Inheritance

Chapter 8

5TH EDITION

Lewis & Loftus

java

Software Solutions

Foundations of Program Design

PEARSON
Addison
Wesley

Inheritance

- **Inheritance is a fundamental object-oriented design technique used to create and organize reusable classes**

- **Chapter 8 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 GUI component class hierarchy**
  - **extending listener adapter classes** will not ask about adapter class and timer class
  - **the `Timer` class**

Outline

- **Creating Subclasses**
- **Overriding Methods**
- **Class Hierarchies**
- **Inheritance and Visibility**
- **Designing for Inheritance**
- **Inheritance and GUIs**
- **The `Timer` Class**

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*** have multiple names

can also call it the derived class

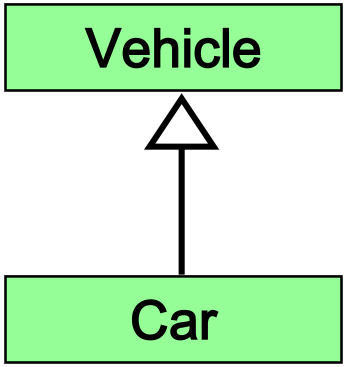
- **The derived class is called the *child class* or *subclass***

- **As the name implies, the child inherits characteristics of the parent**

- **That is, the child class inherits the methods and data defined by the parent class**

Inheritance

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



parent is pointed to by the child

more specific case of the parent

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

Inheritance

- **A programmer can tailor a derived class as needed by adding new variables or methods, or by modifying the inherited ones**

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

- **By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software**

Deriving Subclasses

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

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

only fully defined classes can have children

- **See `Words.java` (page 442)**
- **See `Book.java` (page 443)**
- **See `Dictionary.java` (page 444)**

The protected Modifier

- Visibility modifiers affect the way that class members can be used in a child class

- Variables and methods declared with private visibility cannot be referenced by name in a child class

- They can be referenced in the child class if they are declared with public visibility -- but public variables violate the principle of encapsulation

- There is a third visibility modifier that helps in inheritance situations: `protected`

if something is protected, then things that inherit from that class can access it - but the outside world wont be able to access it.
when there is no modifier, then anything within that package can access it - that means anything within the same directory

The protected Modifier

- **The `protected` modifier allows a child class to reference a variable or method directly in the child class**

