Chapter 9
Inheritance

Java Software Solutions
Foundations of Program Design
9th Edition

John Lewis
William Loftus
Great spirits have always encountered violent opposition from mediocre minds.
A. Einstein
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
  – the JavaFX class hierarchy
  – color and date pickers
  – dialog boxes
Outline

Creating Subclasses
Overriding Methods
Class Hierarchies
Visibility
Designing for Inheritance
Inheritance in JavaFX
Color and Date Pickers
Dialog Boxes
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*

- The child class **inherits** the **methods** and **data** defined by the parent *class* (*not the constructors!*)

- *But sometimes it cannot access them … see later*
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

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

• See `Words.java`
• See `Book.java`
• See `Dictionary.java`
public class Book {

    protected int pages = 1500;

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

    // Pages accessor.
    public int getPages () {
        return pages;
    }
}
public class Dictionary extends Book {
    private int definitions = 52500;
    
    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;
}
}
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());
    }
}
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());
    }
}
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 parent 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.
Class Diagram for Words

**Protected**

<table>
<thead>
<tr>
<th>Book</th>
</tr>
</thead>
<tbody>
<tr>
<td># pages : int</td>
</tr>
<tr>
<td>+ getPages() : int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>- definitions : int</td>
</tr>
<tr>
<td>+ computeRatio() : double</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ main (args : String[]) : void</td>
</tr>
</tbody>
</table>
This appendix summarizes the modifiers that give particular characteristics to Java classes, interfaces, methods, and variables. For discussion purposes, the set of all Java modifiers is divided into two groups: visibility modifiers and all others.

### Visibility Modifiers

Java visibility modifiers are: `default`, `public`, `protected`, and `private`. Each modifier affects the visibility of classes, interfaces, methods, and variables differently.

#### Table: Java Visibility Modifiers

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Classes and interfaces</th>
<th>Methods and variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>default (no modifier)</code></td>
<td>Visible in its package.</td>
<td>Inherited by any subclass in the same package as its class.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accessible by any class in the same package as its class.</td>
</tr>
<tr>
<td><code>public</code></td>
<td>Visible anywhere.</td>
<td>Inherited by all subclasses of its class.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accessible anywhere.</td>
</tr>
<tr>
<td><code>protected</code></td>
<td>N/A</td>
<td>Inherited by all subclasses of its class.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Accessible by any class in the same package as its class.</td>
</tr>
<tr>
<td><code>private</code></td>
<td>Visible to the enclosing class only</td>
<td>Not inherited by any subclass.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not accessible by any other class.</td>
</tr>
</tbody>
</table>

`Default visibility` means that no visibility modifier was explicitly used. Default visibility is sometimes called `package visibility`, but you cannot use the reserved word `package` as a modifier. Classes and interfaces can have default or public visibility; this visibility determines whether a class or interface can be referenced outside of its package. Only an inner class can have private visibility, in which case only the enclosing class may access it.

When applied to methods and variables, the visibility modifiers dictate two specific characteristics:

- **Inheritance**, which determines whether a method or variable can be referenced in a subclass as if it were declared locally.
- **Access**, or the degree of encapsulation, which determines the scope in which a method or variable can be directly referenced. All methods and variables are accessible in the class in which they are declared.
Inheritance

Objects

Superclass

obj1
obj2
Subclass

obj4
obj5

Sets contain objects

Members (data and methods)

Subclass

data3
method1

Superclass

data2
method2

data1

Sets contain data and methods
Inheritance and Visibility

• All members of the superclass are actually inherited by the subclass but the visibility of these members is controlled by the member visibility modifier.
The super Reference

- **Constructors are not inherited**, even though they have public visibility (*why?*)

- 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 (**actually it is always called**)
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**
public class Book2
{
  protected int pages;

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

  continue
public void setPages (int numPages)
{
    pages = numPages;
}

public int getPages ()
{
    return pages;
}
public class Dictionary2 extends Book2
{
    private int definitions;

    public Dictionary2 (int numPages, int numDefinitions)
    {
        super(numPages);

        definitions = numDefinitions;
    }

    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;
}
public class Words2
{

    // Instantiate 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());
    }
}
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());
    }
}
Constructors

• The constructor of a subclass **always calls a constructor of the superclass**

• If it is not explicitly called using the `super` reference, then the **default** constructor, i.e., that with no arguments, is called **automatically**

• If you have defined a constructor for the superclass remember that the default constructor **is not present anymore**

• Then if you have a subclass constructor that is not explicitly calling a constructor of the superclass you must implement the default constructor (i.e., that with no parameters).
Quiz

The instruction super( ); does which of the following?

