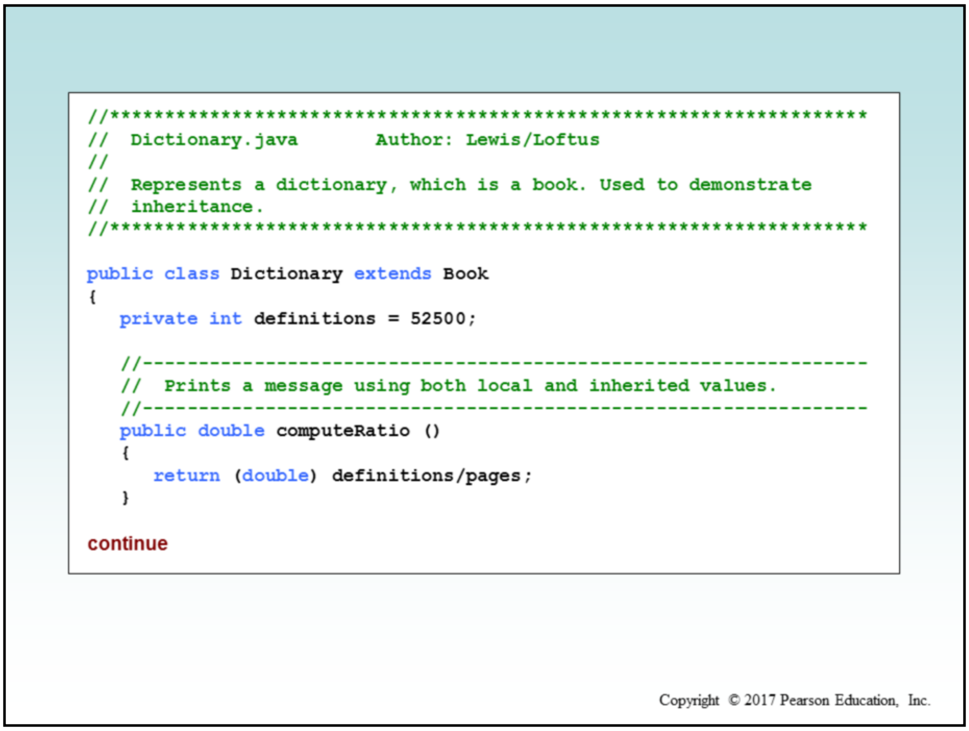
-Note the new **protected** visibility modifier on the instance variable

-We'll see that this modifier allows access to child classes (classes derived from the class)

-It does **not**, however, allow access to any other classes outside the containing package

-In this way, protected provides a type of visibility (or access) this is in between private and public

-It allows the class itself and derived classes access, but not other classes outside the containing package

