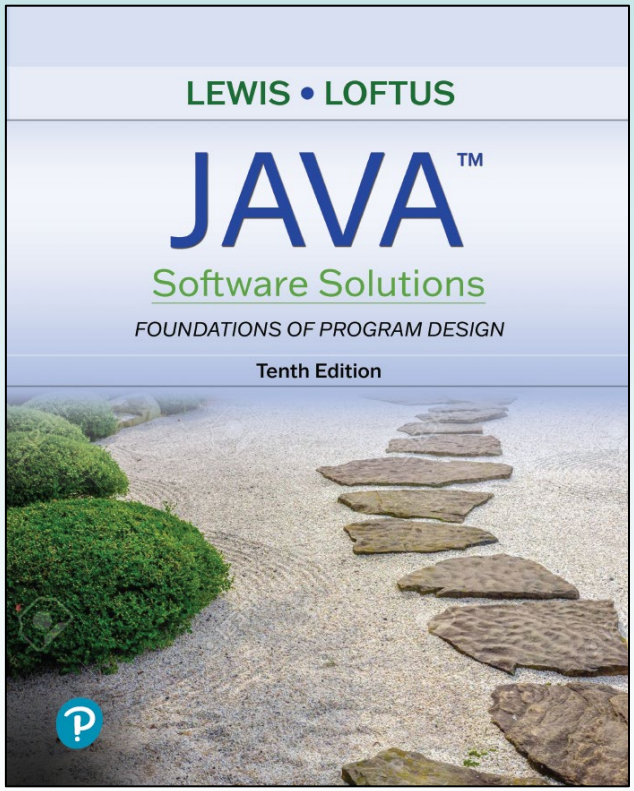
Chapter 9
Inheritance



Java Software Solutions

Foundations of Program Design

10th Edition

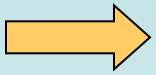
(edits: 05/20/2024)

John Lewis
William Loftus

Inheritance

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

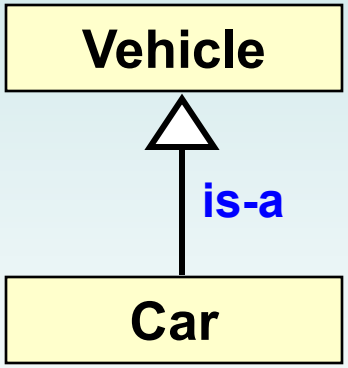
☺ **Dialog Boxes**

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,* or *derived class*
- As the name implies, the child inherits characteristics of the parent
  - The child class inherits the methods and variables defined by the parent class

Inheritance

- Inheritance in a UML class diagram uses a solid arrow with an unfilled triangular arrowhead pointing to the parent class



- Inheritance creates an *is-a* relationship
- The child *is a* more specific version of the parent

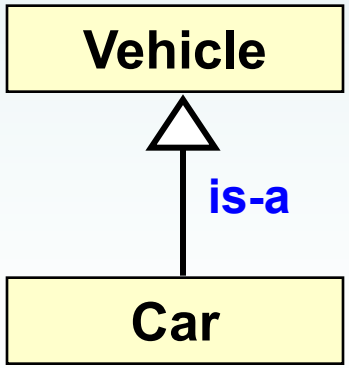
Inheritance

- 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 classes to create new ones, we build upon the design, implementation, and testing of the parent class
- Inheritance is between classes, not between objects.

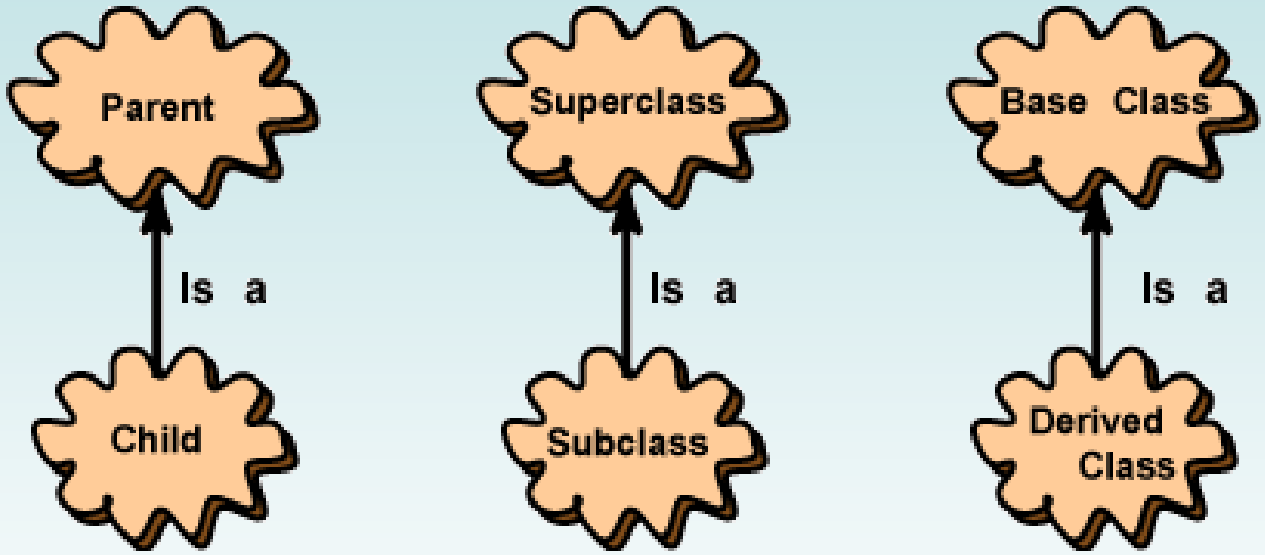
Deriving Subclasses

Use `extends` to establish inheritance

```
public class Car extends Vehicle
{
    // new code for Car goes
    // here.
}
```