A) calls the method super as defined in the current class
B) calls the method super as defined in the current class' parent class
C) calls the method super as defined in java.lang
D) calls the constructor as defined in the current class
E) calls the constructor as defined in the current class' parent class
Quiz

The instruction super(); does which of the following?

A) calls the method super as defined in the current class
B) calls the method super as defined in the current class' parent class
C) calls the method super as defined in java.lang
D) calls the constructor as defined in the current class
E) calls the constructor as defined in the current class' parent class
public class AClass {
    protected int x;
    protected int y;

    public AClass(int a, int b) {
        x = a;
        y = b;
    }

    public String toString() {
        return x + "   " + y;
    }
}
Quiz

Consider that you want to extend AClass to BClass. BClass will have a third int instance data, z. Which of the following would best define BClass' constructor?

A) public BClass(int a, int b, int c) {
    super(a, b, c);
}

B) public BClass(int a, int b, int c) {
    x = a;
    y = b;
    z = c;
}

C) public BClass(int a, int b, int c) {
    z = c;
}

D) public BClass(int a, int b, int c) {
    super(a, b);
    z = c;
}

E) public BClass(int a, int b, int c) {
    super( );
}
Consider that you want to extend AClass to BClass. BClass will have a third int instance data, z. Which of the following would best define BClass' constructor?

A) public BClass(int a, int b, int c) {
   super(a, b, c);
}

B) public BClass(int a, int b, int c) {
   x = a;
   y = b;
   z = c;
}

C) public BClass(int a, int b, int c) {
   z = c;
}

D) public BClass(int a, int b, int c) {
   super(a, b);
   z = c;
}

E) public BClass(int a, int b, int c) {
   super( );
}
Quiz

Inheritance through an extended (derived) class supports which of the following concepts?

A) interfaces
B) modularity
C) information hiding
D) code reuse
E) correctness
Quiz

Inheritance through an extended (derived) class supports which of the following concepts?

A) interfaces
B) modularity
C) information hiding
D) code reuse
E) correctness

By extending a class and inheriting from it, the new class does not have to reimplement any of those inherited methods or instance data, thus saving the programmer an effort.
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.
Quiz

Which of the following is an example of multiple inheritance? Draw the class diagram with the mentioned classes.

A) A computer can be a mainframe or a PC
B) A PC can be a desktop or a laptop
C) A laptop is both a PC and a portable device
D) A portable device is a lightweight device
E) Macintosh and IBM PC are both types of PCs
Quiz

Which of the following is an example of multiple inheritance?
A) A computer can be a mainframe or a PC
B) A PC can be a desktop or a laptop
C) A laptop is both a PC and a portable device
D) A portable device is a lightweight device
E) Macintosh and IBM PC are both types of PCs

Multiple inheritance means that a given class inherits from more than one parent class.
Quiz

• How does inheritance support software reuse?

• What does the protected modifier accomplish?
Quiz

• How does inheritance support software reuse?
  – Because a new class can be derived from an existing class, the characteristics of the parent class can be reused without the error-prone process of copying and modifying code.

• What does the protected modifier accomplish?
  – The protected modifier establishes a visibility level that takes inheritance into account. A variable or method declared with protected visibility can be referenced in the derived class (but also from other classes in the same package).
Outline

Creating Subclasses
Overriding Methods
Class Hierarchies
Visibility
Designing for Inheritance
Inheritance in JavaFX
Color and Date Pickers
Dialog Boxes
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** (name and parameters' types) 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` (extends `Thought`)
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 {
    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
    }
}
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();  // Advice extends Thought

        parked.message();  // "I feel ..."

