CSE 1201
Object Oriented Programming

Inheritance

Acknowledgement

- For preparing the slides I took materials from the following sources
  - Course Slides of Dr. Tagrul Dayar, Bilkent University
  - Java book "*Java Software Solutions*" by Lewis & Loftus.

Inheritance

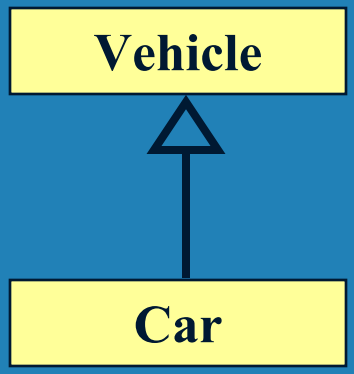
- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class*, or *superclass*, or *base class*
- The derived class is called the *child class* or *subclass*.
- As the name implies, the child inherits characteristics of the parent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

Inheritance

- To tailor a derived class, the programmer can add new variables or methods, or can modify the inherited ones
- *Software reuse* is at the heart 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

Inheritance

- Inheritance relationships often are shown graphically in a UML class diagram, with an arrow with an open arrowhead pointing to the parent class



Inheritance should create an *is-a relationship*, meaning the child *is a* more specific version of the parent

Deriving Subclasses

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

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

Book and Dictionary

```
public class Book
{
   protected int pages = 1500;

   //-----------------------------------------
   //  Prints a message about the pages of this
       book.
   //-----------------------------------------
   public void pageMessage ()
   {
      System.out.println ("Number of     pages:
       " + pages);
   }
}
```

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

   //-----------------------------------------------
   //  Prints a message using both local and
       inherited values.
   //-----------------------------------------------
   public void definitionMessage ()
   {
      System.out.println ("Number of
       definitions: " + definitions);

      System.out.println ("Definitions per
       page: " + definitions/pages);
   }
}
```

Dictionary webster=new Dictionary();

webster.pageMessage();
webster.definitionMessage();

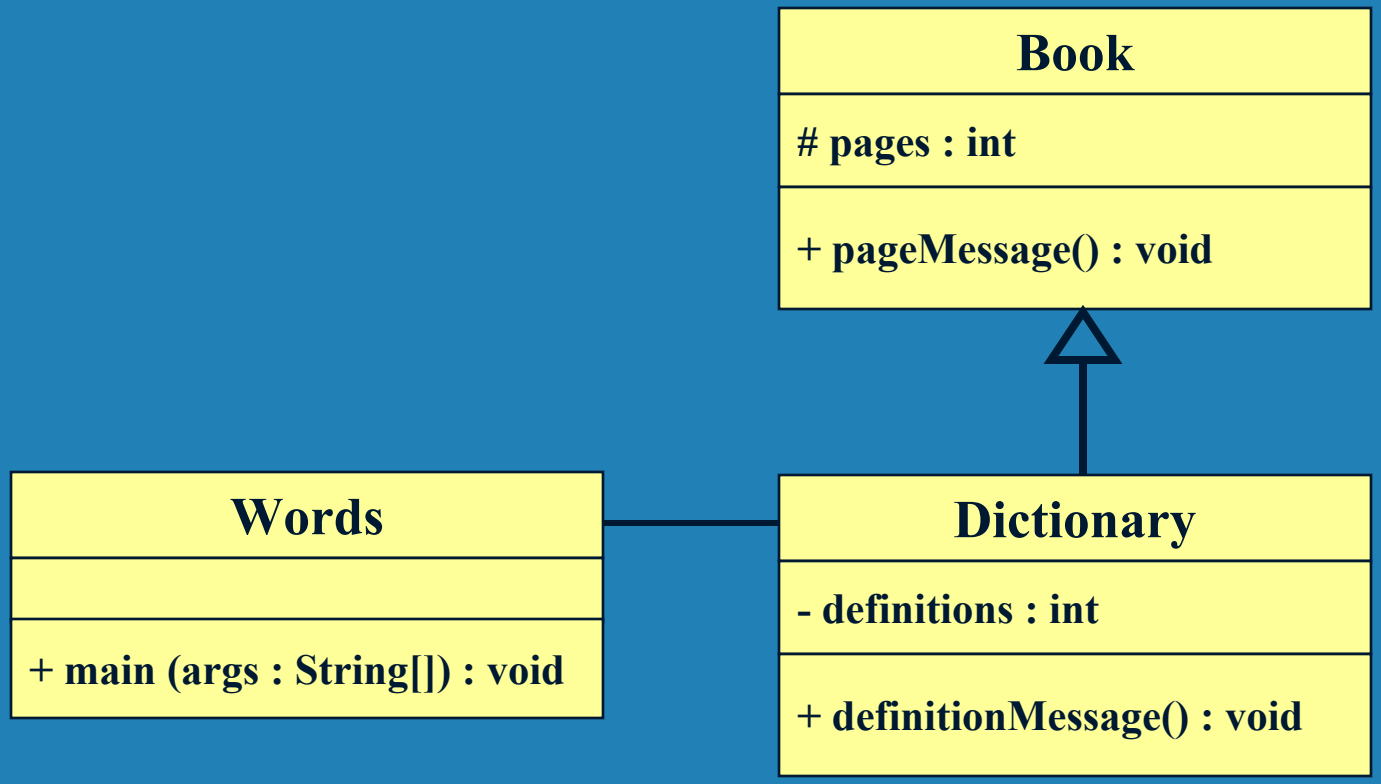
The protected Modifier

- 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
- Protected variables and methods can be shown with a # symbol preceding them in UML diagrams

UML 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

Book and Dictionary