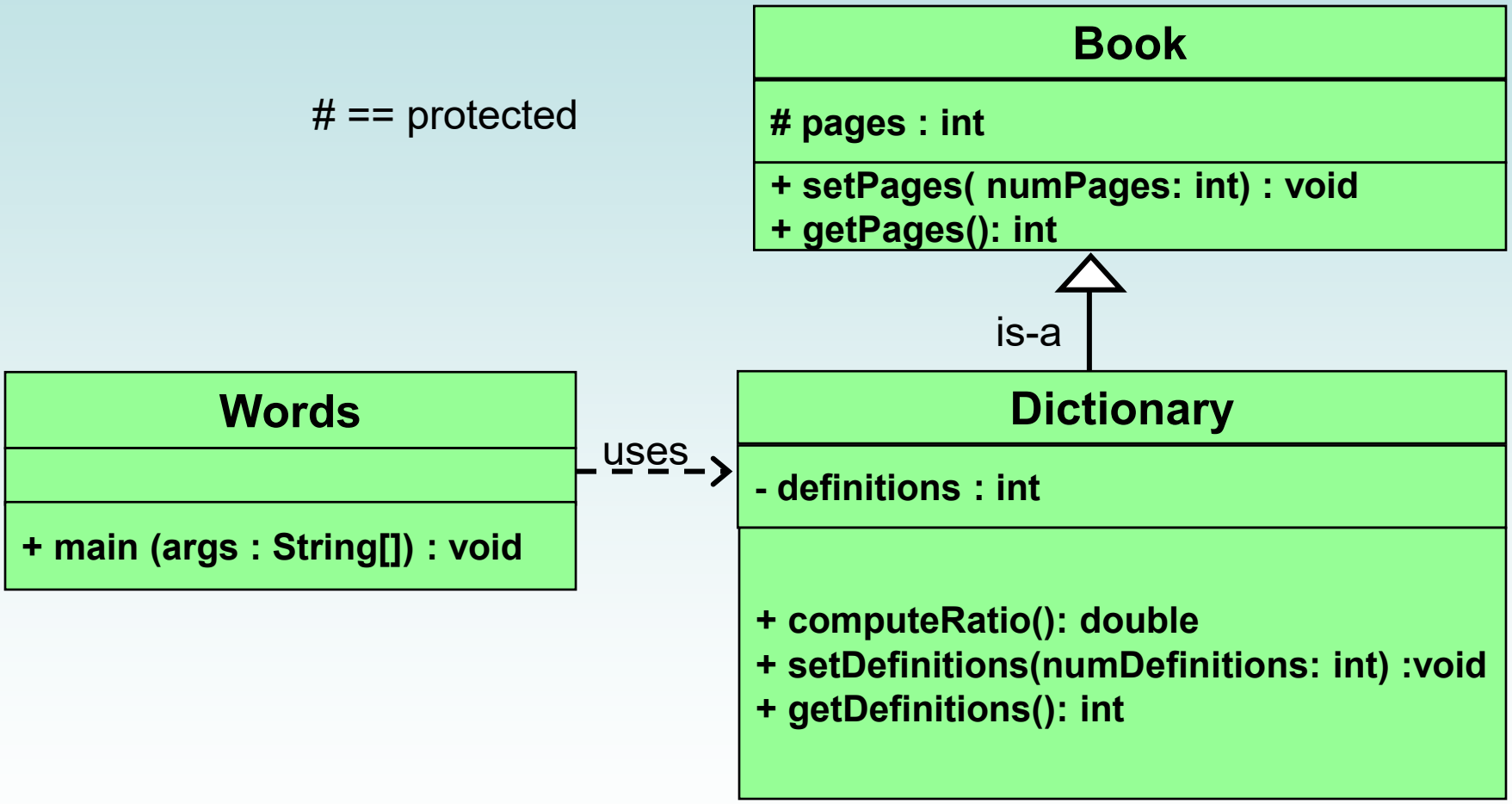


Parent / Child classes



Pairs of names used for inheritance relationships.

Class Diagram for Words



```
//********************************************
//  Words.java    Author: Lewis/Loftus
//
//  Demonstrates the use of an inherited
//  method.
//********************************************

public class Words
{
   //-----------------------------------------------------------------
   //  Instantiates a derived class and invokes its inherited and
   //  local methods.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Dictionary webster = new Dictionary(); // uses default constructor

      System.out.println("Number of pages: " +
                         webster.getPages()); // uses method inherited from Book

      System.out.println("Number of definitions: " +
                         webster.getDefinitions()); // uses method in Dictionary

      System.out.println("Definitions per page: " +
                         webster.computeRatio()); // uses method in Dictionary
   }
}
```

Output

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

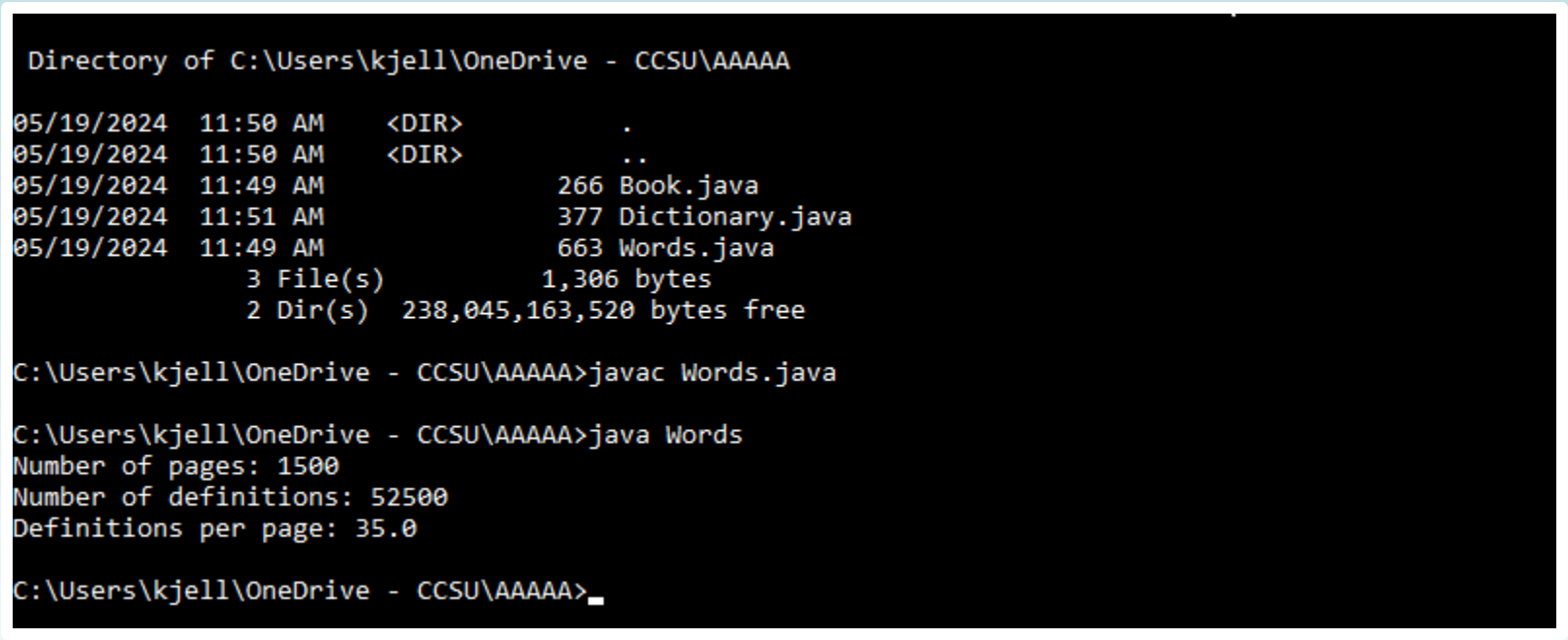
Challenges: add a book; Add another Dictionary

```java
//********************************************************************
//  Book.java       Author: Lewis/Loftus
//
//  Represents a book. Used as the parent of a derived class to
//  demonstrate inheritance.
//********************************************************************

public class Book
{
   protected int pages = 1500;  // protected: children can see this

   //-----------------------------------------------------------------
   //  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;

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

```
//*****************************************
//  Words.java     copy and save to a file
//*****************************************

public class Words
{
   public static void main(String[] args)
   {
      Dictionary webster = new Dictionary(); // uses default constructor

      System.out.println("Number of pages: " +
                  webster.getPages()); // uses method in Book

      System.out.println("Number of definitions: " +
                  webster.getDefinitions()); // uses method in Dictionary

      System.out.println("Definitions per page: " +
                  webster.computeRatio()); // uses method in Dictionary
   }
}
```

```
/* copy and save to Book.java */
public class Book
{
   protected int pages = 1500;  // protected: children can see this

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

   public int getPages()
   {
      return pages;
   }
}


/* copy and save to Dictionary.java */
public class Dictionary extends Book
{
   private int definitions = 52500;
   public double computeRatio()
   {
      return (double) definitions/pages;
   }
   public void setDefinitions(int numDefinitions)
   {
      definitions = numDefinitions;
   }
   public int getDefinitions()
   {
      return definitions;
   }
}
```



Javac compiles all files needed by Words.java

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 used in a child class
- Variables and methods declared with public visibility can be used in a child class (and other unrelated classes)
  - but this violates encapsulation
- There is a third visibility modifier: **`protected`**

The protected Modifier

- Variables and methods declared with protected visibility can be used in a child class (but not other classes)
- This provides more encapsulation than public visibility, but not as much as private visibility
- A protected variable is also visible to any class in the same package as the parent class
- Protected variables and methods have a # symbol preceding them in UML diagrams

The super Reference

- Constructors are not inherited,
  even though they have public visibility
- It would make no sense for Dictionary to have an inherited constructor named Book
- Yet we often want to use the parent's constructor to set up the parent's part of the object
- Use the **`super`** reference to refer to the parent class
  - often used to invoke the parent's constructor
- Do this in the first line of a child's constructor

The super Reference

- The first line of a child's constructor should use the `super` reference to call the parent's constructor
- The `super` reference can also be used to reference other variables and methods defined in the parent's class
- See `Words2.java`
- See `Book2.java`
- See `Dictionary2.java`

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

public class Words2
{
   //-----------------------------------------------------------------
   //  Instantiates a derived class and invokes its inherited and
   //  local methods.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      // uses constructor defined in the class Dictionary2
      Dictionary2 webster = new Dictionary2(1500, 52500);

      System.out.println("Number of pages: " + webster.getPages());

      System.out.println("Number of definitions: " +
                         webster.getDefinitions());

      System.out.println("Definitions per page: " +
                         webster.computeRatio());
   }
}
```