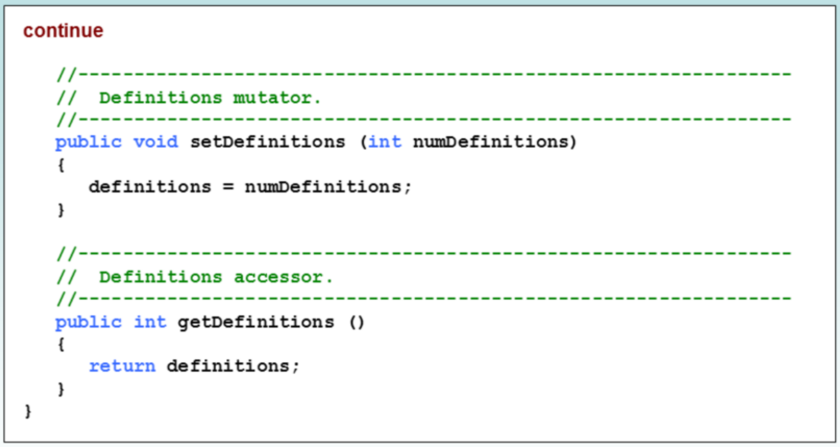


```
//**********************************************************************
//  Dictionary.java       Author: Lewis/Loftus
//
//  Represents a dictionary, which is a book. Used to demonstrate
//  inheritance.
//**********************************************************************

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

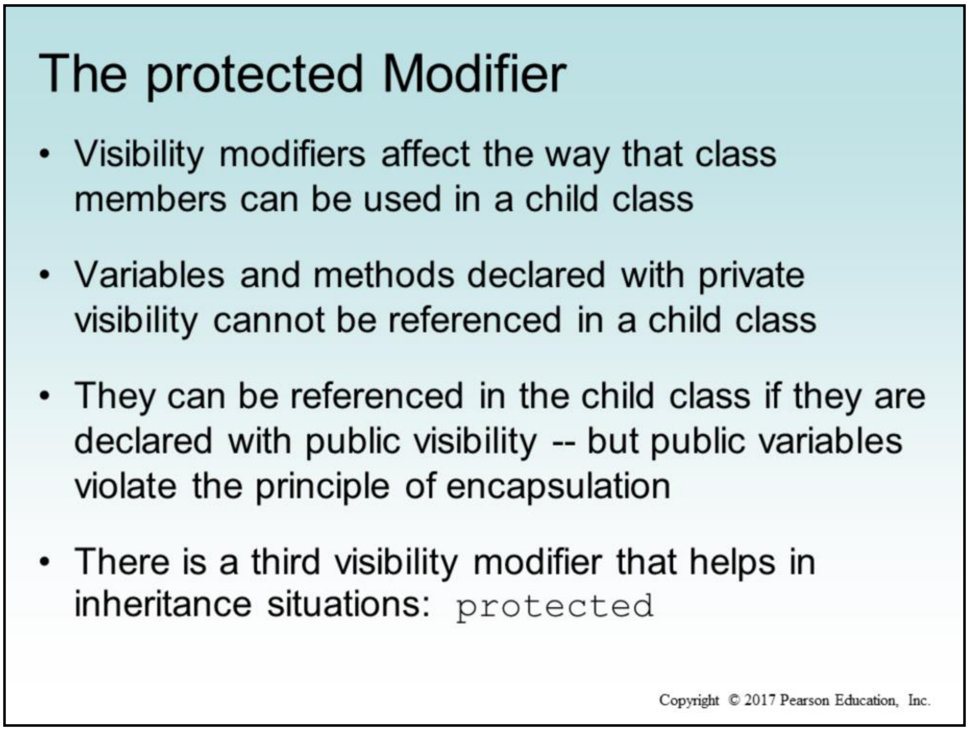
-Note how the derived class (Dictionary) can use the instance variable (pages) from the parent class in its method

-It is as if this pages instance variable is part of the Dictionary class (even though we don't actually see it listed)



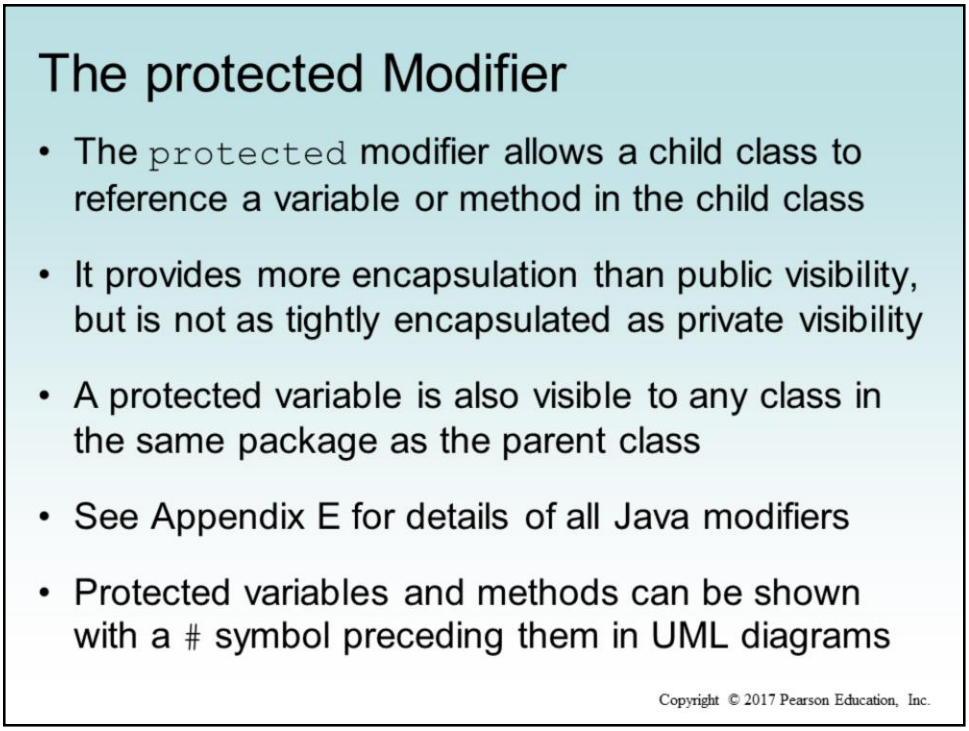
continue

