Java – Generics, Interface, and Inheritance

The ArrayList Class

- The `ArrayList` class is part of the `java.util` package
- Like an array, it can store a list of values and reference each one using a numeric index
- However, you cannot use the bracket syntax with an `ArrayList` object
- Furthermore, an `ArrayList` object grows and shrinks as needed, adjusting its capacity as necessary

The ArrayList Class

- An `ArrayList` object is created as follows:

```
ArrayList band = new ArrayList();
```

- A list of methods supported by `ArrayList` class is given in Chapter 7 of the text book. Some examples of methods:

```
void add(int index, Object obj)

Object get(int index)
```

- Elements can be inserted or removed with a single method invocation. For example:

```
band.add(2,"Paul");

bandMember = band.get(1);
```

The ArrayList Class

- When an element is inserted, the other elements "move aside" to make room
- Likewise, when an element is removed, the list "collapses" to close the gap
- The indexes of the elements adjust accordingly

The ArrayList Class

- An `ArrayList` stores references to the `Object` class, which allows it to store any kind of object
- We can also define an `ArrayList` object to accept a particular type of object
- The following declaration creates an `ArrayList` object that only stores `Family` objects

```
ArrayList<Family> reunion = new ArrayList<Family>();
```

- This is an example of *generics* and we can call `ArrayList` a *generic type*

ArrayList Efficiency

- The `ArrayList` class is implemented using an underlying array
- The array is manipulated so that indexes remain continuous as elements are added or removed
- If elements are added to and removed from the end of the list, this processing is fairly efficient
- But as elements are inserted and removed from the front or middle of the list, the remaining elements are shifted

Collection Classes

- The Java standard library contains several classes that represent collections, often referred to as the *Java Collections API*
- Their underlying implementation is implied in the class names such as `ArrayList` and `LinkedList`
- The classes are implemented as generic types
- It means that the type of object can be established when an object of that collection type is instantiated.

Generic Types

- A class can be defined to operate on a generic data type which is specified when the class is instantiated:

```
LinkedList<Book> myList = new LinkedList<Book>();
```

- By specifying the type stored in a collection, only objects of that type can be added to it

- Furthermore, when an object is removed, its type is already established

Interfaces

- A Java *interface* is a collection of abstract methods and constants
- An *abstract method* is a method header without a method body
- An abstract method can be declared using the modifier `abstract`, but because all methods in an interface are abstract, usually it is left off
- An interface is used to establish a set of methods that a class will implement

Interfaces

interface is a reserved word

None of the methods in an interface are given a definition (body)

```
public interface Doable
{
   public void doThis();
   public int doThat();
   public void doThis2 (float value, char ch);
   public boolean doTheOther (int num);
}
```

A semicolon immediately follows each method header

Interfaces

- An interface cannot be instantiated
- Methods in an interface have public visibility by default
- A class formally implements an interface by:
  - stating so in the class header
  - providing implementations for each abstract method in the interface
- If a class asserts that it implements an interface, it *must* define all methods in the interface

Interfaces

```
public class CanDo implements Doable
{
   public void doThis ()
   {
      // whatever
   }

   public void doThat ()
   {
      // whatever
   }

   // etc.
}
```