        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.

```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(); // Advice extends Thought

        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 name and 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**
public class AClass {
    protected int x;
    protected int y;

    public AClass(int a, int b) {
        x = a;
        y = b;
    }

    public String toString() {
        return x + "   " + y;
    }
}

Copyright © 2012 Pearson Education, Inc.
Quiz

Consider that you want to extend AClass to BClass. BClass will have a third int instance data, z. Which of the following would best redefine the toString method for BClass?

A) public String toString(int z) {
    return x + "   " + y + "   " + z;
}

B) public String toString() {
    return super.toString();
}

C) public String toString() {
    return super.toString() + "   " + z;
}

D) public String toString() {
    return x + "   " + y + "   " + z;
}
Quiz

Which of the following would best redefine the toString method for BClass?

A) public String toString(int z) {
    return x + "   " + y + "   " + z;
}

B) public String toString( ) {
    return super.toString( );
}

C) public String toString( ) {
    return super.toString( ) + "   " + z;
}

D) public String toString( ) {
    return x + "   " + y + "   " + z;
}
Outline

Creating Subclasses
Overriding Methods
Class Hierarchies
Visibility
Designing for Inheritance
Inheritance in JavaFX
Color and Date Pickers
Dialog Boxes
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.
public class A1 {
    public int x;
    private int y;
    protected int z;
    ...
}
public class A2 extends A1 {
    protected int a;
    private int b;
    ...
}
public class A3 extends A2 {
    private int q;
    ...
}
Quiz

Which of the following is true with respect to A1, A2 and A3?

A) A1 is a subclass of A2 and A2 is a subclass of A3
B) A3 is a subclass of A2 and A2 is a subclass of A1
C) A1 and A2 are both subclasses of A3
D) A2 and A3 are both subclasses of A1
E) A1, A2 and A3 are all subclasses of the class A
Quiz

Which of the following is true with respect to A1, A2 and A3?

A) A1 is a subclass of A2 and A2 is a subclass of A3
B) A3 is a subclass of A2 and A2 is a subclass of A1
C) A1 and A2 are both subclasses of A3
D) A2 and A3 are both subclasses of descended from A1
E) A1, A2 and A3 are all subclasses of the class A
Quiz

Which of the following lists of instance data are accessible in class A2?

A) x, y, z, a, b
B) x, y, z, a
C) x, z, a, b
D) z, a, b
E) a, b

```java
public class A1 {
    public int x;
    private int y;
    protected int z;
    ...
}

public class A2 extends A1 {
    protected int a;
    private int b;
    ...
}

public class A3 extends A2 {
    private int q;
    ...
}
```
Quiz

Which of the following lists of instance data are accessible in class A2?

A) x, y, z, a, b
B) x, y, z, a
C) x, z, a, b
D) z, a, b
E) a, b

```java
public class A1 {
    public int x;
    private int y;
    protected int z;
    ...
}
public class A2 extends A1 {
    protected int a;
    private int b;
    ...
}
public class A3 extends A2 {
    private int q;
    ...
}
```
Quiz

Which of the following lists of instance data are accessible in A3?

A) x, y, z, a, b, q
B) a, b, q
C) a, q
D) x, z, a, q
E) x, a, q

```java
public class A1 {
    public int x;
    private int y;
    protected int z;
    ...
}

public class A2 extends A1 {
    protected int a;
    private int b;
    ...
}

public class A3 extends A2 {
    private int q;
    ...
}
```
Quiz

Which of the following lists of instance data are accessible in A3?

A) x, y, z, a, b, q
B) a, b, q
C) a, q
D) x, z, a, q
E) x, a, q
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
Quiz

Java does not support multiple inheritance, but some of the abilities of multiple inheritance are available by

A) importing classes
B) implementing interfaces
C) overriding parent class methods
D) creating aliases
E) using public rather than protected or private modifiers
Quiz

Java does not support multiple inheritance, but some of the abilities of multiple inheritance are available by
A) importing classes
B) implementing interfaces
C) overriding parent class methods
D) creating aliases
E) using public rather than protected or private modifiers

Since a class can implement any number of interfaces, this class is inheriting the methods and constants of the interfaces. Further, the class could extend another class and thus inherit directly and indirectly from multiple classes.
Quiz

Which of the following is not a method of the Object class?

A) clone
B) compareTo
C) equals
D) toString
E) all of the above are methods of the Object class
Quiz

Which of the following is not a method of the Object class?
A) clone
B) compareTo
C) equals
D) toString
E) all of the above are methods of the Object class

compareTo is not implemented by Object and must be explicitly implemented in any class that wants to implement the Comparable interface.
Abstract Classes

• An abstract class is a **placeholder** in a class hierarchy that represents a generic concept

• An abstract class **cannot be instantiated**

• We use the modifier **abstract** on the class header to declare a class as abstract:

```java
public abstract class Product {
    // class contents
}
```
Abstract Classes

• **An abstract class** *often* contains **abstract methods** with no definitions (like an interface)
  
  – `public abstract double pay();`

• Unlike an interface, the **abstract** modifier must be applied to each abstract method
  
  – because can also **contain non abstract methods** with full definitions

• A class declared as abstract does not have to contain abstract methods -- simply declaring it as abstract makes it so
Abstract Classes

• The **child** of an abstract class **must override the abstract methods** of the **parent**, or it too will be considered abstract

• An abstract method cannot be defined as **final** (Why?) or **static** (Why?)

• The use of abstract classes is an important element of software design – it allows us to establish common elements in a hierarchy that are too general to instantiate
Quiz

If you instantiate an Abstract class, the class or object you wind up with

A) is also an Abstract class
B) is a normal class
C) is an Interface
D) is a reference to an Object
E) can't exist you cannot instantiate an Abstract class
Quiz

If you instantiate an Abstract class, the class or object you wind up with

A) is also an Abstract class
B) is a normal class
C) is an Interface
D) is a reference to an Object
E) can't exist you cannot instantiate an Abstract class

Explanation: E) You only can instantiate concrete classes not Abstract ones. But you can extend Abstract classes as well as Interfaces.
Interface Hierarchies

• Inheritance can be applied to interfaces

• 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 interfaces

• Class hierarchies and interface hierarchies are distinct (they do not overlap)
Quick Check

What are some methods defined by the `Object` class?

What is an abstract class?
Quick Check

What are some methods defined by the `Object` class?

- `String toString()`
- `boolean equals(Object obj)`
- `Object clone()`

What is an abstract class?

An abstract class is a placeholder in the class hierarchy, defining a general concept and gathering elements common to all derived classes. An abstract class cannot be instantiated.
Quiz

Abstract methods are used when defining
A) interface classes
B) derived classes
C) classes that have no constructor
D) arrays
E) classes that have no methods
Quiz

Abstract methods are used when defining
A) interface classes
B) derived classes
C) classes that have no constructor
D) arrays
E) classes that have no methods

An interface is a class that has defined some of its components, but leaves other components (methods) for you to implement. So, these components (methods) are referred to as abstract and defined in the interface class as abstract.
Abstract Classes vs. Interfaces (I)

• Consider using **abstract classes** if any of these statements apply to your situation:
  – You want to **share code** among several **closely related** classes.
  – You expect that classes that extend your abstract class **have many common methods or fields**, or require access modifiers other than public (such as protected and private).
  – You want to declare **non-static or non-final fields**. This enables you to define methods that can access and modify the state of the object to which they belong.
Abstract Classes vs. Interfaces (II)

- Consider using interfaces if any of these statements apply to your situation:
  - You expect that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
  - You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
  - You want to take advantage of multiple inheritance of type.
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
//********************************************************************
//  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; // cannot be accessed in subclass
    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
// Computes and returns the number of calories in this food item due to fat.
private int calories() // cannot be accessed in subclass
{
    return fatGrams * CALORIES_PER_GRAM;
}

// Computes and returns the number of fat calories per serving.
public int caloriesPerServing()
{
    return (calories() / servings);
}
Since fatGrams is private in FoodItem you cannot access fatGrams in any of the Pizza's methods.
public class FoodAnalyzer
{
    public static void main (String[] args)
    {
        Pizza special = new Pizza (275);
        System.out.println ("Calories per serving: " + special.caloriesPerServing());
    }
}

This is a method of FoodItem and therefore can access the private method calories() that access the private data member fatGrams
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());
    }
}
Outline

Creating Subclasses
Overriding Methods
Class Hierarchies
Visibility
Designing for Inheritance
Inheritance in JavaFX
Color and Date Pickers
Dialog Boxes
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
Quiz

• Analyze the following requirements and identify possible classes, is-a relationships and aggregations

• The user must be allowed to select one category among: events, hotels, attractions, suggestions. Searching for hotels the user may specify the booking period, the required hotel facilities and preferred cost. Results could be ordered by distance to a target position (e.g., the current one), by price by the system predicted user utility. Attractions are searched by tags: the system provides the attractions that have been tagged with a user specified tag.
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`.

  - `final`: stops inheritance and reuse
  - `abstract`: means that it will be extended/inherited
public class Foo {
    private int x;
    public Foo(int newValue) {
        x = newValue;
    }
}

If q1 and q2 are objects of class Foo, then q1.equals(q2):
A) is a syntax error since equals is not defined in the Foo class
B) is true if q1 and q2 both store the same value of x
C) is true if q1 and q2 reference the same Foo object
D) is never true
E) throws a NullPointerException
public class Foo {
    private int x;
    public Foo(int newValue) {
        x = newValue;
    }
}

If q1 and q2 are objects of Foo, then q1.equals(q2)
A) is a syntax error since equals is not defined in the Foo class
B) is true if q1 and q2 both store the same value of x
C) is true if q1 and q2 reference the same Foo object
D) is never true
E) throws a NullPointerException
Summary (I)

• Fields not declared as private can be used directly, just like any other fields

• You can declare a field in the subclass with the same name as the one in the superclass, thus hiding it (not recommended)

• You can declare new fields in the subclass that are not in the superclass

• The inherited methods can be used directly as they are.
Summary (II)

• You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it

• You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it

• You can declare new methods in the subclass that are not in the superclass

• You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword `super`.
Example