```
   //-----------------------------------------------------------------
   //  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`

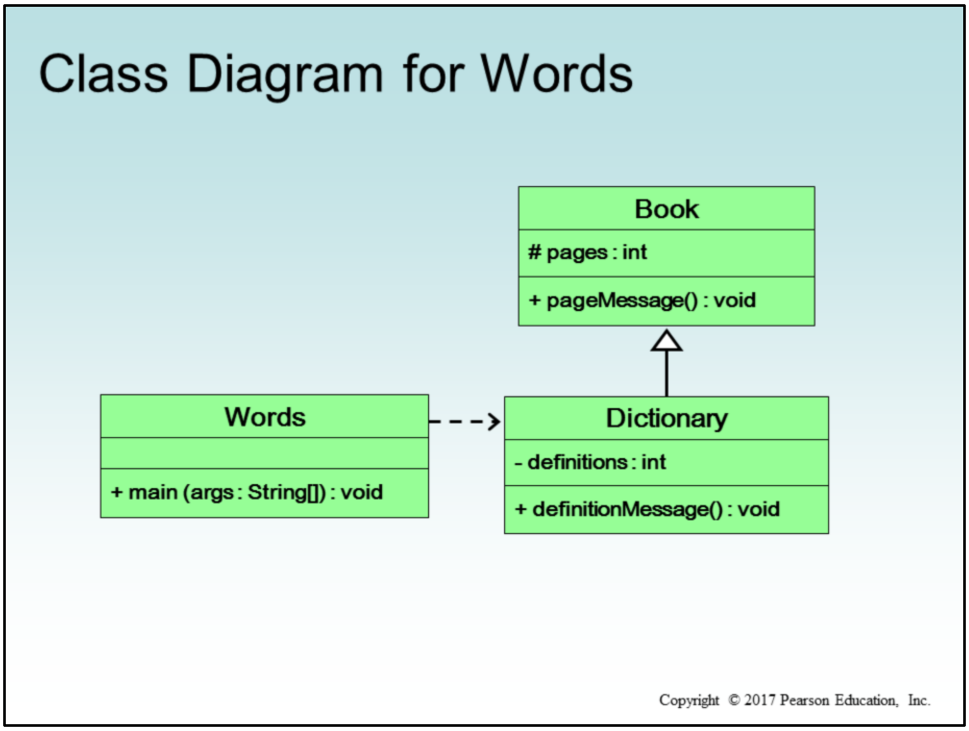


The protected Modifier

- 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
- See Appendix E for details of all Java modifiers
- Protected variables and methods can be shown with a # symbol preceding them in UML diagrams

-Note in Appendix E the visibility for other classes contained in the same **package**

-For example, classes in the same package, as well as derived classes, can access protected variables/methods