implements is a reserved word

Each method listed in Doable is given a definition

Interfaces

- A class that implements an interface can implement other methods as well

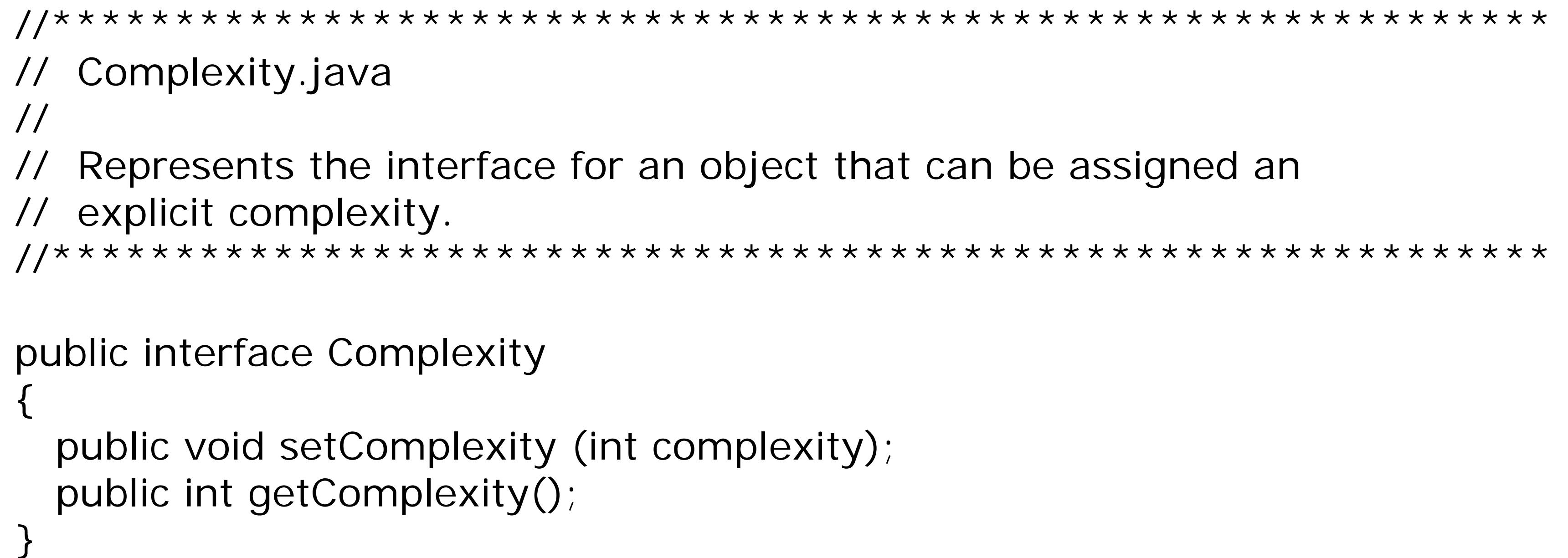
  See Complexity.java

  See Question.java

  See MiniQuiz.java

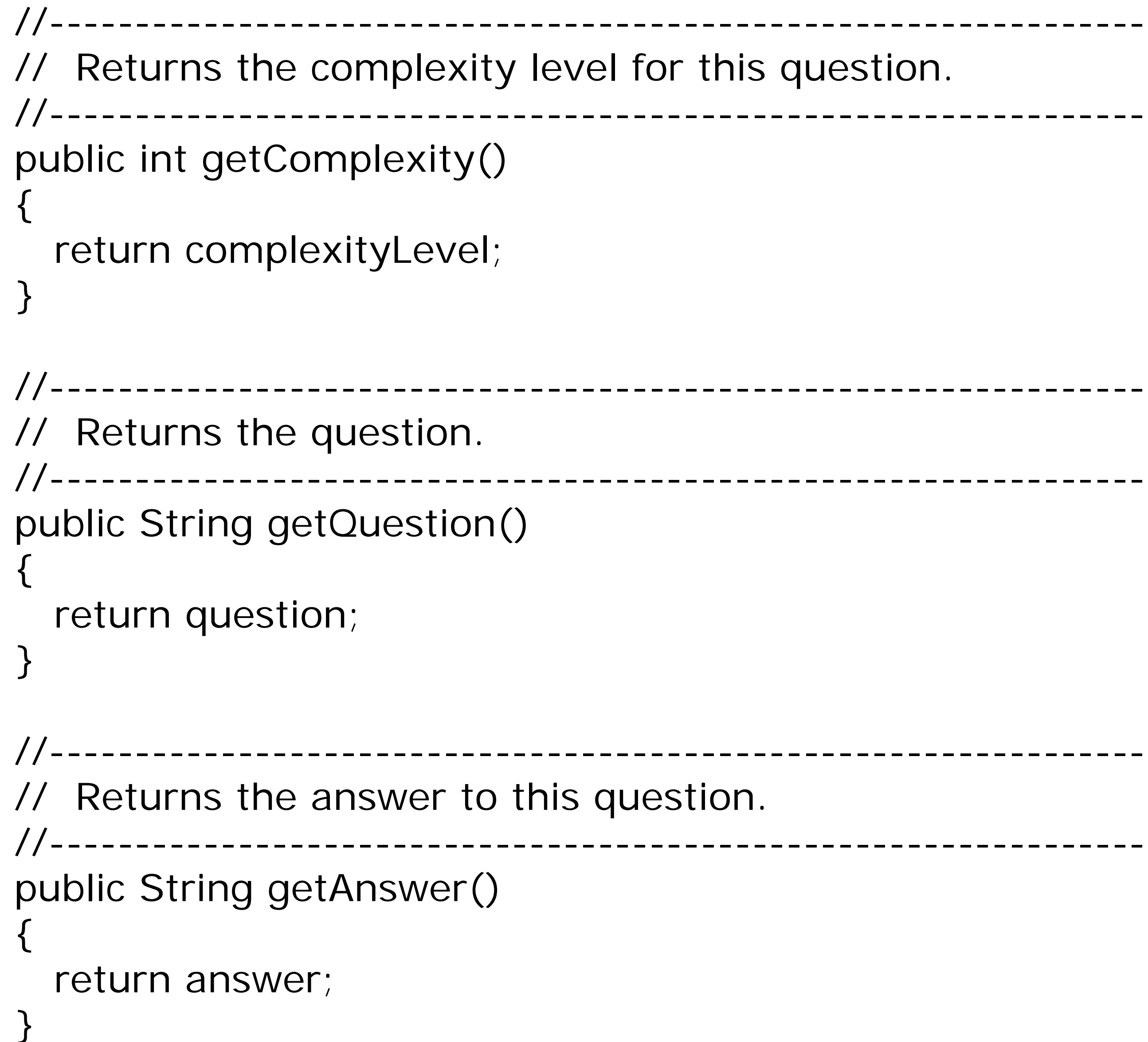
- In addition to (or instead of) abstract methods, an interface can contain constants

- When a class implements an interface, it gains access to all its constants

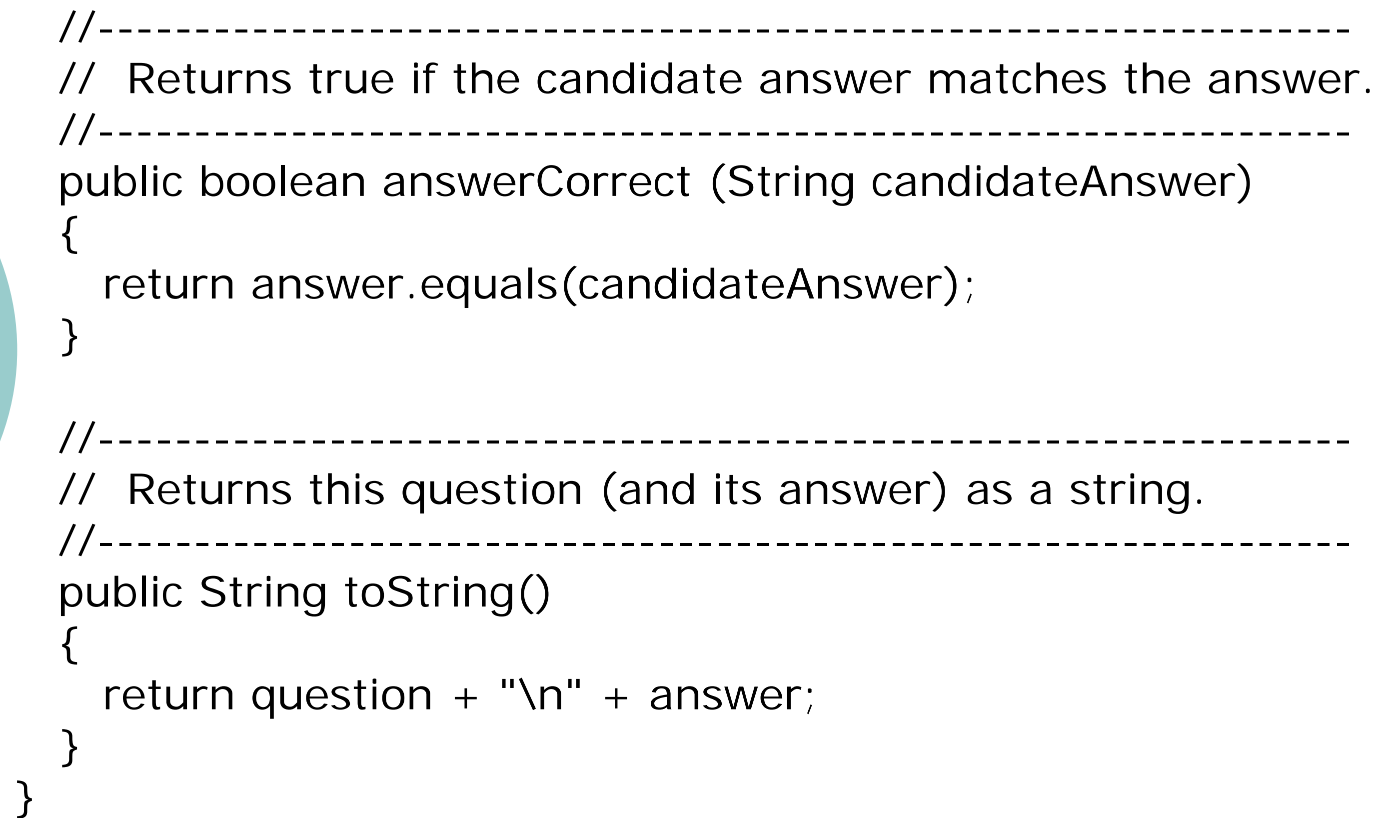


```java
//*************************************************************
//  Complexity.java
//
//  Represents the interface for an object that can be assigned an
//  explicit complexity.
//*************************************************************

public interface Complexity
{
   public void setComplexity (int complexity);
   public int getComplexity();
}
```