```java
public class Car {

    private double speed;
    private static int numCars;

    Car() {
        speed = 0.0;
        numCars++;
    }

    public void accelerate() {
        speed *= 1.1;
    }

    public static int getNumCars() {
        return numCars;
    }
}
```

```java
public class SportCar extends Car{
    private boolean kers;
    private static int numSportCars;

    SportCar() {
        super();
        numSportCars++;
        kers = false;
    }

    //overriding
    public void accelerate() {
        kers = true;
        super.accelerate();
        kers = false;
    }

    //hiding
    public static int getNumCars() {
        return numSportCars;
    }
}
```
Quiz

• Which is the only Java class that does not have a parent class?

• Why it is a contradiction to define a final, abstract class?

• Are all members of a parent class inherited by the children? Discuss.
Quiz

• Which is the only Java class that does not have a parent class?
  – Object

• Why it is a contradiction to define a final, abstract class?
  – A final class cannot be extended, while an abstract class is defined because we want to define subclasses that implement the abstract method of the parent.

• Are all members of a parent class inherited by the children? Discuss.
  – Yes, but may not be directly accessible, if they were defined as private.
Quiz

Which of the following is true regarding Java classes?

A) All classes must have 1 parent but may have any number of children (derived or extended) classes
B) All classes must have 1 child (derived or extended) class but may have any number of parent classes
C) All classes must have 1 parent class and may have a single child (derived or extended) class
D) All classes can have any number (0 or more) of parent classes and any number of children (derived or extended) classes
E) All classes can have either 0 or 1 parent class and any number of children (derived or extended) classes
Quiz

Which of the following is true regarding Java classes?

A) All classes must have 1 parent but may have any number of children (derived or extended) classes
B) All classes must have 1 child (derived or extended) class but may have any number of parent classes
C) All classes must have 1 parent class and may have a single child (derived or extended) class
D) All classes can have any number (0 or more) of parent classes and any number of children (derived or extended) classes
E) All classes can have either 0 or 1 parent class and any number of children (derived or extended) classes
Outline

Creating Subclasses
Overriding Methods
Class Hierarchies
Visibility
Designing for Inheritance
Inheritance in JavaFX
Color and Date Pickers
Dialog Boxes
Inheritance in JavaFX

Node
  ├── Shape
  │   └── Text
  │       ├── Line
  │       │   └── Rectangle
  │       │       ├── Circle
  │       │           └── Ellipse
  │       │               └── Arc
  │       └── Polygon
  │           └── Polyline
  │               └── Path
  └── ImageView

  ├── Parent
  │   └── Region
  │       └── Group
  └── Control
         ├── Labeled
         │   └── ButtonBase
         │       └── CheckBox
         │           └── ToggleButton
         │               └── RadioButton
         └── Pane
                  ├── HBox
                  │       └── VBox
                  │           └── FlowPane
                  │               └── GridPane
                  │                   └── Group
                  └── Group

  └── Pane

Abstract
Inheritance in JavaFX

• All shape classes are derived from Shape, which manages stroke and fill

• Nodes derived from Parent can hold other nodes
  – So shapes cannot hold other nodes

• Any Region can be styled with CSS

• All layout panes are derived from Pane

• Controls have various intermediate classes to organize common characteristics
Inheritance in JavaFX

• Note the difference between the scene graph (containment) and the inheritance hierarchy

• Only a Parent can serve as the root node of a scene

• So a Pane can be the root node of a scene, and could contain a Circle and a Button

• A ChoiceBox could be the root node of a scene, but a Rectangle could not
Color and Date Pickers

- A *color picker* is a control that allows the user to select a color from a palette or other model.

- A *date picker* allows the user to select a calendar date.

- Both appear as a single field and use drop-down panes for the selection.

- See PickerDemo.java
```java
import java.time.LocalDate;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.ColorPicker;
import javafx.scene.control.DatePicker;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.scene.text.Font;
import javafx.scene.text.FontPosture;
import javafx.scene.text.FontWeight;
import javafx.scene.text.Text;
import javafx.stage.Stage;