Class Diagram for Words

Book

pages : int

+ pageMessage() : void

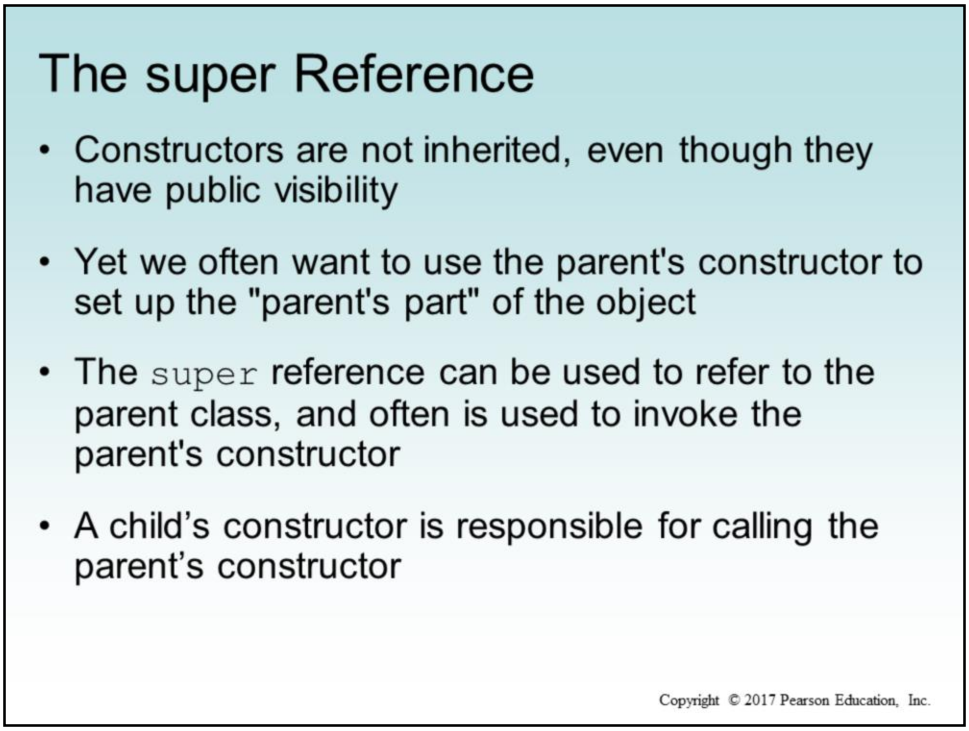
Words

+ main (args : String[]) : void

Dictionary

- definitions : int

+ definitionMessage() : void



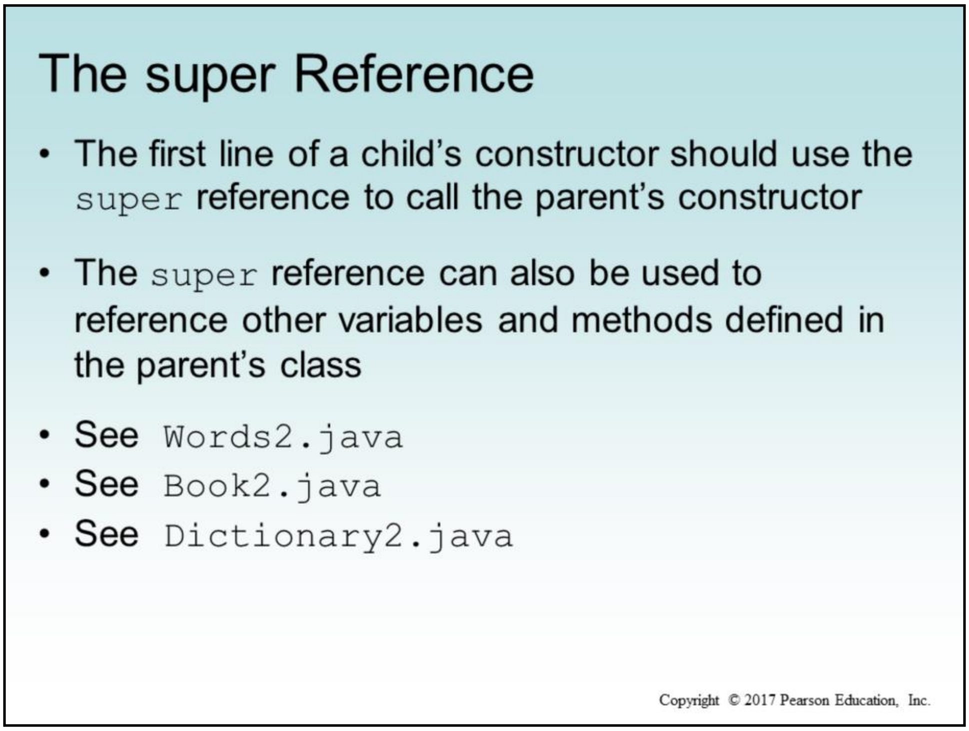
The super Reference

- 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 reserved word **super** is used to refer to the parent class

-Just as we use the **this** keyword to refer to the class itself, **super** can be used to refer to the parent

-This keyword is most often used to call a parent class constructor within a derived class 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`

-This example demonstrates how to call a parent constructor in the derived constructor class

-Note that in the preceding example we were not required to do this because we there wasn't a parent constructor

-(Recall that we can write a class without a constructor)

(Case 1)

-If we don't have a parent constructor **OR** the parent constructor **does not** have arguments,

- then the parent constructor is called **automatically** when the derived constructor is called

(Case 2)

-If we have a parent constructor that **does** have arguments,

- then we need to explicitly call the parent constructor with its arguments using the super keyword