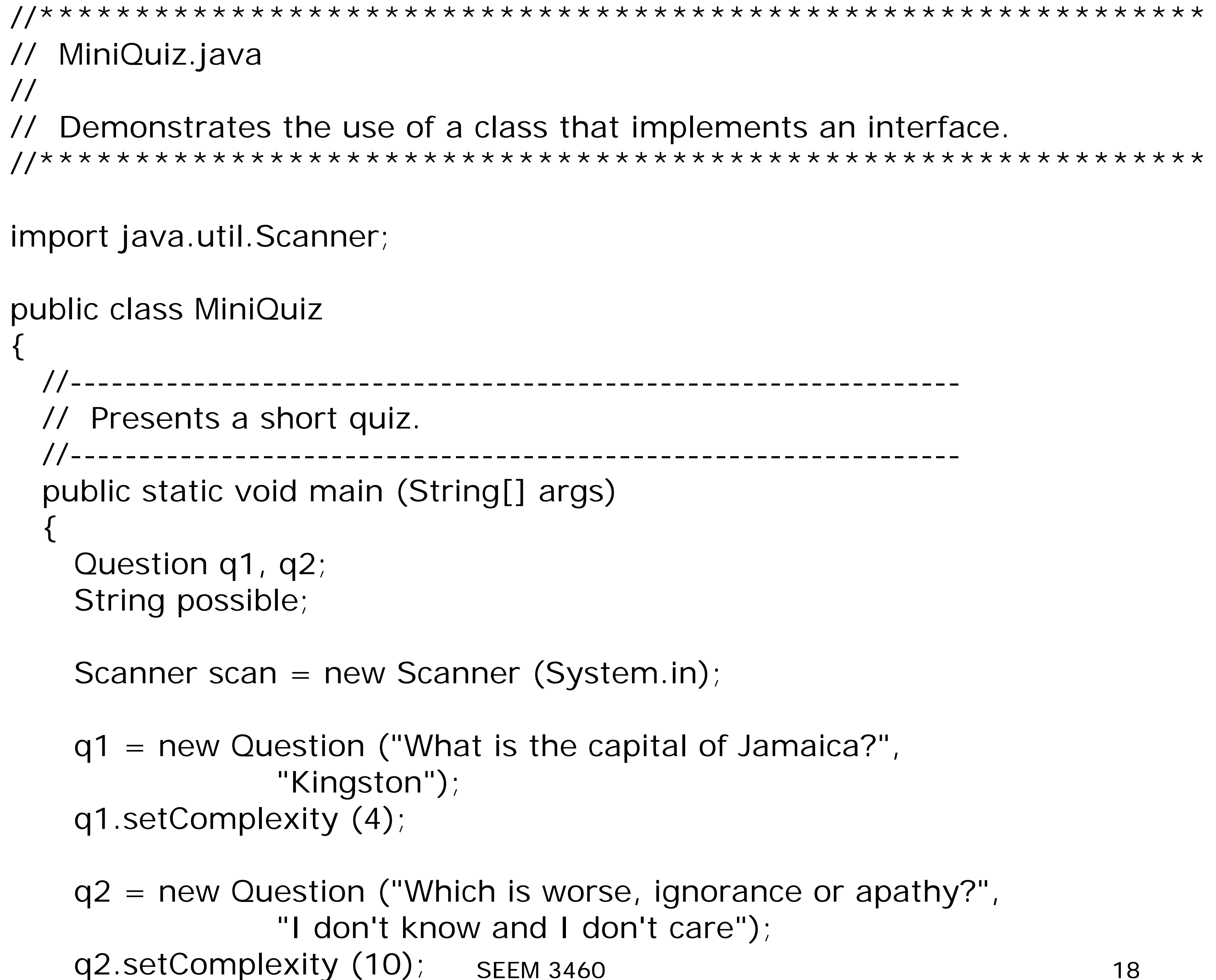
```
//********************************************************************
//  Question.java
//
//  Represents a question (and its answer).
//********************************************************************
public class Question implements Complexity
{
   private String question, answer;
   private int complexityLevel;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the question with a default complexity.
   //-----------------------------------------------------------------
   public Question (String query, String result)
   {
      question = query;
      answer = result;
      complexityLevel = 1;
   }

   //-----------------------------------------------------------------
   //  Sets the complexity level for this question.
   //-----------------------------------------------------------------
   public void setComplexity (int level)
   {
      complexityLevel = level;
   }
```



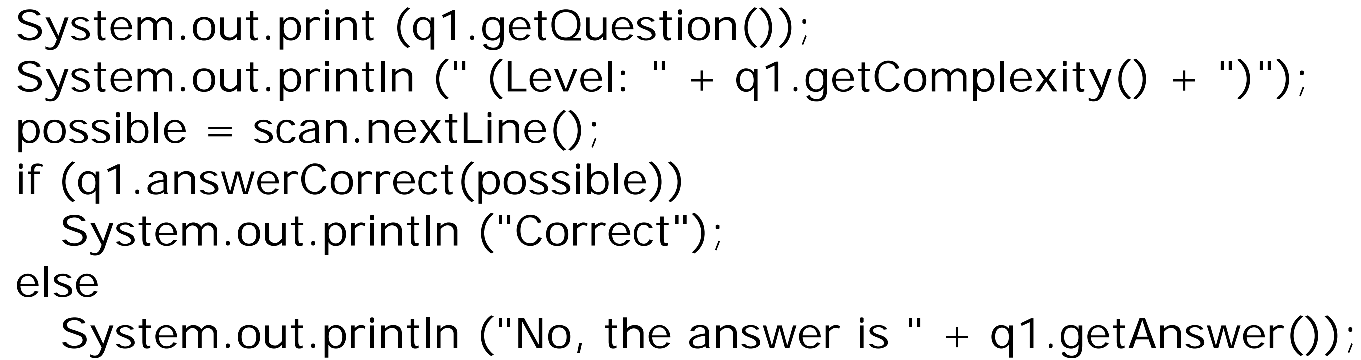
```
//-----------------------------------------------------------------
//  Returns the complexity level for this question.
//-----------------------------------------------------------------
public int getComplexity()
{
   return complexityLevel;
}

//-----------------------------------------------------------------
//  Returns the question.
//-----------------------------------------------------------------
public String getQuestion()
{
   return question;
}

//-----------------------------------------------------------------
//  Returns the answer to this question.
//-----------------------------------------------------------------
public String getAnswer()
{
   return answer;
}
```



```
//-------------------------------------------------------------
//  Returns true if the candidate answer matches the answer.
//-------------------------------------------------------------
public boolean answerCorrect (String candidateAnswer)
{
   return answer.equals(candidateAnswer);
}

//-------------------------------------------------------------
//  Returns this question (and its answer) as a string.
//-------------------------------------------------------------
public String toString()
{
   return question + "\n" + answer;
}
}
```



```java
//********************************************************************
//  MiniQuiz.java
//
//  Demonstrates the use of a class that implements an interface.
//********************************************************************

import java.util.Scanner;

public class MiniQuiz
{
   //-----------------------------------------------------------------
   //  Presents a short quiz.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Question q1, q2;
      String possible;

      Scanner scan = new Scanner (System.in);

      q1 = new Question ("What is the capital of Jamaica?",
                         "Kingston");
      q1.setComplexity (4);

      q2 = new Question ("Which is worse, ignorance or apathy?",
                         "I don't know and I don't care");
      q2.setComplexity (10);
```



```
    System.out.print (q1.getQuestion());
    System.out.println (" (Level: " + q1.getComplexity() + ")");
    possible = scan.nextLine();
    if (q1.answerCorrect(possible))
      System.out.println ("Correct");
    else
      System.out.println ("No, the answer is " + q1.getAnswer());

    System.out.println();
    System.out.print (q2.getQuestion());
    System.out.println (" (Level: " + q2.getComplexity() + ")");
    possible = scan.nextLine();
    if (q2.answerCorrect(possible))
      System.out.println ("Correct");
    else
      System.out.println ("No, the answer is " + q2.getAnswer());
  }
}
```

MiniQuiz.java - Sample Execution

- The following is a sample execution of MiniQuiz.class

cuse93> *java MiniQuiz*
What is the capital of Jamaica? (Level: 4)
Kingston
Correct

Which is worse, ignorance or apathy? (Level: 10)
sorry i dont know
No, the answer is I don't know and I don't care

Interfaces

- A class can implement multiple interfaces
- The interfaces are listed in the `implements` clause
- The class must implement all methods in all interfaces listed in the header

```
class ManyThings implements interface1, interface2
{
   // all methods of both interfaces
}
```

Interfaces

- The Java standard class library contains many helpful interfaces
- The `Comparable` interface contains one abstract method called `compareTo`, which is used to compare two objects
- The `String` class implements `Comparable`, giving us the ability to put strings in lexicographic order