Output

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

```java
//********************************************************************
//  Book2.java       Author: Lewis/Loftus
//********************************************************************

public class Book2
{
   protected int pages;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the book with the specified number of
   //  pages. (Original Book did not have a constructor.)
   //-----------------------------------------------------------------
   public Book2(int numPages)
   {
      pages = numPages;
   }

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

   public int getPages()
   {
      return pages;
   }
}
```

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

public class Dictionary2 extends Book2
{
   private int definitions;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the dictionary with the specified number
   //  of pages and definitions. Original Dictionary did not have a
   //  constructor.
   //-----------------------------------------------------------------
   public Dictionary2(int numPages, int numDefinitions)
   {
      super(numPages);  // The whole point of this example

      definitions = numDefinitions;
   }
```

continue

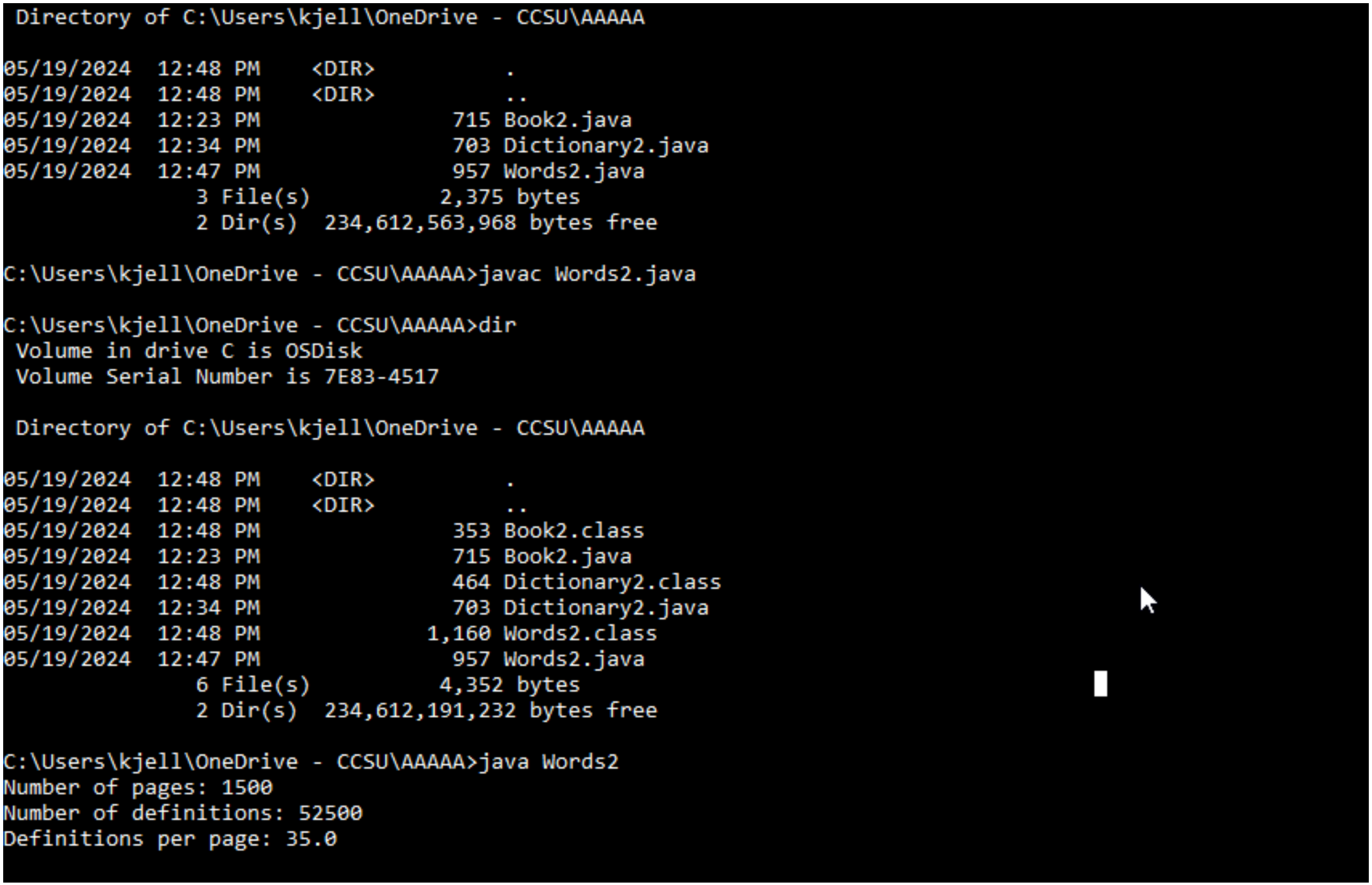
continue

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

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

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

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

public class Words2
{
   //-----------------------------------------------------------------
   //  Instantiates a derived class and invokes its inherited and
   //  local methods.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      // uses constructor defined in the class Dictionary2
      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       Author: Lewis/Loftus
//********************************************************************

public class Book2
{
   protected int pages;

   //-----------------------------------------------------------------
   //  Constructor: Sets up the book with the specified number of
   //  pages. (Original Book did not have a constructor.)
   //-----------------------------------------------------------------
   public Book2(int numPages)
   {
      pages = numPages;
   }

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

   public int getPages()
   {
      return pages;
   }
}
```

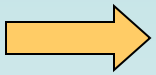
```
//********************************************************************
//  Dictionary2.java       Author: Lewis/Loftus
//********************************************************************

public class Dictionary2 extends Book2
{
   private int definitions;

   public Dictionary2(int numPages, int numDefinitions)
   {
      super(numPages);  // The whole point of this example

      definitions = numDefinitions;
   }

   public double computeRatio()
   {
      return (double) definitions/pages;
   }

   public void setDefinitions(int numDefinitions)
   {
      definitions = numDefinitions;
   }

   public int getDefinitions()
   {
      return definitions;
   }
}
```



Multiple Inheritance

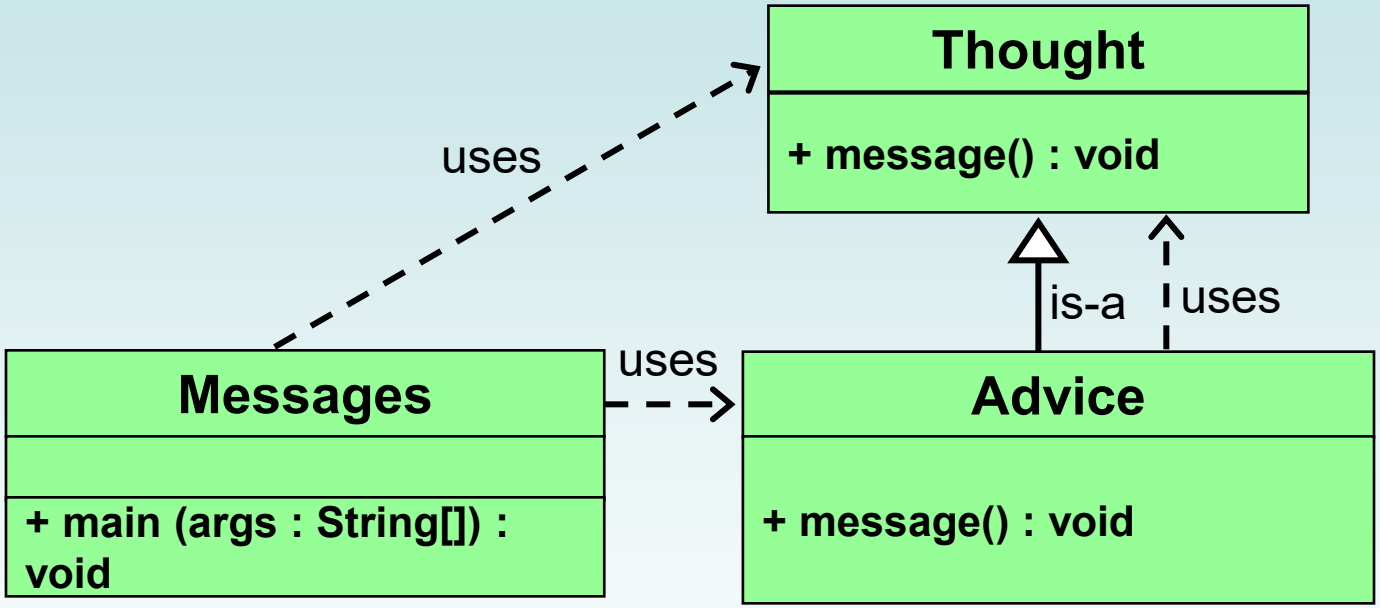
- Java has only *single inheritance*
  - a derived class can have only one parent class
- Java does not support *Multiple inheritance*
  - a derived class can have several parent classes
  - a derived class inherits the members of all parents
  - collisions, such as the same variable name in two parents, cause problems
- Multiple inheritance is not needed, so Java does not support it

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 as the parent's method, but can have a different body
- The type of the object that is executing the method determines which version of the method is invoked
- See `Messages.java`
- See `Thought.java`
- See `Advice.java`



The message() in Advice overrides the message() in Thought.