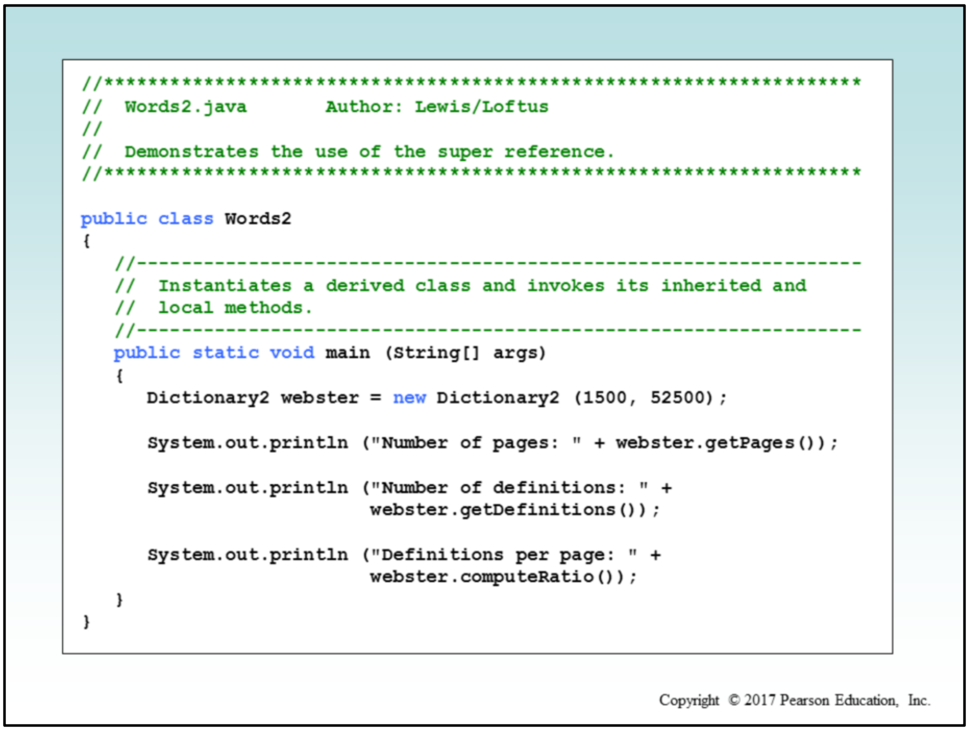
(Case 3)

-If we have a parent constructor that **does** have arguments and we don't call it explicitly in the child constructor,

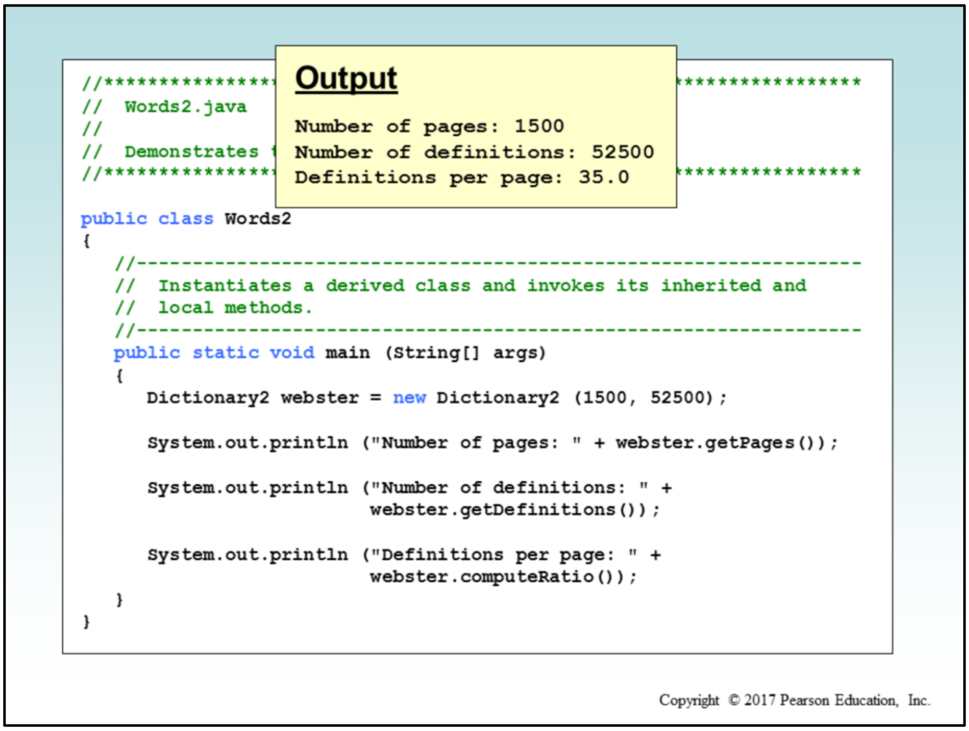
- then Java will automatically call the parent constructor super() (without arguments)