```
public class Book2 {
   protected int pages;

  public Book2 (int numPages) {
      pages = numPages;
   }

  public void pageMessage ()
   {
      System.out.println ("Number of pages: "
       + pages);
   }
}
```

```
public class Dictionary2 extends Book2 {
   private int definitions;

  public Dictionary2 (int numPages, int
       numDefinitions) {
      super (numPages);
      definitions = numDefinitions;
   }

  public void definitionMessage () {
      System.out.println ("Number of
       definitions: " + definitions);
      System.out.println ("Definitions per
       page: " + definitions/pages);
   }
}
```

```
Dictionary2 webster = new Dictionary2
                (1500, 52500);
webster.pageMessage();
webster.definitionMessage();
```

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

Book and Dictionary

```
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();
   }
}
```

```
public class Advice extends Thought {
   //  Prints a message. This method
       overrides the parent's version.
    //  It also invokes the parent's version
       explicitly using super.

    public void message() {
       System.out.println ("Warning:
       Dates in calendar are closer " +
       "than they appear.");

       super.message();
    }
}
```

```
Thought parked = new Thought();
Advice dates = new Advice();

parked.message();
dates.message();  // overridden
```

Overriding

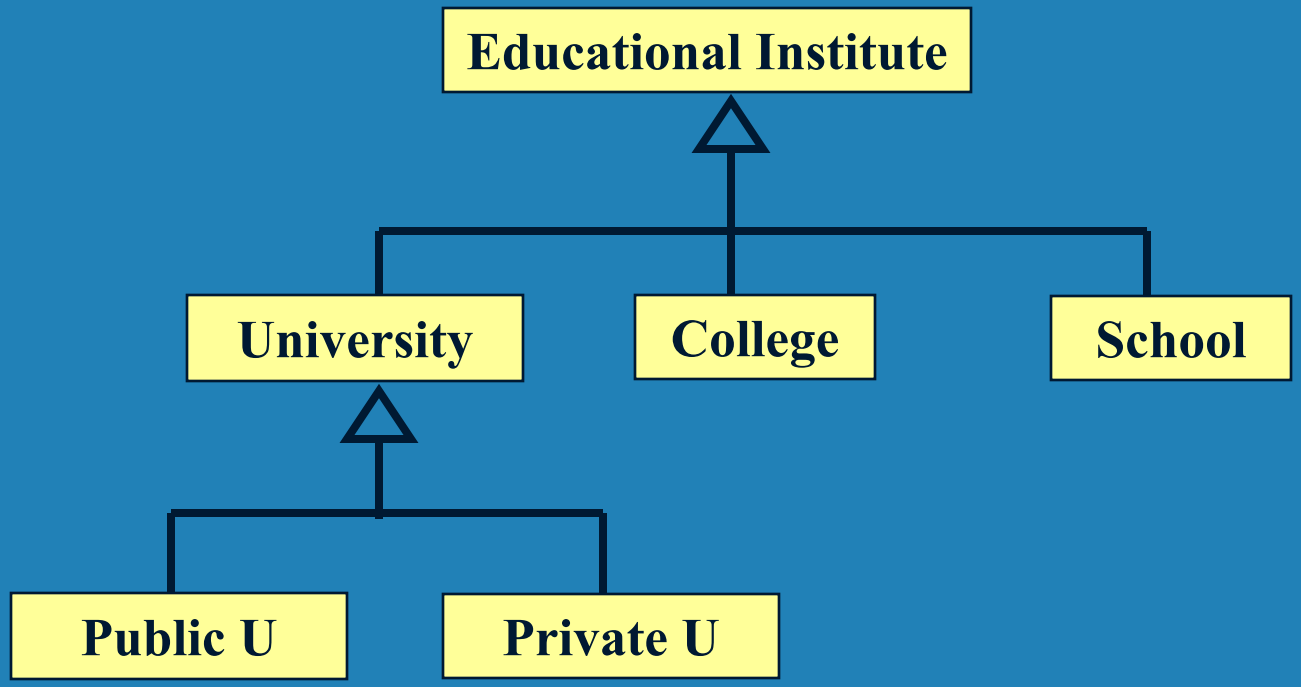
- A parent method can be invoked explicitly using the `super` reference
- If a method is declared with the `final` modifier, it cannot be overridden
- The concept of overriding can be applied to data and is called *shadowing variables*
- Shadowing variables should be avoided because it tends to cause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

Overloading vs. Overriding

- Don't confuse the concepts of overloading and 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 data
- 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*



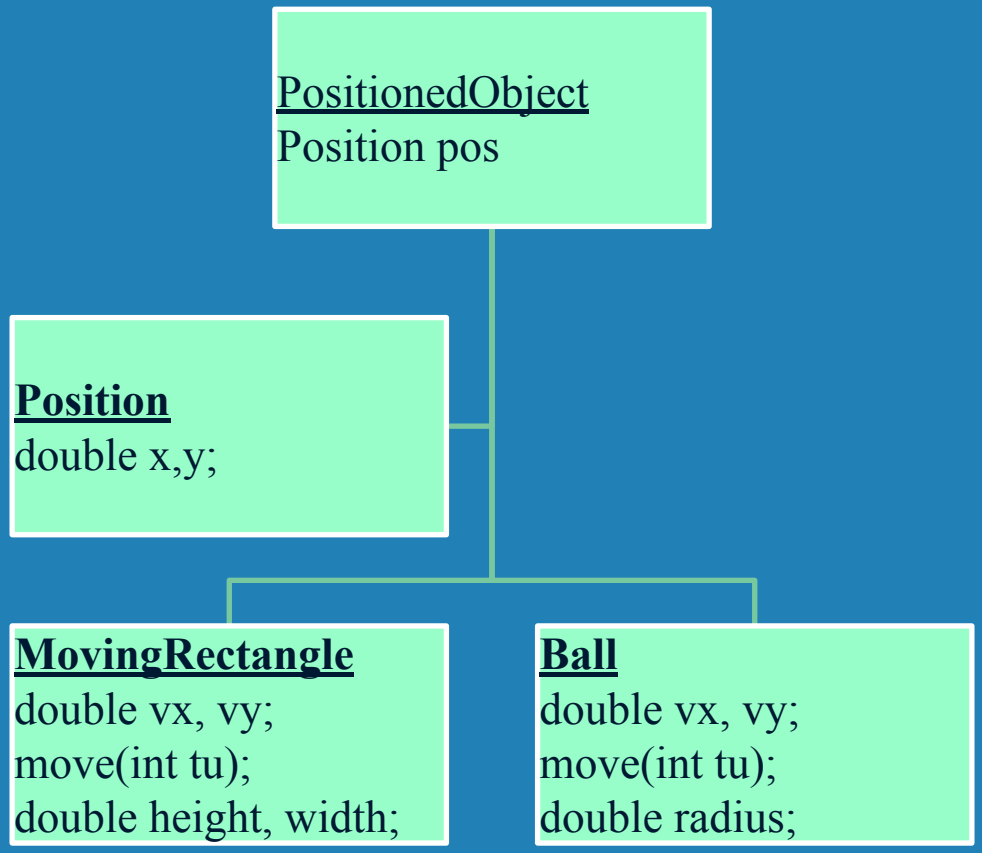
Class Hierarchies

- Two children of the same parent are called *siblings*
- Common features should be put as high in the hierarchy as is reasonable (otherwise code is duplicated)
- 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