Also, the message() in Advice invokes the message() in Thought

```
//********************************************************************
//  Messages.java       Author: Lewis/Loftus
//
//  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(); // parent class
      Advice  dates  = new Advice();  // child class

      parked.message(); // method of parent is run
      System.out.println("-----------------");

      dates.message();  // method of child is run
   }
}
```

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.
```

```
//-----------------------------------------------------------------
//  Creates two objects and invokes the message method in each.
//-----------------------------------------------------------------
public static void main(String[] args)
{
   Thought parked = new Thought(); // parent class
   Advice  dates  = new Advice();  // child class

   parked.message(); // method of parent is run
   System.out.println("-----------------");

   dates.message();  // method of child is run
}
}
```

```
//********************************************************************
//  Thought.java       Author: Lewis/Loftus
//
//  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();
   }
}
```

```java
//********************************************************************
//  Advice.java       Author: Lewis/Loftus
//
//  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 invoke the parent's version
   }
}
```

Overriding and Shadowing

- 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
- Overriding can be applied to data and is called *shadowing variables*
  - a variable in a child class with the same name as a variable in the parent class hides the parent's variable in a shadow
- Avoid shadowing variables because it causes 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

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.

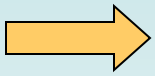
A child class may define a variable with the same name as a variable in the parent.

Quick Check

True or False?

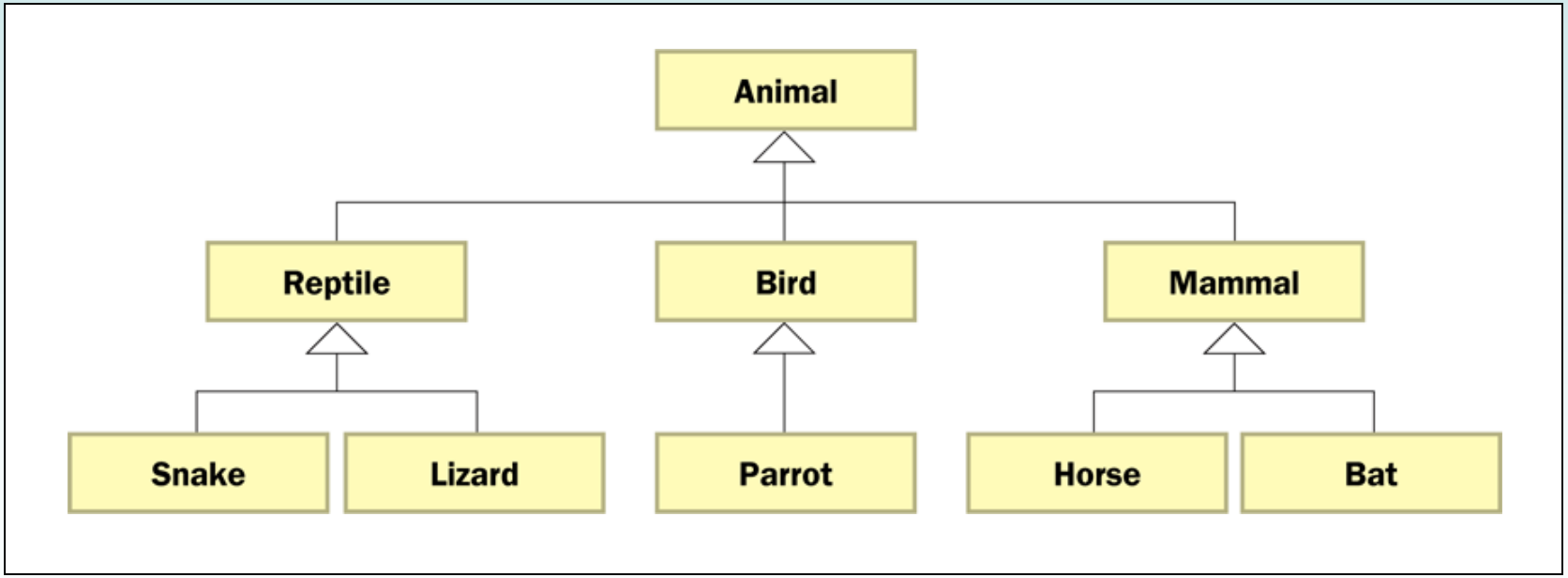
A child class may define a method with the same name as a method in the parent.	True
A child class can override the constructor of the parent class.	False
A child class cannot override a `final` method of the parent class.	True
It is considered poor design when a child class overrides a method from the parent.	False
A child class may define a variable with the same name as a variable in the parent.	True, but unwise

Outline

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

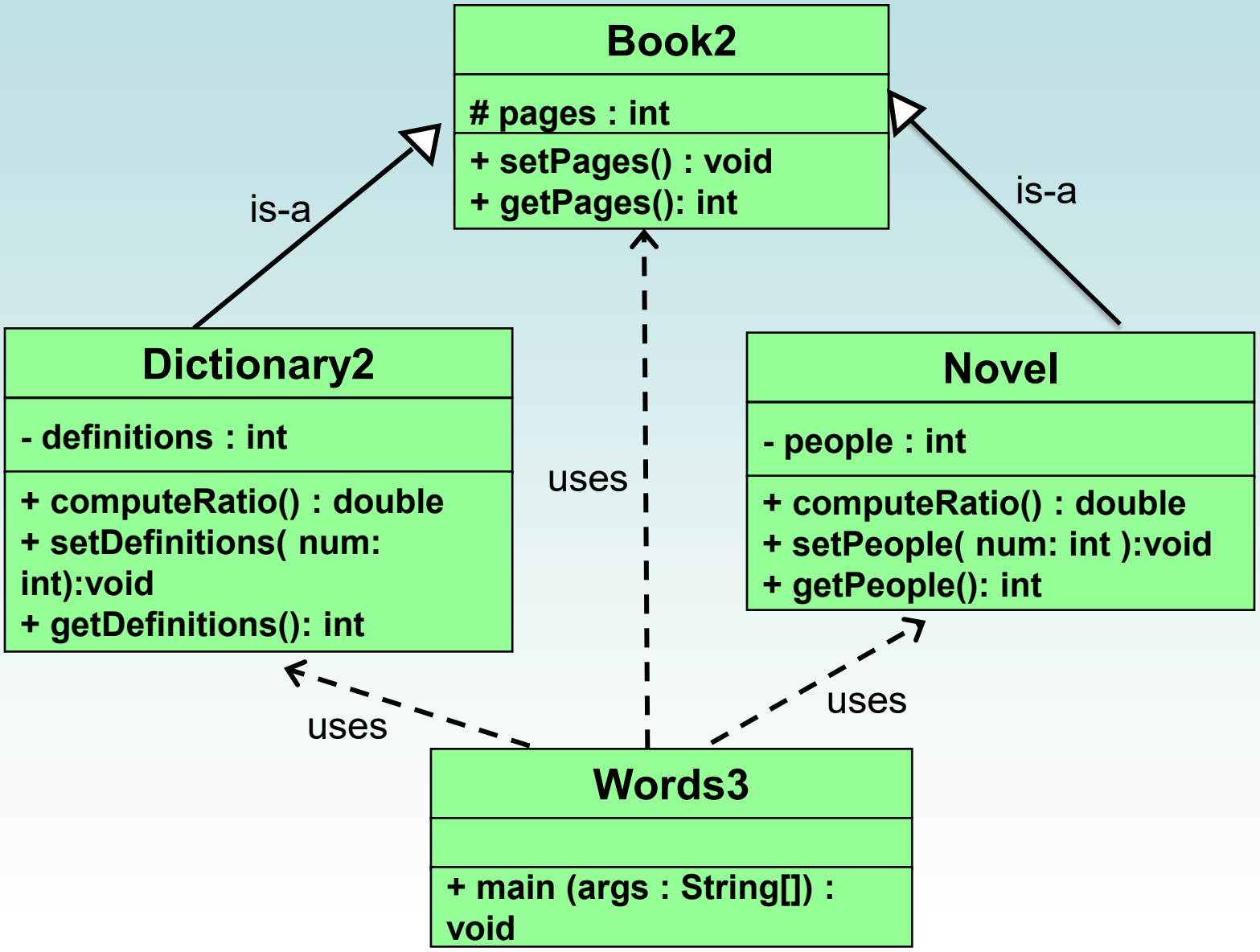
Class Hierarchies

A child class of a parent can be the parent of a 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