-The previous example in the text and slides demonstrated the first case above

-This next example in the text and slides demonstrates the second case above



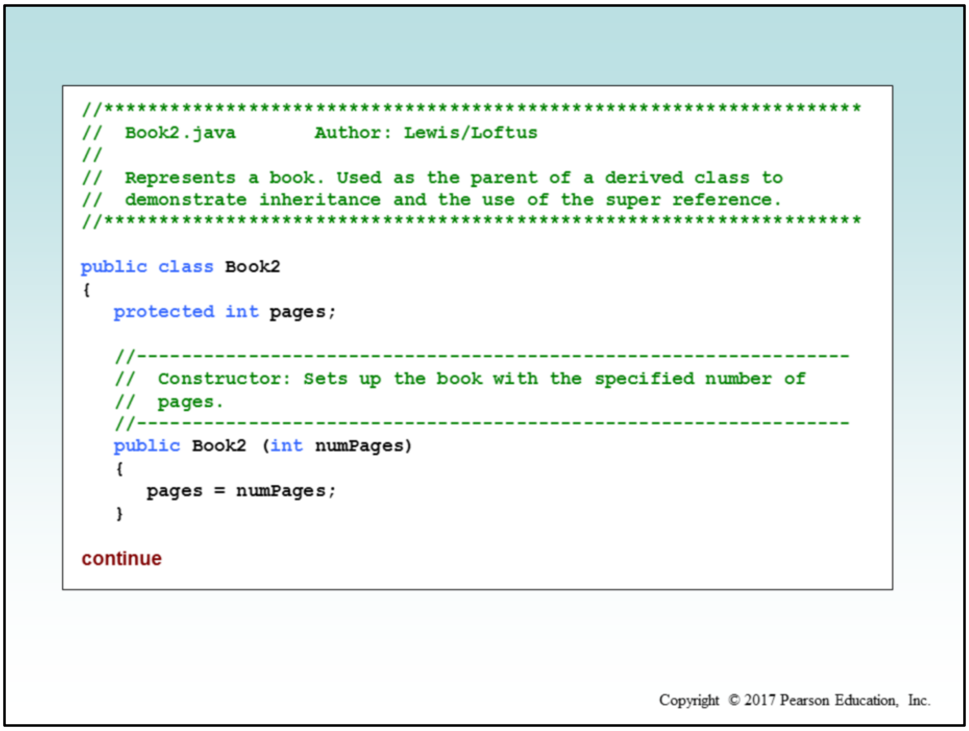
```
//********************************************************************
//  Words2.java       Author: Lewis/Loftus
//
//  Demonstrates the use of the super reference.
//********************************************************************

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());
   }
}
```