//************************************************************************
// PickerDemo.java    Author: Lewis/Loftus
//
// Demonstrates the use of color picker and date picker controls.
//************************************************************************

public class PickerDemo extends Application
{
    private Text message;
    private DatePicker datePicker;
    private ColorPicker colorPicker;

    public static void main(String[] args)
    {
        launch(args);
    }

    public void start(Stage primaryStage)
    {
        // Configure scene
        Scene scene = new Scene(new HBox(), 300, 250);
        scene.getStylesheets().add(getClass().getResource("styles.css").toExternalForm());

        // Create label
        Text message = new Text("Welcome to Picker Demo!");
        message.setFont(new Font("Arial", 16));
        message.setFill(Color.RED);

        // Create date picker
        datePicker = new DatePicker();
        datePicker.setPrefSize(100, 20);

        // Create color picker
        colorPicker = new ColorPicker();
        colorPicker.setPrefSize(100, 20);

        // Add components to HBox
        HBox hBox = new HBox();
        hBox.getChildren().add(message);
        hBox.getChildren().add(datePicker);
        hBox.getChildren().add(colorPicker);

        // Set alignment
        hBox.setAlignment(Pos.CENTER);  

        // Add HBox to scene
        scene.setRoot(hBox);

        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```
public void start(Stage primaryStage) {
    datePicker = new DatePicker(LocalDate.now());
    datePicker.setOnAction(this::processDateChoice);

    colorPicker = new ColorPicker(Color.BLACK);
    colorPicker.setOnAction(this::processColorChoice);

    message = new Text("HAPPY " + LocalDate.now().getDayOfWeek());
    message.setFont(Font.font("Helvetica", FontWeight.BOLD,
                       FontPosture.REGULAR, 24));

    HBox pickers = new HBox(datePicker, colorPicker);
    pickers.setSpacing(15);
    pickers.setAlignment(Pos.CENTER);

    VBox root = new VBox();
    root.setStyle("-fx-background-color: white");
    root.setSpacing(20);
    root.setAlignment(Pos.CENTER);
    root.getChildren().addAll(pickers, message);
}
Scene scene = new Scene(root, 400, 150);

primaryStage.setTitle("Picker Demo");
primaryStage.setScene(scene);
primaryStage.show();

//--------------------------------------------------------------------
// Gets the value of the date from the date picker and updates the
// message with the corresponding day of the week.
//--------------------------------------------------------------------
public void processDateChoice(ActionEvent event)
{
    LocalDate date = datePicker.getValue();
    message.setText("HAPPY " + date.getDayOfWeek());
}

//--------------------------------------------------------------------
// Gets the color specified in the color picker and sets the
// color of the displayed message.
//--------------------------------------------------------------------
public void processColorChoice(ActionEvent event)
{
    message.setFill(colorPicker.getValue());
}
Scene scene = new Scene(root, 400, 150);

primaryStage.setTitle("Picker Demo");
primaryStage.setScene(scene);
primaryStage.show();

//--------------------------------------------------------------------
// Gets the value of the date from the date picker and updates the
// message with the corresponding day of the week.
//--------------------------------------------------------------------
public void processDateChoice(ActionEvent event)
{
    LocalDate date = datePicker.getValue();
    message.setText("HAPPY " + date.getDayOfWeek());
}

//--------------------------------------------------------------------
// Gets the color specified in the color picker and sets the
// color of the displayed message.
//--------------------------------------------------------------------
public void processColorChoice(ActionEvent event)
{
    message.setFill(colorPicker.getValue());
}
Dialog Boxes

• A *dialog box* is a window that pops up to provide brief user interaction with a specific purpose.