- Any class can implement `Comparable` to provide a mechanism for comparing objects of that type

```
if (obj1.compareTo(obj2) < 0)
   System.out.println ("obj1 is less than obj2");
```

The value returned from `compareTo` should be negative is `obj1` is less that `obj2`, 0 if they are equal, and positive if `obj1` is greater than `obj2`

When a programmer designs a class that implements the `Comparable` interface, it should follow this intent

The Comparable Interface

- It's up to the programmer to determine what makes one object less than another
- For example, you may define the `compareTo` method of an `Employee` class to order employees by name (alphabetically) or by employee number
- The implementation of the method can be as straightforward or as complex as needed for the situation

Interfaces

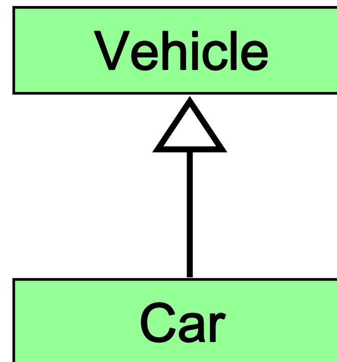
- You could write a class that implements certain methods (such as `compareTo`) without formally implementing the interface (`Comparable`)
- However, formally establishing the relationship between a class and an interface allows Java to deal with an object in certain ways
- Interfaces are a key aspect of object-oriented design in Java

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
- *Software reuse* is a fundamental benefit of inheritance
- 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

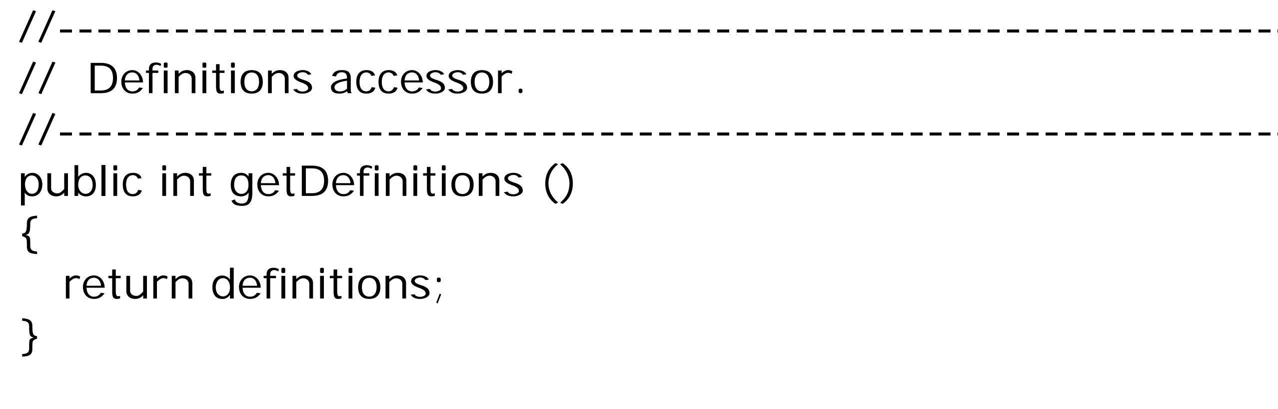
```
class Car extends Vehicle
{
   // class contents
}
```

See Words.java
See Book.java
See Dictionary.java

```
//********************************************************************
//  Words.java
//
//  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());
   }
}
```

```
//********************************************************************
//  Book.java
//
//  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
//
//  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 definitions/pages;
   }

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



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

- The following is a sample execution of Words.class