```
//********************************************
//  Words3.java       Author: Lewis/Loftus, H
//********************************************

public class Words3
{
   public static void main(String[] args)
   {
      Book2 book = new Book2( 523 );
      System.out.println ("Number of pages:"

      Dictionary2 webster = new Dictionary2(1500, 52500);
      System.out.println("webster pages:" + webster.getPages());
      System.out.println("webster definitions:" + webster.getDefinitions());
      System.out.println("Definitions per page:" + webster.computeRatio());
      System.out.println("---------- " );

      Novel dickens = new Novel( 320, 32 );
      System.out.println("dickens pages: " + dickens.getPages());
      System.out.println("dickens people:" + dickens.getPeople());
      System.out.println("Pages per person:" + dickens.computeRatio());
   }
}
```

Output

```
Number of pages: 523
webster pages: 1500
webster definitions: 52500
Definitions per page: 35.0
-----------
dickens pages: 320
dickens people: 32
Pages per person: 10
```

Challenges: `add another book; add a title`

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

public class Book2
{
   protected int pages;

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

continue

continue

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

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

```
//********************************************************************
//  Dictionary2.java       SAME AS PREVIOUS EXAMPLE
//
//  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); // Invoke the parent's constructor

      definitions = numDefinitions;
   }
```

continue

continue

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

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

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

```java
//*************************************************************************
//  Novel.java
//*************************************************************************
public class Novel extends Book2
{
   private int people;

   public Novel(int numPages, int numPeople)
   {
      super(numPages); // Invoke the parent's constructor

      people = numPeople;
   }

   public double computeRatio()
   {
      return (double) pages/people;
   }

   public void setPeople(int numPeople)
   {
      people = numPeople;
   }

   public int getPeople()
   {
      return people;
   }
}
```

```
//*****************************************************************
//  Novel.java
//*****************************************************************
public class Novel extends Book2
{
   private int people;

   public Novel(int numPages, int numPeople)
   {
      super(numPages); // Invoke the parent's constructor

      people = numPeople;
   }

   public double computeRatio()
   {
      return (double) pages/people;
   }

   public void setPeople(int numPeople)
   {
      people = numPeople;
   }

   public int getPeople()
   {
      return people;
   }
}
```

```
/* copy to Words3.java */

public class Words3
{
   public static void main(String[] args)
   {
      Book2 book = new Book2( 523 );
      System.out.println ("Number of pages:" + book.getPages() );

      Dictionary2 webster = new Dictionary2(1500, 52500);
      System.out.println("webster pages:" + webster.getPages());
      System.out.println("webster definitions:" + webster.getDefinitions());
      System.out.println("Definitions per page:" + webster.computeRatio());
      System.out.println("---------- " );

      Novel dickens = new Novel( 320, 32 );
      System.out.println("dickens pages: " + dickens.getPages());
      System.out.println("dickens people:" + dickens.getPeople());
      System.out.println("Pages per person:" + dickens.computeRatio());
   }
}
```

```
C:\Users\kjell\OneDrive - CCSU\AAAAA>javac Words3.java

C:\Users\kjell\OneDrive - CCSU\AAAAA>dir
 Volume in drive C is OSDisk
 Volume Serial Number is 7E83-4517

 Directory of C:\Users\kjell\OneDrive - CCSU\AAAAA

05/19/2024  12:38 PM    <DIR>          .
05/19/2024  12:38 PM    <DIR>          ..
05/19/2024  12:38 PM               353 Book2.class
05/19/2024  12:23 PM               715 Book2.java
05/19/2024  12:38 PM               464 Dictionary2.class
05/19/2024  12:34 PM               703 Dictionary2.java
05/19/2024  12:38 PM               437 Novel.class
05/19/2024  12:19 PM               609 Novel.java
05/19/2024  12:38 PM             1,490 Words3.class
05/19/2024  12:19 PM               765 Words3.java
               8 File(s)          5,536 bytes
               2 Dir(s)  235,089,350,656 bytes free

C:\Users\kjell\OneDrive - CCSU\AAAAA>java Words3
Number of pages:523
webster pages:1500
webster definitions:52500
Definitions per page:35.0
----------
dickens pages: 320
dickens people:32
Pages per person:10.0

C:\Users\kjell\OneDrive - CCSU\AAAAA>
```

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 a child of the `Object` class
- Therefore, the `Object` class is the ultimate root of all class hierarchies

The Object Class

- The `Object` class contains 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 returns 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

  This is the same as what `==` does

- 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` to be more useful

```
//********************************************************************
//  Words4.java       Author: Lewis/Loftus, Hacked
//********************************************************************

public class Words4
{
   public static void main(String[] args)
   {
      Book4 bookA = new Book4( 523 );
      Book4 bookB = new Book4( 523 );

      if ( bookA == bookB ) // check if the two pointers are the same
        System.out.println("One object, two aliases" ) ;
      else
        System.out.println("Two objects") ; // they are

      if ( bookA.equals( bookB ) ) // using method defined in book
        System.out.println("Equivalent objects (or one object)") ;
      else
        System.out.println("Two non-equivalent objects") ;
   }
}
```

Output

```
Two objects
Equivalent objects
```

```
//********************************************************************
//  Book4.java       Author:
//
//********************************************************************

public class Book4
{
   protected int pages;

   //-----------------------------------------------------------------
   //  Constructor: Give the book the specified number of pages.
   //-----------------------------------------------------------------
   public Book4(int numPages)
   {
      pages = numPages;
   }

   //-----------------------------------------------------------------
   //  This method overrides
   //  the method inherited from Object
   //-----------------------------------------------------------------
   public boolean equals( Book4 other)
   {
      return pages == other.pages;
   }
}
```

Abstract Classes

- An *abstract class* is a placeholder in a class hierarchy that represents a generic concept
  - might have an abstract class Animal that includes features all animals must have
  - but don't want an object that is only Animal and nothing else
  - classes derived from Animal can be instantiated
    - Dog, Cat, Spider
- An abstract class cannot be instantiated

Abstract Classes

The modifier `abstract` on the class header declares a class as abstract:

```
public abstract class Animal
{
    // class contents
}
```

Abstract Classes

- An abstract class often contains abstract methods with no definitions (like an interface)
- Unlike an interface, the `abstract` modifier must be applied to each abstract method
- Also, an abstract class typically contains 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` or `static`
- Abstract classes are an important element of software design
  - used to establish common features in a hierarchy that are too general to instantiate

Interface Hierarchies

- Inheritance can be applied to interfaces
- 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)
```

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.