Hierarchies

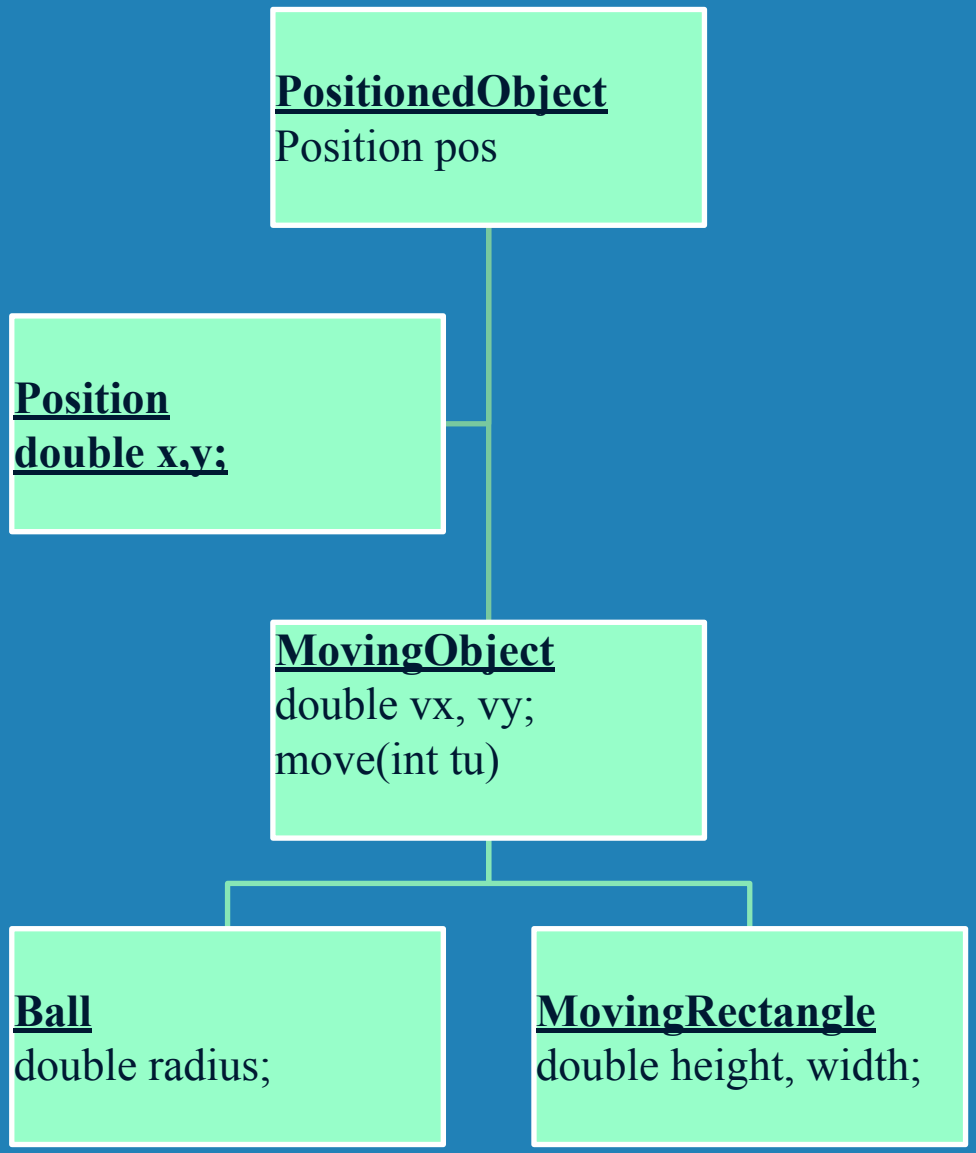
- Lets say we want to create a MovingRectangle class
- A MovingRectangle has a Position, velocity, height and width
- We already have Position and Ball classes
- How can we create a class hierarchy?
- Notice that both Ball and Moving Rectangle has-a Position
- Positioned Object?

First Try



- Although this is better than previous, vx, vy and the code for move is duplicated

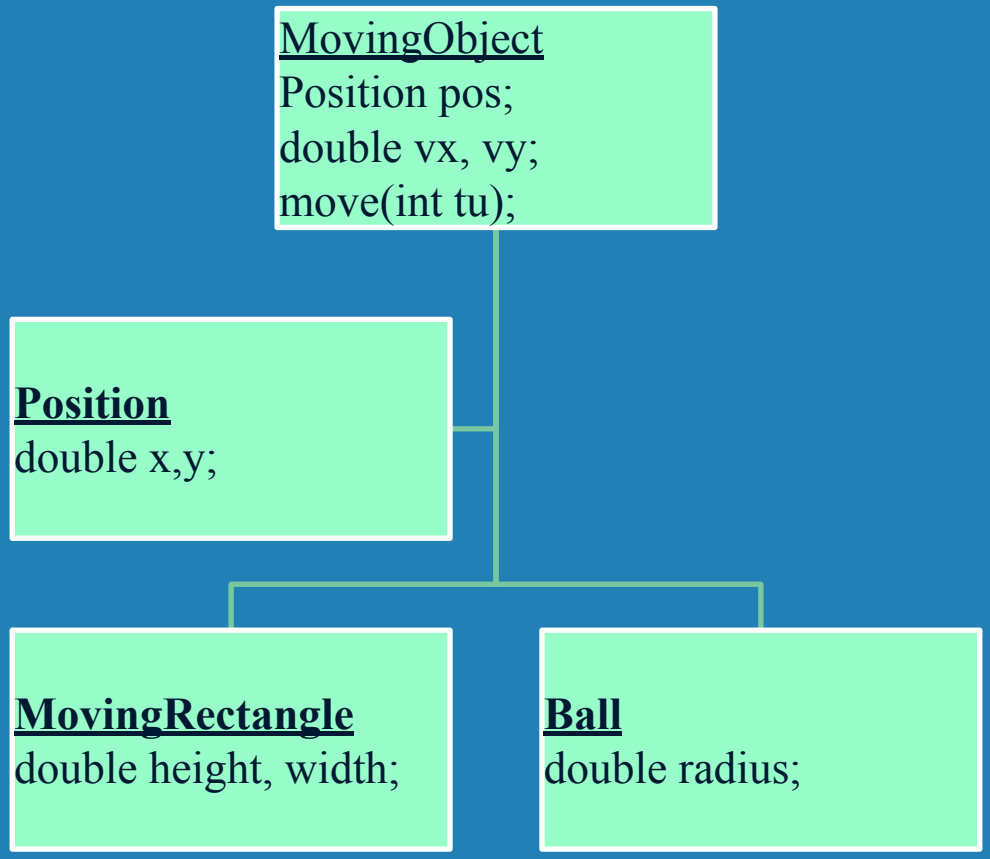
Second Try



- This is an example of overdoing inheritane.

  Too many layers

Third Try



- Here, no code duplication, no unnecessary layers.
- Given the current requirements, this seems like the best hierarchy

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 have defined `toString`, we have actually been overriding an existing definition
- The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class together along with some other information

The Object Class

- All objects are guaranteed to have a `toString` method via inheritance
- Thus the `println` method can call `toString` for any object that is passed to it

toString() Example