```
cuse93> java Words
Number of pages: 1500
Number of definitions: 52500
Definitions per page: 35.0
```

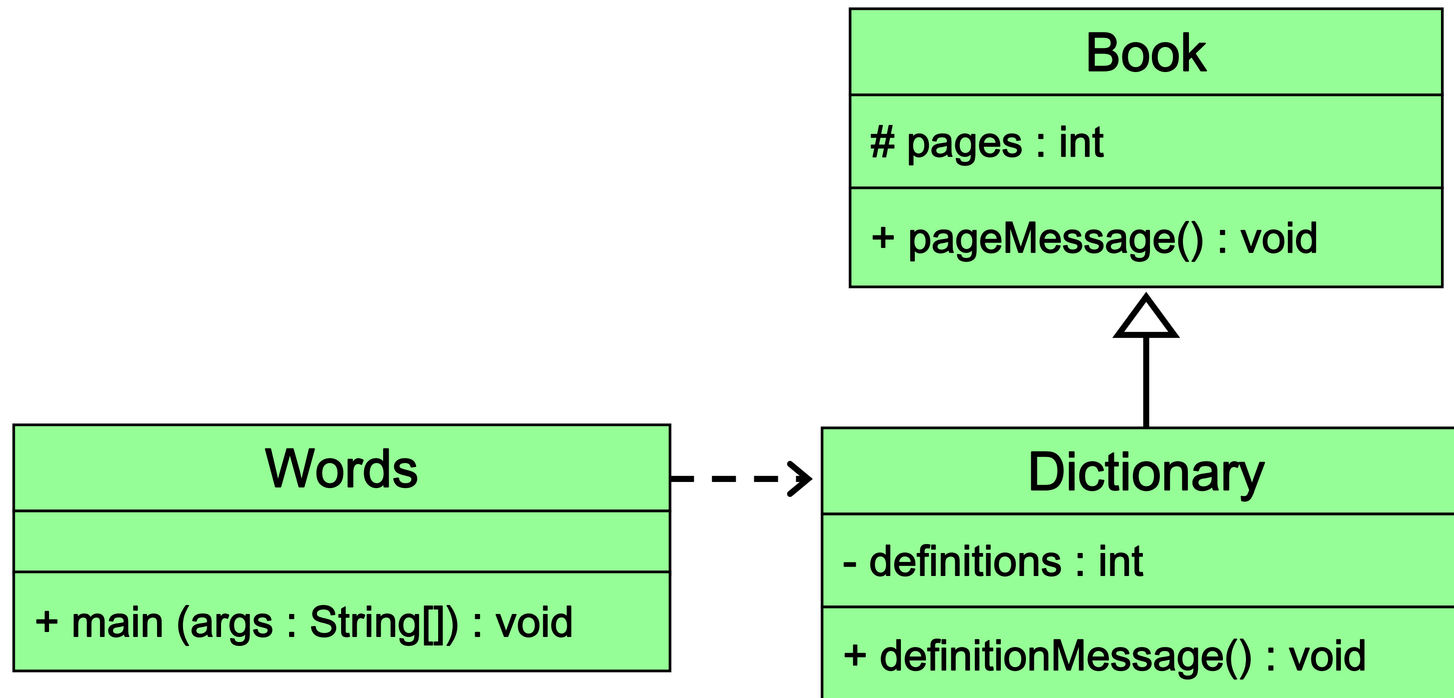
The protected Modifier

- 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 by name 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 directly in the child class
- It provides more encapsulation than public visibility, but is not as tightly encapsulated as private visibility
- A protected variable is visible to any class in the same package as the parent class
- The details of all Java modifiers are discussed in Appendix E of the text book
- Protected variables and methods can be shown with a `#` symbol preceding them in UML diagrams

Class Diagram for Words