Outline

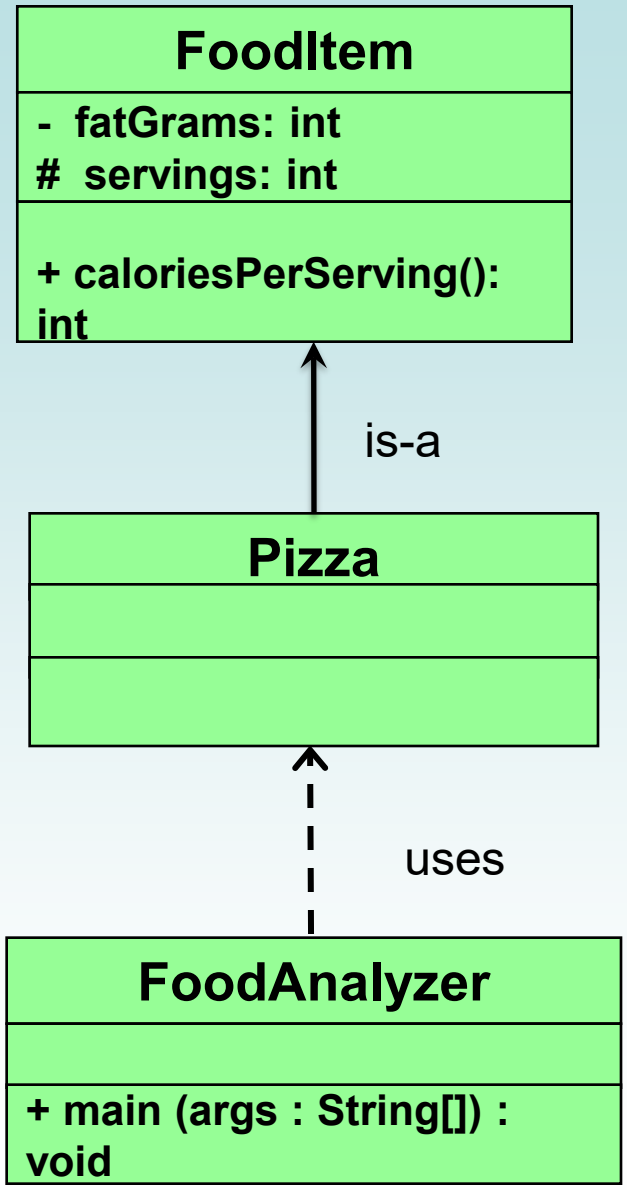
- Creating Subclasses
- Overriding Methods
- Class Hierarchies
- Visibility
- Designing for Inheritance
- Inheritance in JavaFX
- Color and Date Pickers
- Dialog Boxes

Visibility Revisited

- A subtle issue related to inheritance and visibility:
- All variables and methods of a parent class, even private members, are inherited by its children
  - but the child might not see them directly
- Private members in a parent class 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 a parent can refer to its private member, a child can reference it indirectly using its parent's methods
- The **`super`** reference can be used to refer to the parent class, even if no object of the parent exists
- See `FoodAnalyzer.java`
- See `FoodItem.java`
- See `Pizza.java`



```
//********************************************************************
//  FoodAnalyzer.java       Author: Lewis/Loftus
//
//  Demonstrates indirect access to inherited private members.
//********************************************************************

public class FoodAnalyzer
{
   //-----------------------------------------------------------------
   //  Instantiates a Pizza object and prints its calories per
   //  serving.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Pizza special = new Pizza(275);

      System.out.println("Calories per serving: " +
                         special.caloriesPerServing());
   }
}
```

```
//********************************************************************
//  FoodAnalyzer.java       Author: Lewis/Loftus
//
//  Demonstrates indirect access to
//********************************                                  **

public class FoodAnalyzer
{
   //-----------------------------------------------------------------
   //  Instantiates a Pizza object and prints its calories per
   //  serving.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      Pizza special = new Pizza(275); // No FoodItem object exists

      System.out.println("Calories per serving: " +
                         special.caloriesPerServing());
                         // but we can use this method defined in it
   }
}
```

Output

```
Calories per serving: 309
```

```java
//********************************************************************
//  FoodItem.java       Author: Lewis/Loftus
//
//********************************************************************

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);
   }
}
```

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

public class Pizza extends FoodItem
{
   //-----------------------------------------------------------------
   //  Sets up a pizza with the specified amount of fat (assumes
   //  eight servings).
   //
   //  fatGrams in the parent class is private, so Pizza objects
   //  cannot use it directly.
   //-----------------------------------------------------------------
   public Pizza(int fatGrams)
   {
      super(fatGrams, 8);
   }
}
```

Outline

Creating Subclasses

Overriding Methods

Class Hierarchies

Visibility

Designing for Inheritance

Inheritance in JavaFX

Color and Date Pickers

Dialog Boxes

Designing for Inheritance

- Take the time to create a good software design
- Inheritance is an important part of an object-oriented design
- Properly designed inheritance contribute to the elegance, maintainability, and reuse of the software

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 avoid (shadow) inherited variables

Inheritance Design Issues

- Allow each class to manage its own data; use the `super` reference to invoke the parent's constructor to set up its data
- Override general methods such as `toString` and `equals` with appropriate definitions
- Use abstract classes to represent general concepts that derived classes have in common
- Use visibility modifiers carefully to provide needed access without violating encapsulation

Restricting Inheritance

- If the **`final`** modifier is applied to a method, that method cannot be overridden in any derived classes
- If the **`final`** modifier is applied to an entire class, then that class cannot be used to derive any children at all
- Therefore, an abstract class cannot be declared as final

Outline

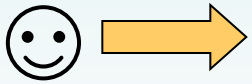
Creating Subclasses

Overriding Methods

Class Hierarchies

Visibility

Designing for Inheritance

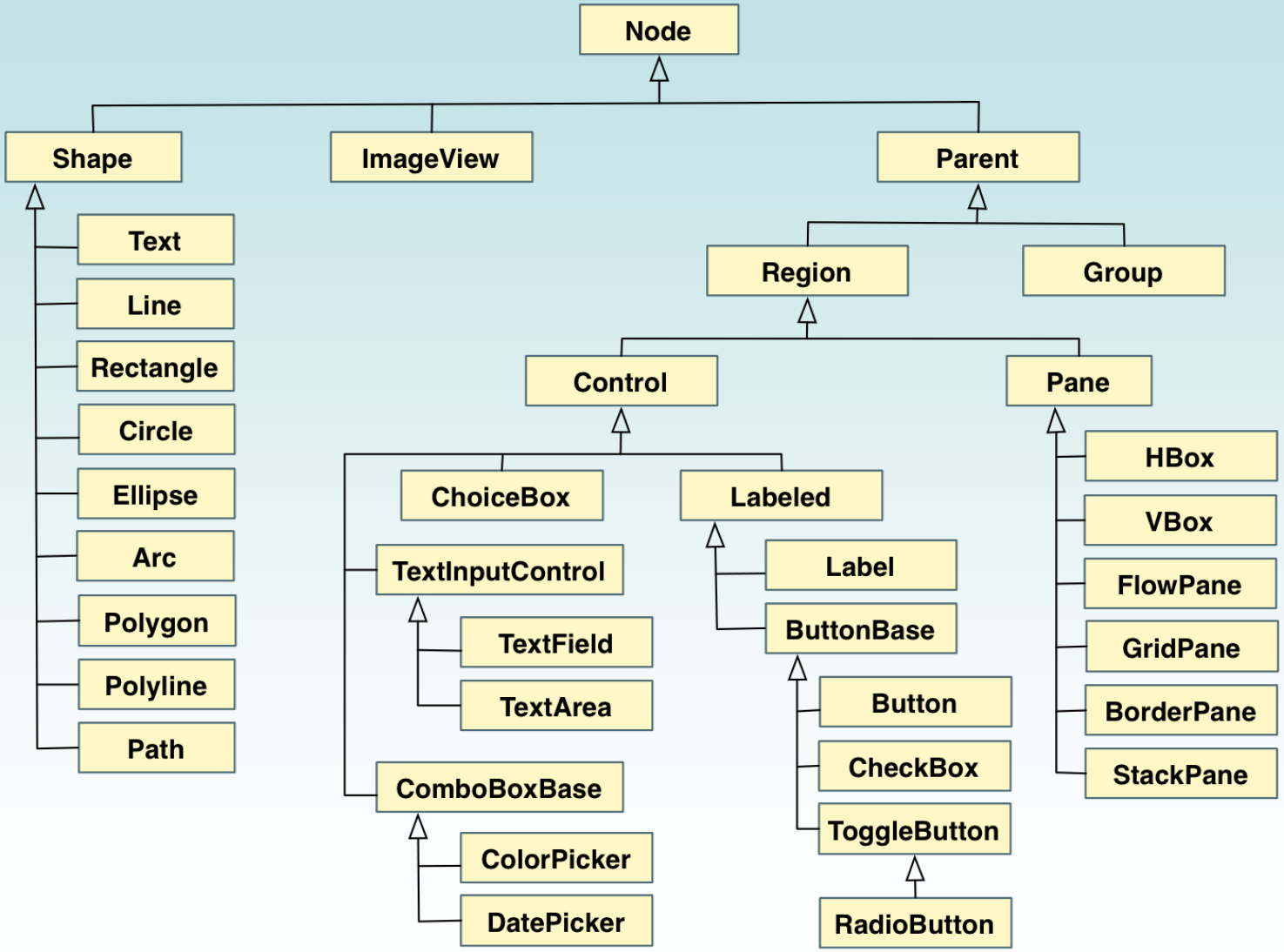


Inheritance in JavaFX

Color and Date Pickers

Dialog Boxes

Inheritance in JavaFX



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 be a root node of a scene
- So a `Pane` can be the root node of a scene, and could contain a `Circle` and a `Button`
  - `Pane` isa `Region` isa `Parent`
- A `ChoiceBox` could be the root node of a scene, but a `Rectangle` could not

Outline

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

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`