```
public class Student {
  protected String name;
  protected int numCourses;

  public Student (String studentName, int courses)
      {
    name = studentName;
    numCourses = courses;
  }

  public String toString() {
    String result = "Student name: " + name + "\n“
     +"Number of courses: " + numCourses;

    return result;
  }
}
```

```
public class GradStudent extends Student {
  private String source;
  private double rate;

  public GradStudent (String studentName, int
      courses,
                String support, double payRate) {
    super (studentName, courses);

    source = support;
    rate = payRate;
  }

  public String toString() {
    String result = super.toString();
    result += "\nSupport source: " + source + "\n";
    result += "Hourly pay rate: " + rate;

    return result;
  }
}
```

```
Student susan = new Student ("Susan", 5);
GradStudent frank = new GradStudent ("Frank", 3, "GTA", 12.75);

System.out.println (susan);
System.out.println (frank);
```

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
- The `String` class (as we've seen) defines the `equals` method to return true if two `String` objects contain the same characters
- Therefore the `String` class has overridden the `equals` method inherited from `Object` in favor of its own version

Equals() example

```
public boolean equals(Object obj) {
    Ball b = (Ball) obj; // gets an exception if obj is not of type Ball
    if (position.equals(b.getPosition()) && radius == b.radius &&
      vx == b.getVx() && vy == b.getVy() )
      return true;
    else
      return false;
 }
```

Indirect Use of Members

- A protected or public member can be referenced directly by name in the child class, as if it were declared in the child class
- But even if a method or variable is private, it can still be accessed indirectly through parent methods

FoodItem

```
public class FoodItem {
  final private int CALORIES_PER_GRAM = 9;
  private int fatGrams;
  protected int servings;
  public FoodItem (int numFatGrams, int
      numServings) {
    fatGrams = numFatGrams;
    servings = numServings;
  }
  private int calories() {
    return fatGrams * CALORIES_PER_GRAM;
  }
 public int caloriesPerServing() {
    return (calories() / servings);
  }
}
```

```
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);
  }
}
```