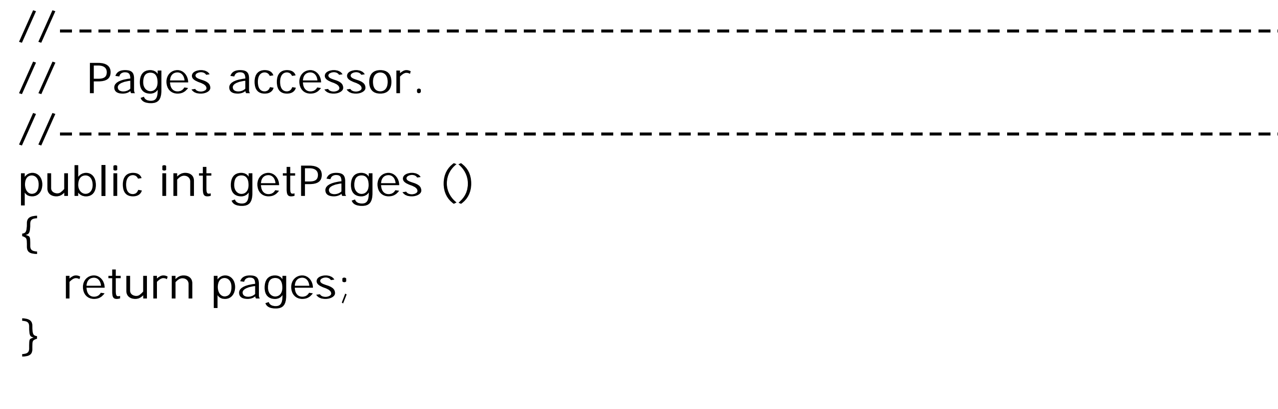
The super Reference

- 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

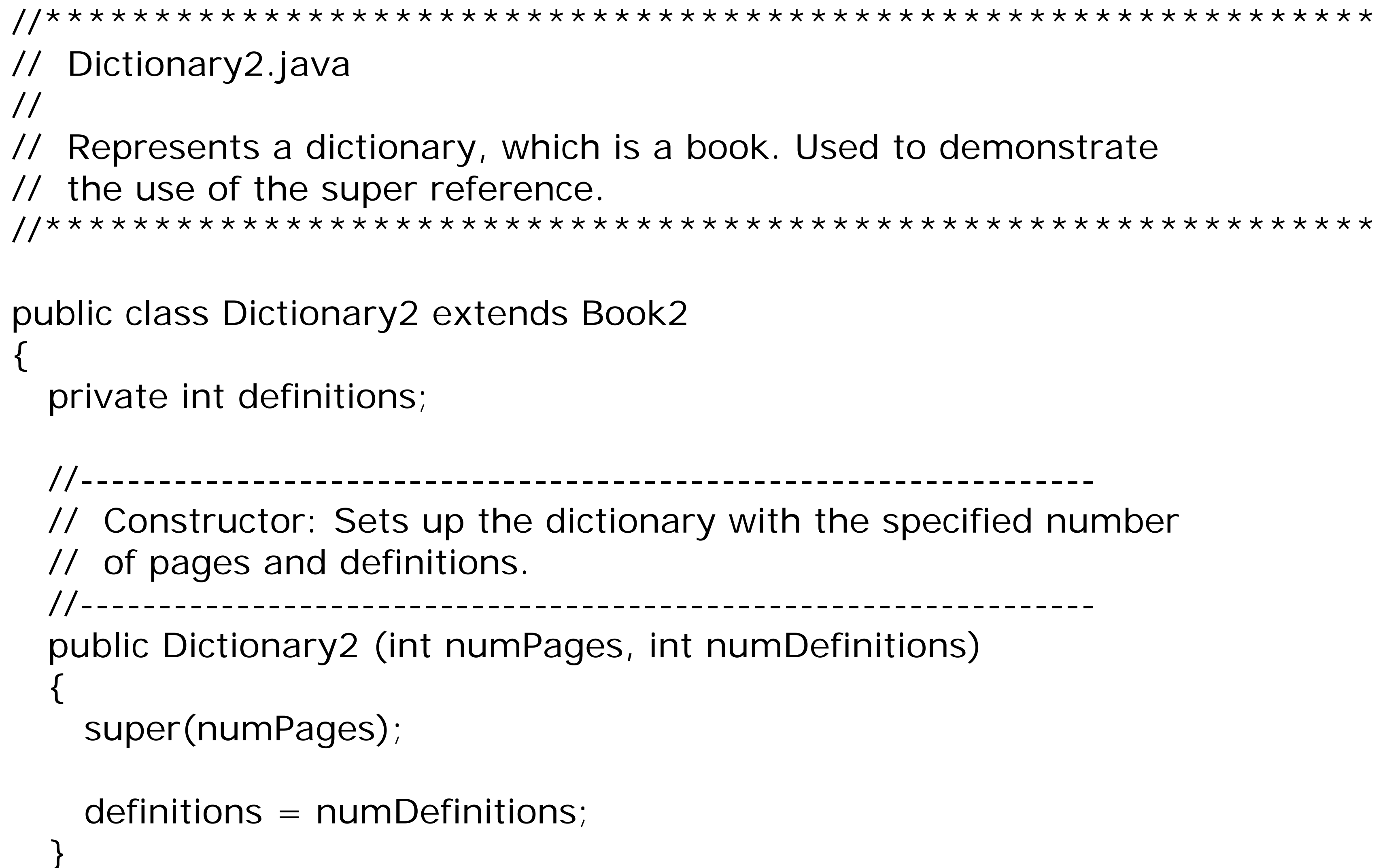
See `Words2.java`
See `Book2.java`
See `Dictionary2.java`

```
//********************************************************************
//  Words2.java
//
//  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());
   }
}
```

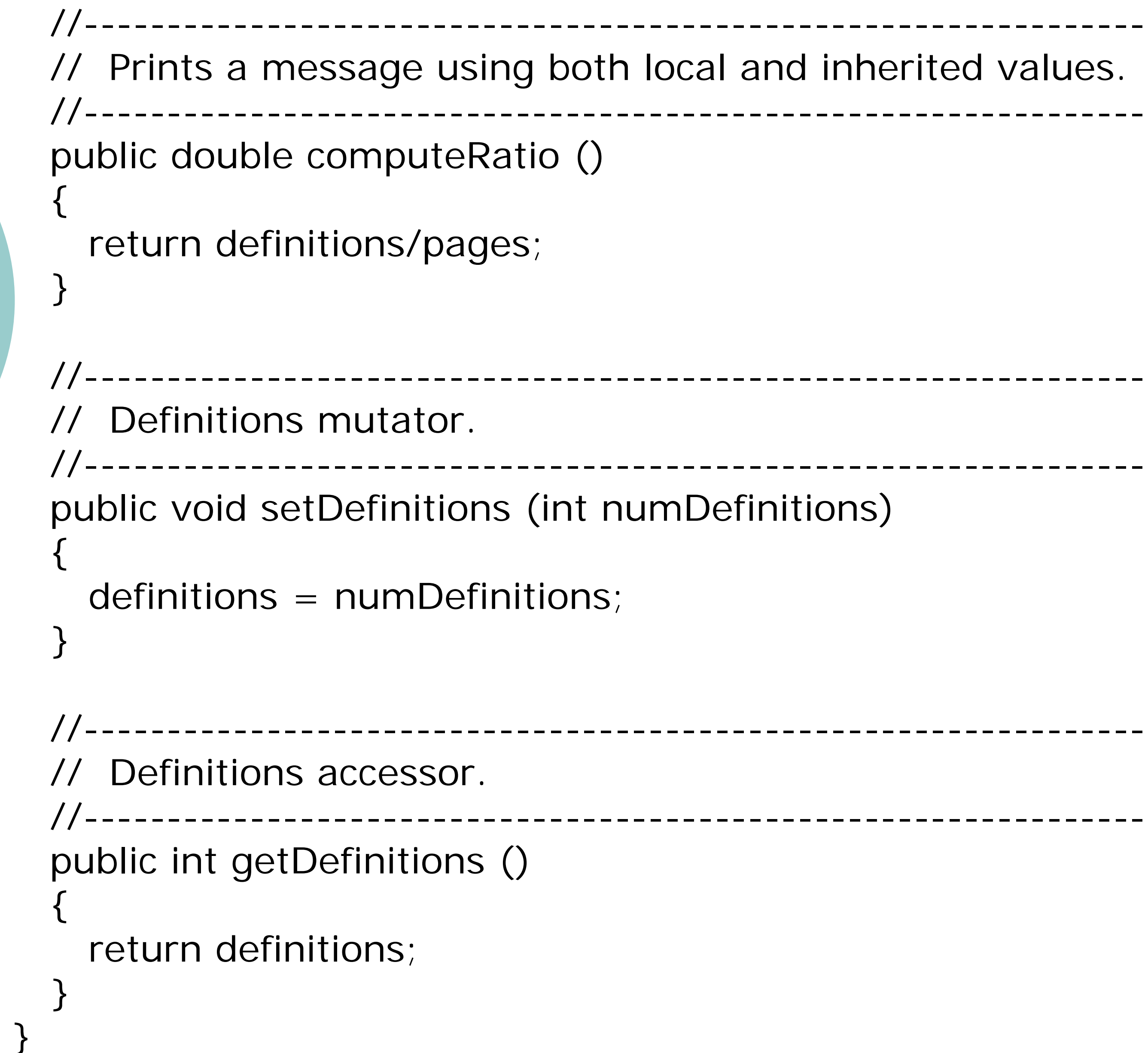
```
//*************************************************************
//  Book2.java
//
//  Represents a book. Used as the parent of a derived class to
//  demonstrate inheritance and the use of the super reference.
//*************************************************************