```
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;
```

continue

continue

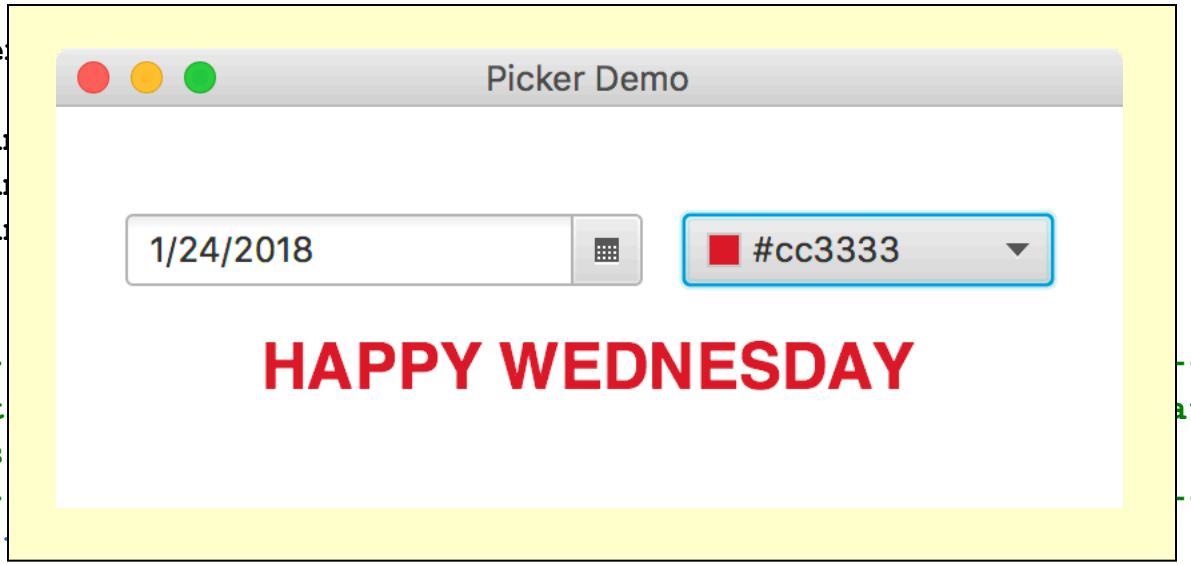
```
    //-----------------------------------------------------------------
    //  Allows the user to select a date and a color. A Text object
    //  displays the day of the week in the color specified.
    //-----------------------------------------------------------------
    public void start(Stage 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);
```

continue

continue

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

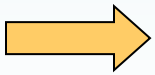
continue




```
        Sce
        pri
        pri
        pri
    }

    //-----                                                    ----------
    //  Get                                                   ates the
    //  mes
    //-----                                                    ----------
    public
    {
        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());
    }
}
```

Outline

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



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

continue

```
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();
```

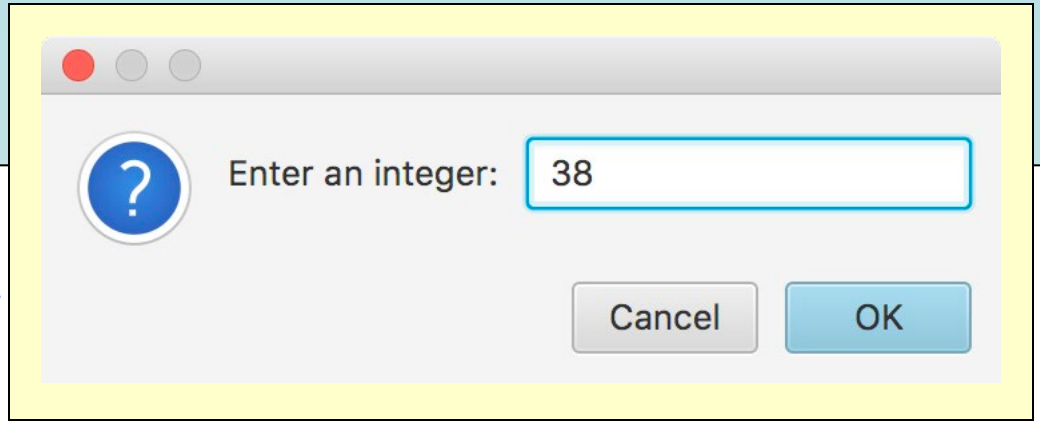
continue

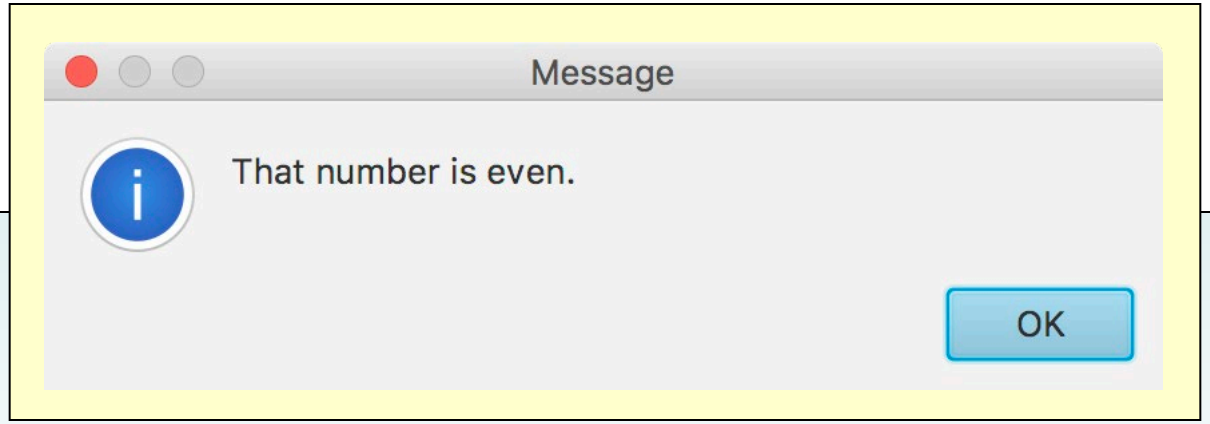
continue

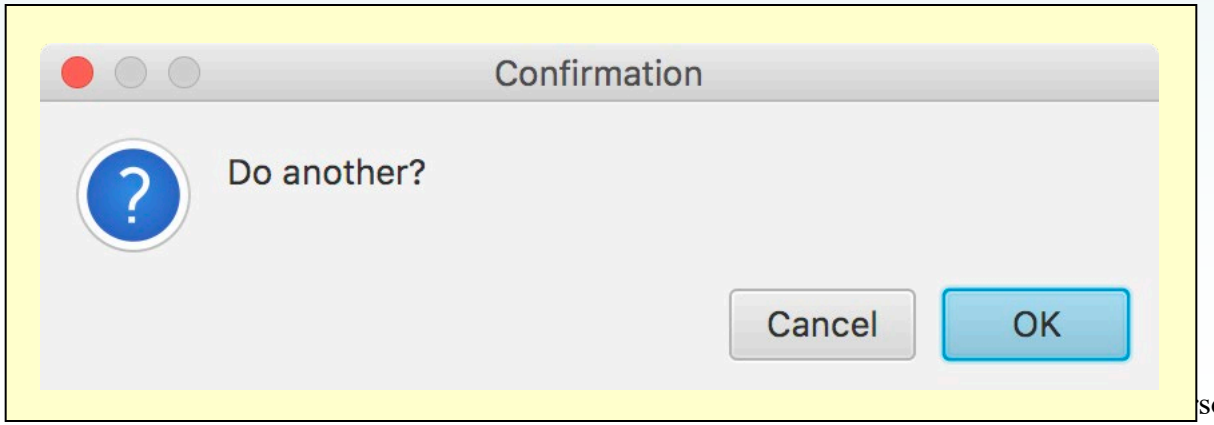
```
                if (another.get() != ButtonType.OK)
                    doAnother = false;
            }
            else
                doAnother = false;
        }
    }
}
```

continue