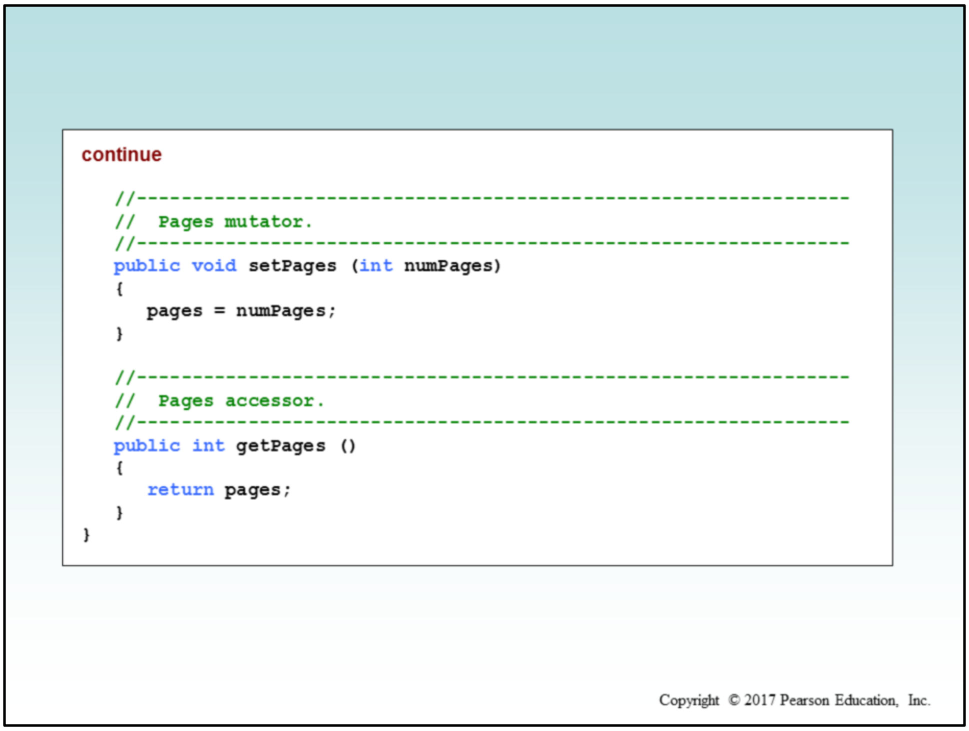
Output

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



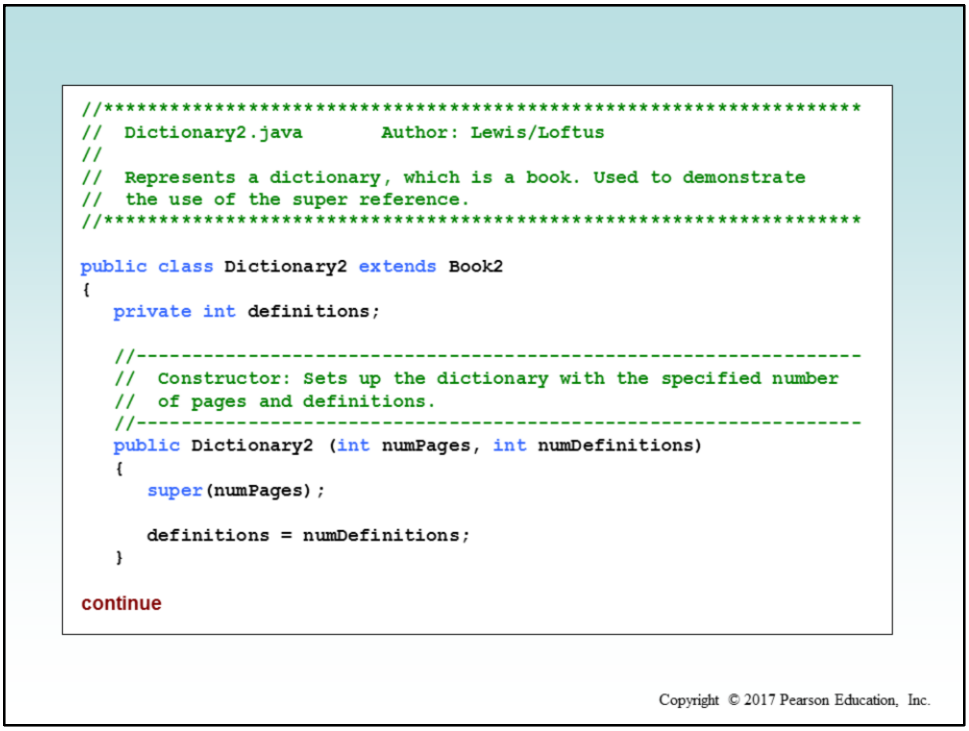
```java
//********************************************************************
//  Book2.java       Author: Lewis/Loftus
//
//  Represents a book. Used as the parent of a derived class to
//  demonstrate inheritance and the use of the super reference.
//********************************************************************

public class Book2
{
   protected int pages;

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

continue



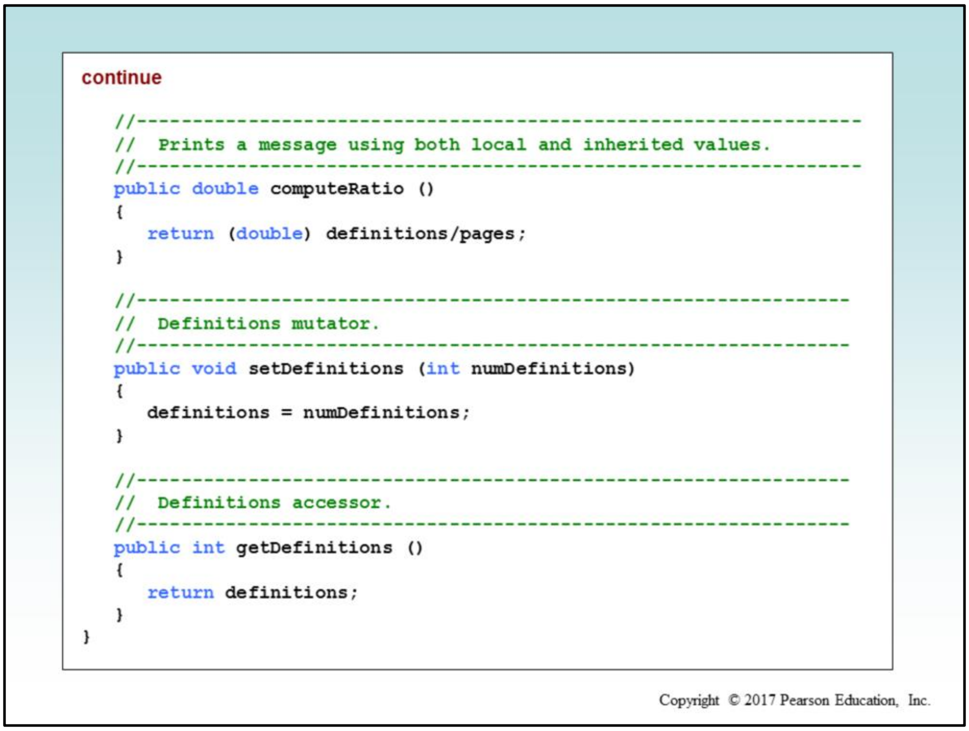
continue

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