public class Book2
{
   protected int pages;

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

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



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



```
//************************************************************
//  Dictionary2.java
//
//  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;
   }
```



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

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

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

Words2.java - Sample Execution

- The following is a sample execution of Words2.class

```
cuse93> java Words2
Number of pages: 1500
Number of definitions: 52500
Definitions per page: 35.0
```

The super Reference

- A child's constructor is responsible for calling the parent's constructor
- 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

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
- Java does not support multiple inheritance
- In most cases, the use of interfaces gives us aspects of multiple inheritance without the overhead

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`

```
//********************************************************************
//  Messages.java
//
//  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
   }
}
```

```
//*************************************************************
//  Thought.java
//
//  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();
   }
}
```

```
//*****************************************************************
//  Advice.java
//
//  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 - Sample Execution

- The following is a sample execution of Messages.class

```
cuse93> java Messages
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.
```

Overriding

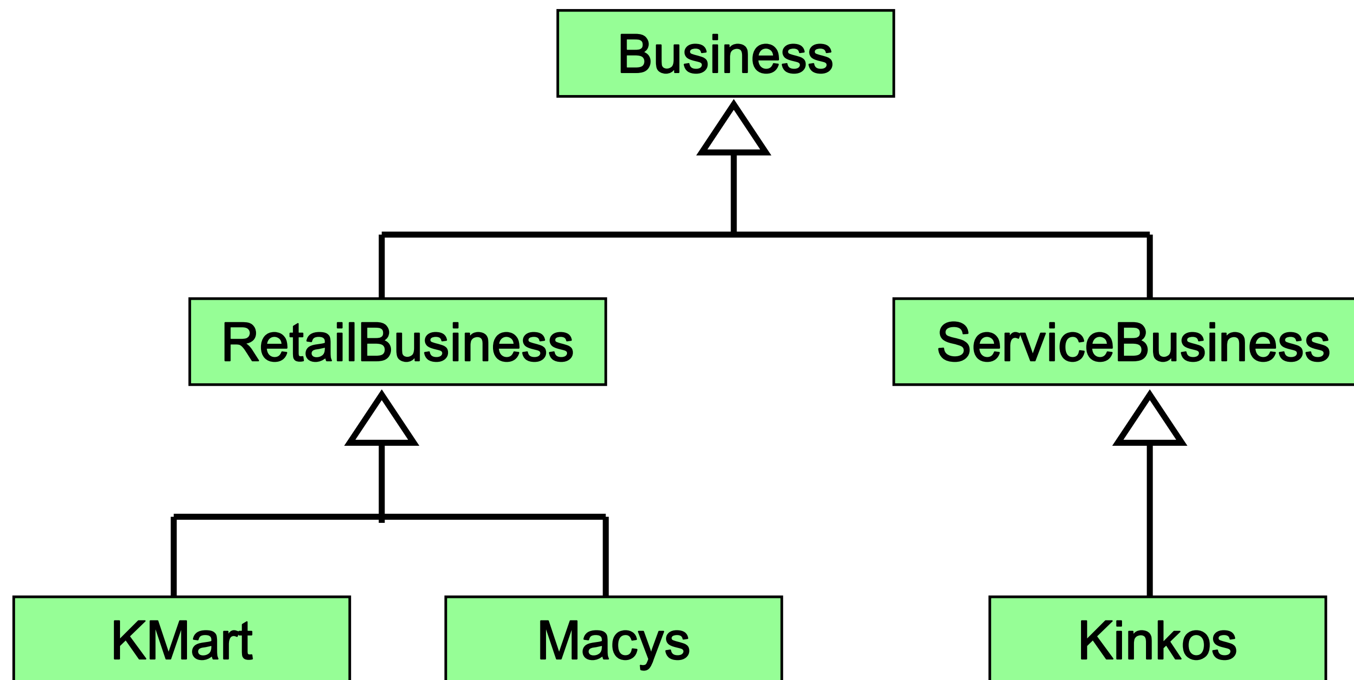
- A method in the parent class can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends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

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 some other information

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

Interface Hierarchies

- Inheritance can be applied to interfaces as well as classes
- That is, one interface can be derived from another interface
- The child interface inherits all abstract methods of the parent