```
                    if
                }
                else

            }
        }
}
```







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`

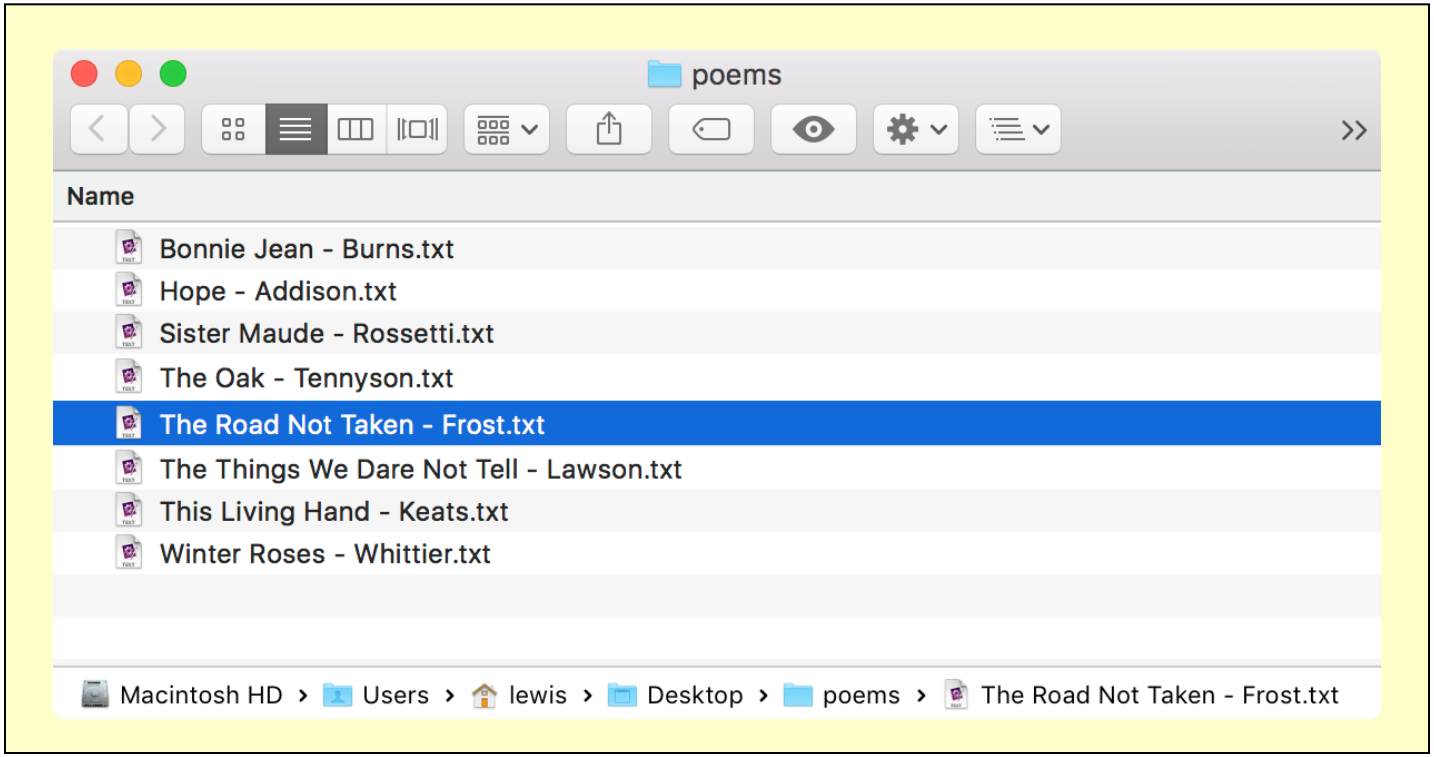
```
import java.io.File;
import java.io.IOException;
import java.util.Scanner;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.TextArea;
import javafx.scene.text.Font;
import javafx.stage.FileChooser;
import javafx.stage.Stage;

//*****************************************************************
//  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

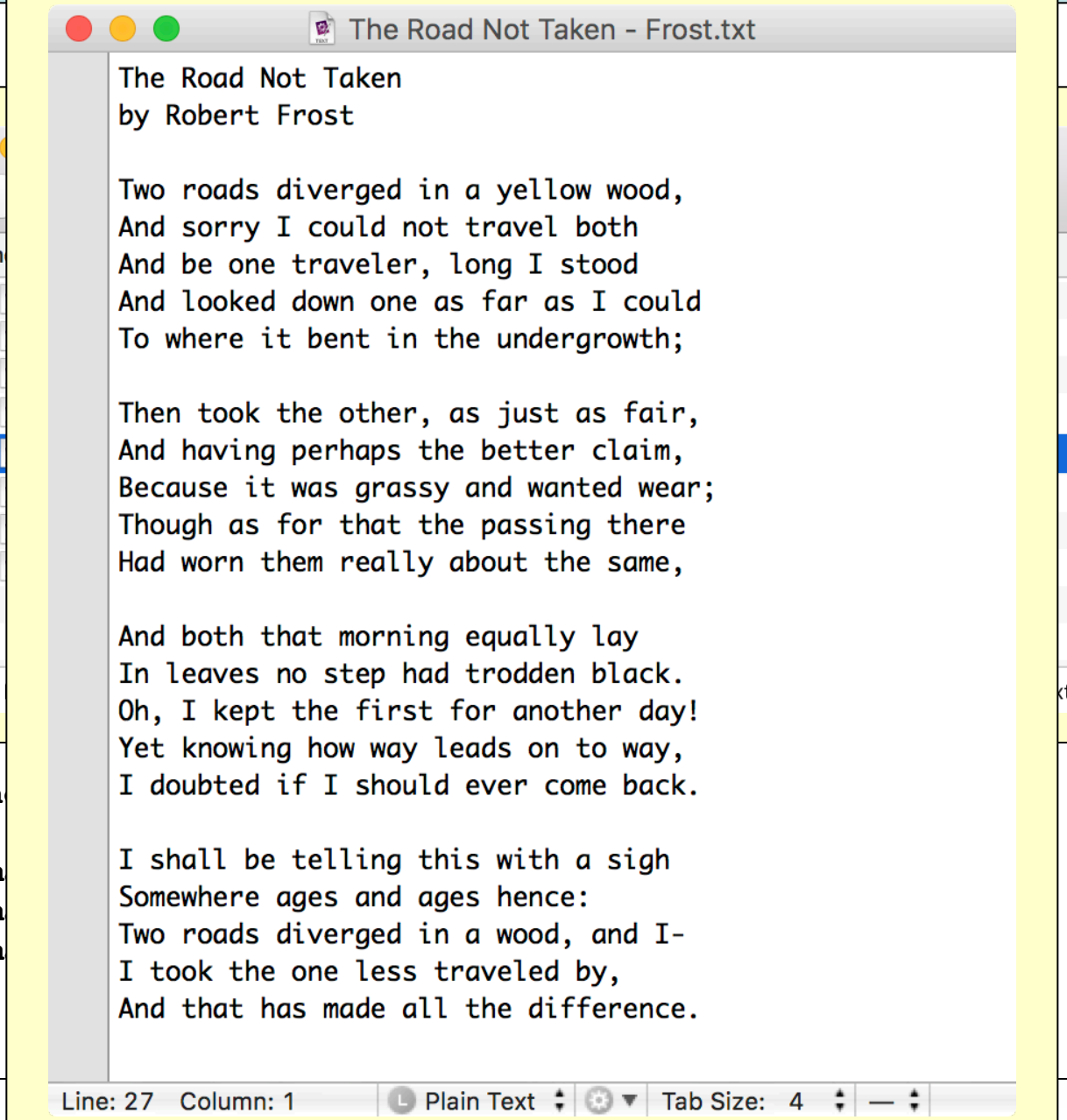
```
        TextArea content = new TextArea();
        content.setFont(new Font("Courier", 12));
        content.setEditable(false);

        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




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

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

continue




```
        Scen

        prim
        prim
        prim
    }
}
```

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