• The *Alert* class provides support for several basic dialog box types:
  
  – `AlertType.INFORMATION`
  – `AlertType.CONFIRMATION`
  – `AlertType.WARNING`
  – `AlertType.ERROR`
Dialog Boxes

- Two other dialog box classes are `TextInputDialog` and `ChoiceDialog`.
- They allow the user to enter input using a text field or a drop-down choice box, respectively.
- See `EventOdd.java`
import java.util.Optional;
import javafx.application.Application;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.ButtonType;
import javafx.scene.control.TextInputDialog;
import javafx.stage.Stage;

//************************************************************************
// EvenOdd.java       Author: Lewis/Loftus
//
// Demonstrates the use of information and confirmation alerts, as well
// as text input dialog boxes.
//************************************************************************

public class EvenOdd extends Application
{
    //--------------------------------------------------------------------
    // Prompts the user for an integer, informs the user if that value
    // is even or odd, then asks if the user would like to process
    // another value. All interaction is performed using dialog boxes.
    //--------------------------------------------------------------------
    public void start(Stage primaryStage) throws Exception
    {
        boolean doAnother = true;
        continue
```java
while (doAnother)
{
    TextInputDialog inputDialog = new TextInputDialog();
    inputDialog.setHeaderText(null);
    inputDialog.setTitle(null);
    inputDialog.setContentText("Enter an integer: ");
    Optional<String> numString = inputDialog.showAndWait();

    if (numString.isPresent())
    {
        int num = Integer.parseInt(numString.get());

        String result = "That number is " +
                ((num % 2 == 0) ? "even." : "odd.");

        Alert answerDialog = new Alert(AlertType.INFORMATION);
        answerDialog.setHeaderText(null);
        answerDialog.setContentText(result);
        answerDialog.showAndWait();

        Alert confirmDialog = new Alert(AlertType.CONFIRMATION);
        confirmDialog.setHeaderText(null);
        confirmDialog.setContentText("Do another?");
        Optional<ButtonType> another = confirmDialog.showAndWait();
    }
}
```

continue

    if (another.get() != ButtonType.OK)
        doAnother = false;
    }
    else
        doAnother = false;
    }
}
continue

if (another.get() != ButtonType.OK)
    doAnother = false;
else
dothan = false;
File Choosers

• A *file chooser* is a specialized dialog box that allows the user to select a file from the hard drive

• The following example opens a text file using a file chooser and displays its contents in a *text area*

• *See* DisplayFile.java
import java.io.File;
import java.io.IOException;
import java.util.Scanner;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TextArea;
import javafx.scene.text.Font;
import javafx.stageFileChooser;
import javafx.stage.Stage;

//************************************************************************
// DisplayFile.java Author: Lewis/Loftus
//
// Demonstrates the use of a file chooser dialog box and a text area.
//************************************************************************

public class DisplayFile extends Application
{
    //-------------------------------------------------------------------------------------
    // Presents a file chooser dialog, reads the selected file and
    // loads it into a text area.
    //-------------------------------------------------------------------------------------
    public void start(Stage primaryStage) throws IOException
    {
        FileChooser chooser = new FileChooser();
        File selectedFile = chooser.showOpenDialog(primaryStage);
        continue
continue

    TextArea content = new TextArea();
    content.setFont(new Font("Courier", 12));
    content.setEditable(false);

    if (selectedFile == null)
        content.setText("No file chosen.");
    else
    {
        Scanner scan = new Scanner(selectedFile);

        String info = ""
        while (scan.hasNext())
            info += scan.nextLine() + "\n"

        content.setText(info);
    }

    Scene scene = new Scene(content, 500, 500);

    primaryStage.setTitle("Display File");
    primaryStage.setScene(scene);
    primaryStage.show();
}
continue

```java
Scene scene = new Scene(content, 500, 500);

primaryStage.setTitle("Display File");
primaryStage.setScene(scene);
primaryStage.show();
```
TextArea content = new TextArea();
content.setFont(new Font("Courier", 12));
content.setEditable(false);
if (selectedFile == null)
    content.setText("No file chosen.");
else {
    Scanner scan = new Scanner(selectedFile);
    String info = ""
    while (scan.hasNext())
        info += scan.nextLine() + "\n";
    content.setText(info);
}
Scene scene = new Scene(content, 500, 500);
primaryStage.setTitle("Display File");
primaryStage.setScene(scene);
primaryStage.show();
Summary

• Chapter 9 focused on:
  – deriving new classes from existing classes
  – the protected modifier
  – creating class hierarchies
  – abstract classes
  – indirect visibility of inherited members
  – designing for inheritance
  – the JavaFX class hierarchy
  – color and date pickers
  – dialog boxes