- A class implementing the child interface must define all methods from both the ancestor and child interfaces
- Note that class hierarchies and interface hierarchies are distinct (they do not overlap)

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

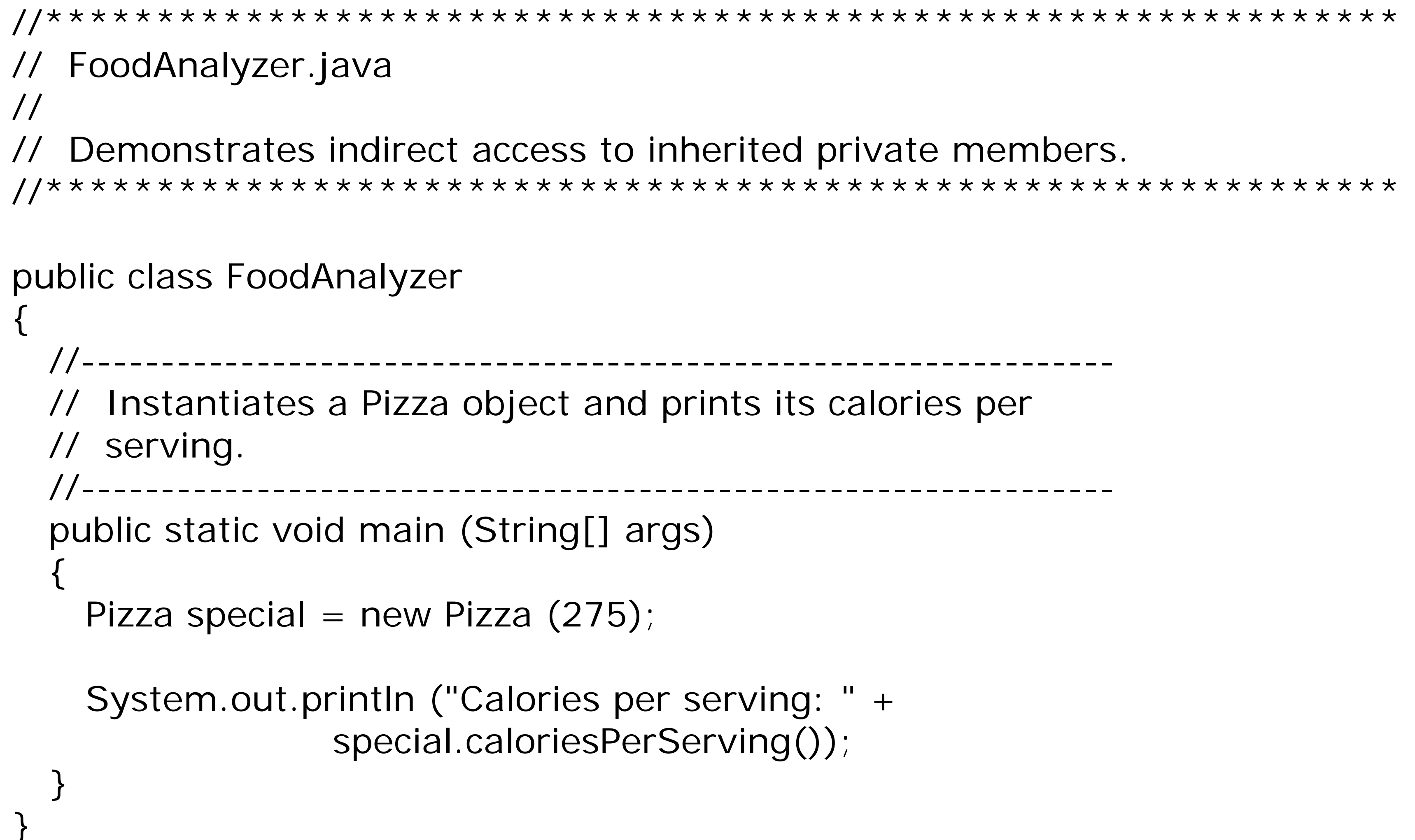
- Because the parent can refer to the private member, the child can reference it indirectly using its parent's methods

- The `super` reference can be used to refer to the parent class, even if no object of the parent exists

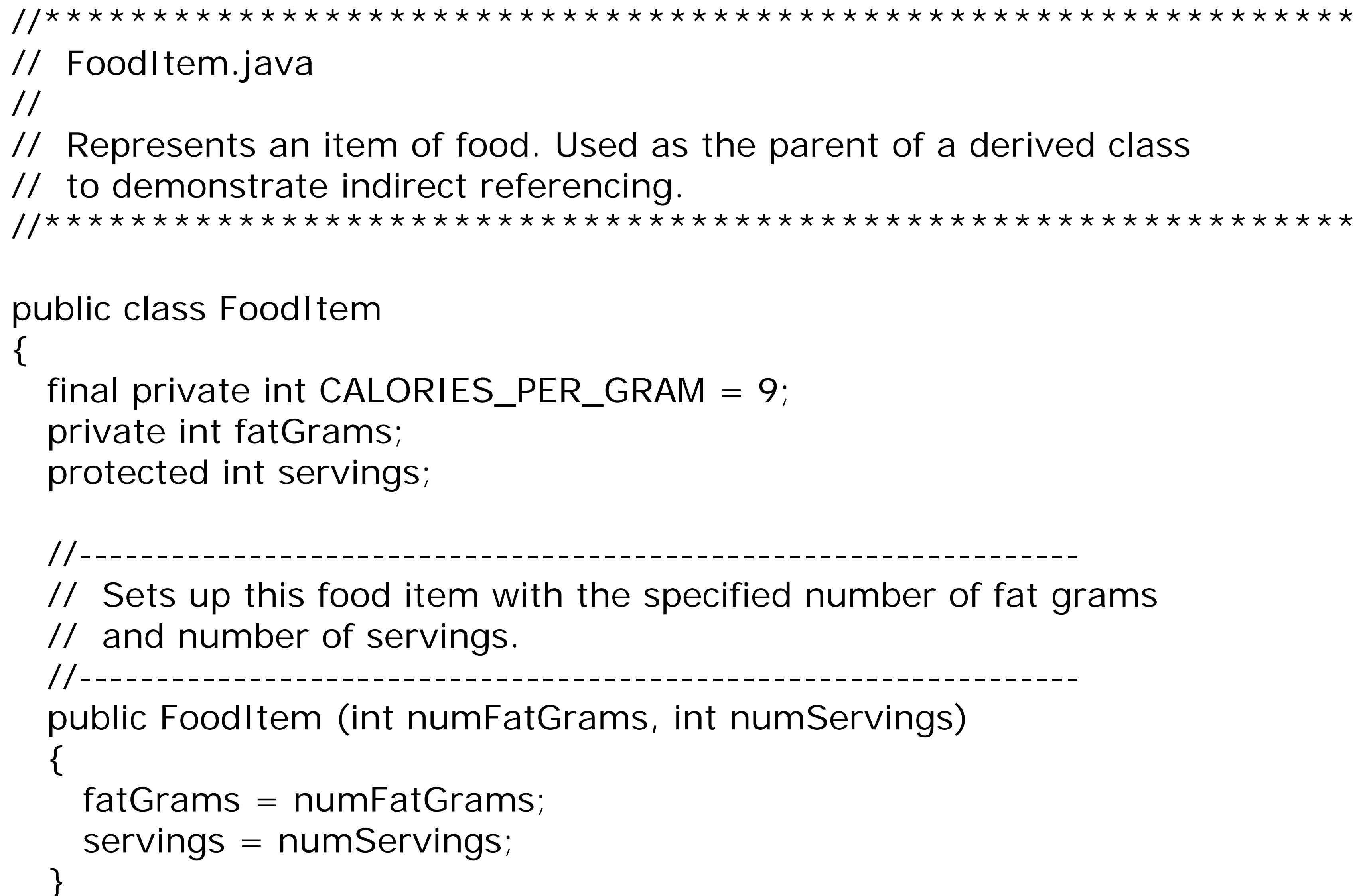
See `FoodAnalyzer.java`

See `FoodItem.java`

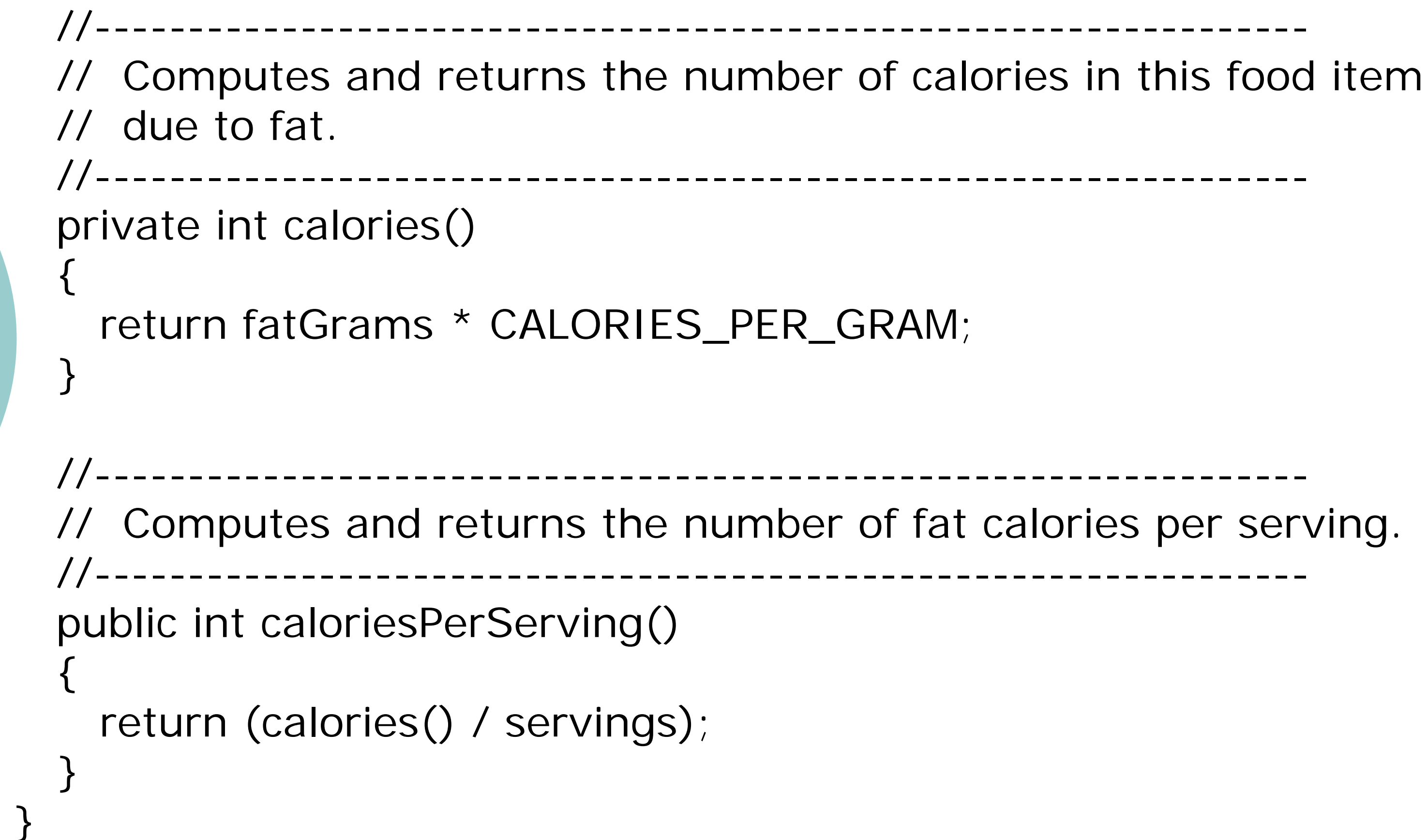
See `Pizza.java`



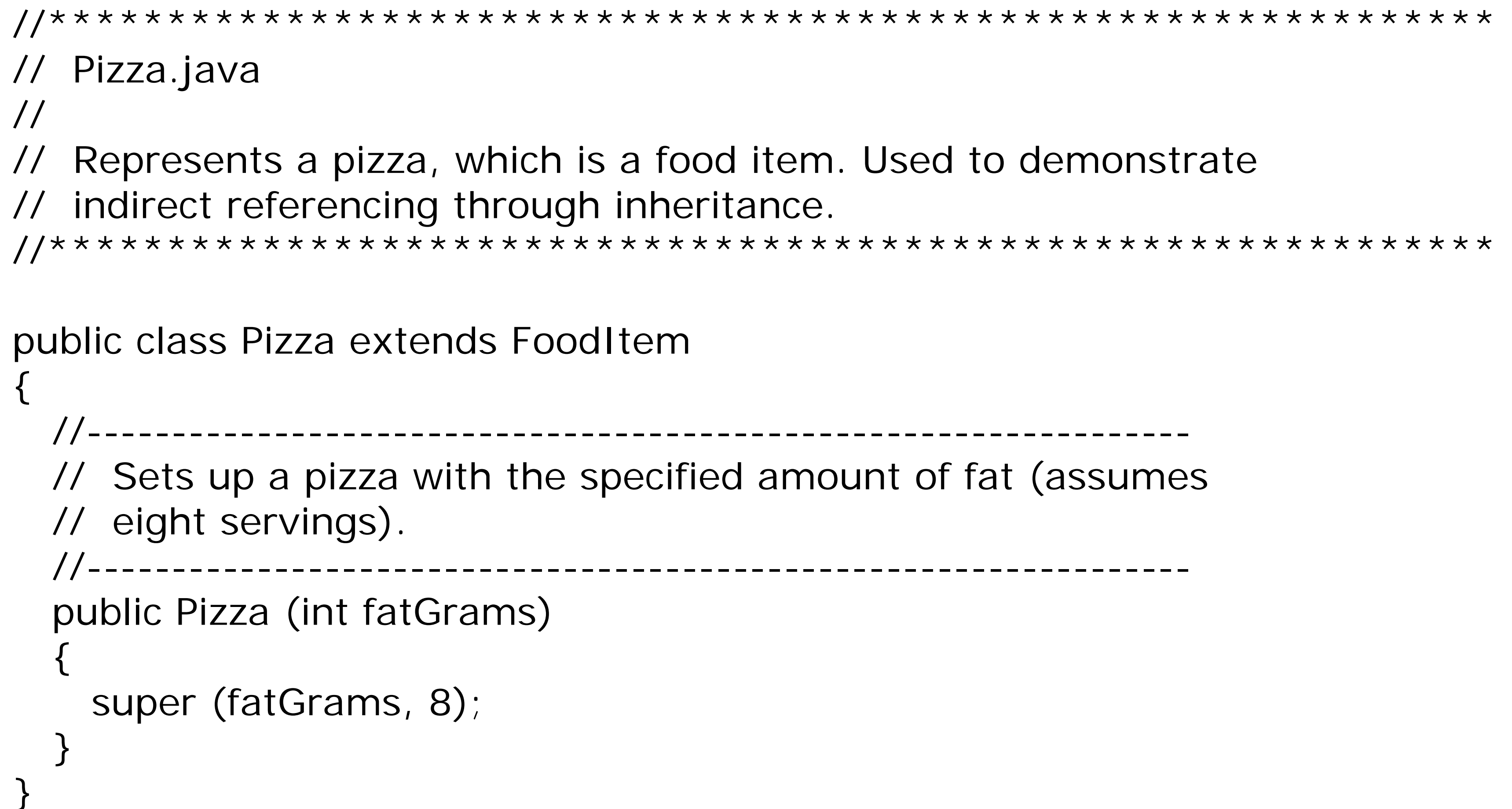
```java
//************************************************************
//  FoodAnalyzer.java
//
//  Demonstrates indirect access to inherited private members.
//************************************************************

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());
   }
}
```



```java
//*************************************************************
//  FoodItem.java
//
//  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;
   }
```



```
   //-----------------------------------------------------------------
   //  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
//
//  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);
   }
}
```

FoodAnalyzer.java - Sample Execution

- The following is a sample execution of FoodAnalyzer.class

```
cuse93> java FoodAnalyzer
Calories per serving: 309
```

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
- Even if there are no current uses for them, override general methods such as `toString` and `equals` with appropriate definitions
- Use abstract classes to represent general concepts that lower classes have in common
- Use visibility modifiers carefully to provide needed access without violating encapsulation

Restricting Inheritance

- The `final` modifier can be used to curtail inheritance
- If the `final` modifier is applied to a method, then that method cannot be overridden in any descendent classes
- If the `final` modifier is applied to an entire class, then that class cannot be used to derive any children at all
  - Thus, an abstract class cannot be declared as final
- These are key design decisions, establishing that a method or class should be used as is