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



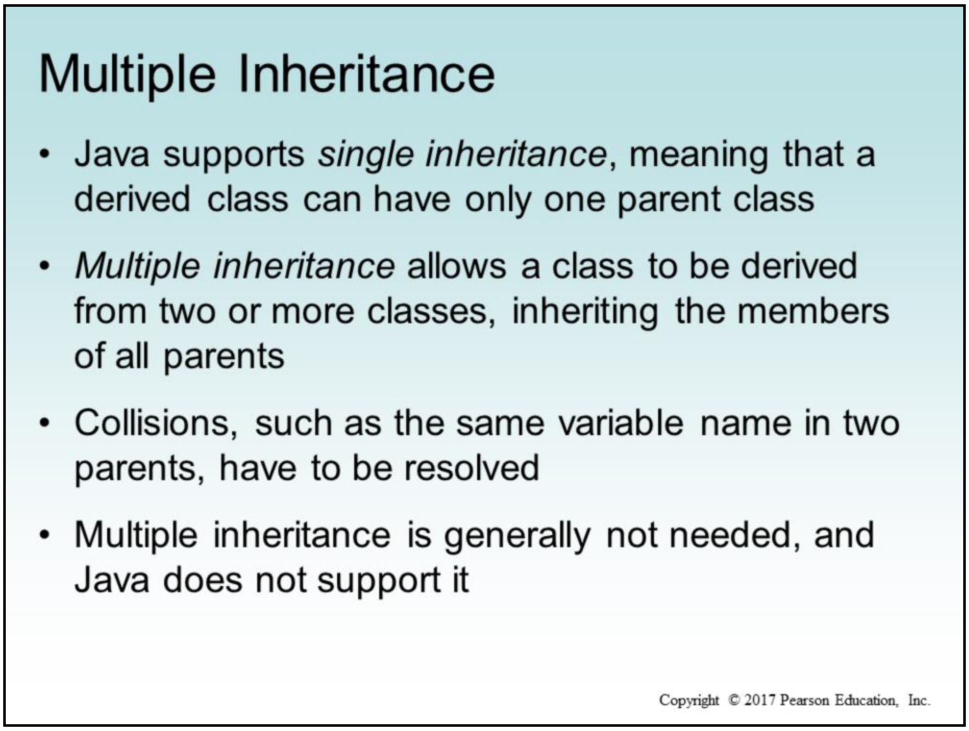
```
//**********************************************************************
//  Dictionary2.java       Author: Lewis/Loftus
//
//  Represents a dictionary, which is a book. Used to demonstrate
//  the use of the super reference.
//**********************************************************************

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

-Note in Java, classes can **only** be derived from a **single** parent!

-Multiple inheritance is NOT allowed in Java (but is allowed in C++)

-As we'll see in advanced studies, using Java interfaces provides a type of multiple inheritance in Java