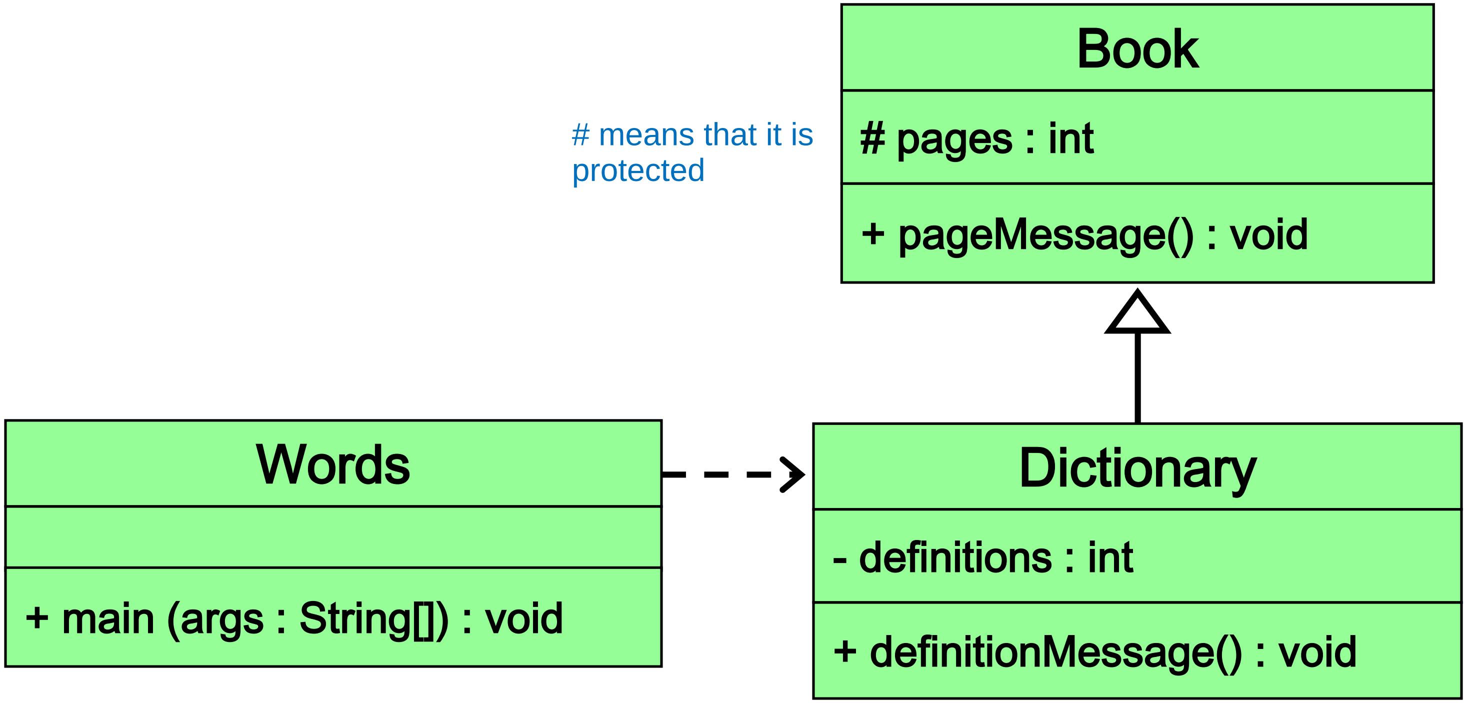
- **It provides more encapsulation than public visibility, but is not as tightly encapsulated as private visibility**

- **A protected variable is visible to any class in the same package as the parent class**

- **The details of all Java modifiers are discussed in Appendix E**

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

Class Diagram for Words



means that it is protected

The super Reference

- **Constructors are not inherited, even though they have public visibility**
- **Yet we often want to use the parent's constructor to set up the "parent's part" of the object**
- **The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor**

use super to construct the parent(you can only inherit one class in java)
it has to be thee first line of the child's constructor

so when you call super, it really means call the parent's constructor.
you use this so that the setup that the parent does you don't have to redo it in the child

- **See `Words2.java` (page 447)**
- **See `Book2.java` (page 448)**
- **See `Dictionary2.java` (page 449)**

The super Reference

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

- **The first line of a child's constructor should use the `super` reference to call the parent's constructor**

- **The `super` reference can also be used to reference other variables and methods defined in the parent's class**

Multiple Inheritance

- **Java supports *single inheritance*, meaning that a derived class can have only one parent class**
- ***Multiple inheritance* allows a class to be derived from two or more classes, inheriting the members of all parents**
- **Collisions, such as the same variable name in two parents, have to be resolved**
- **Java does not support multiple inheritance**
- **In most cases, the use of interfaces gives us aspects of multiple inheritance without the overhead**

Outline

Creating Subclasses

➡ Overriding Methods

Class Hierarchies

Inheritance and Visibility

Designing for Inheritance

Inheritance and GUIs

The `Timer` Class

Overriding Methods

- **A child class can *override* the definition of an inherited method in favor of its own**

can use super to call paren'ts overriden method.
can only go back one generation: can't do super.super to get grandparent's method

- **The new method must have the same signature as the parent's method, but can have a different body**
- **The type of the object executing the method determines which version of the method is invoked**
- **See `Messages.java` (page 452)**
- **See `Thought.java` (page 453)**
- **See `Advice.java` (page 454)**

break 11/2/17

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** if you are final, you cannot have children
- **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**

can you shadow final variables?

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

Outline

Creating Subclasses

Overriding Methods

⇨ Class Hierarchies

Inheritance and Visibility

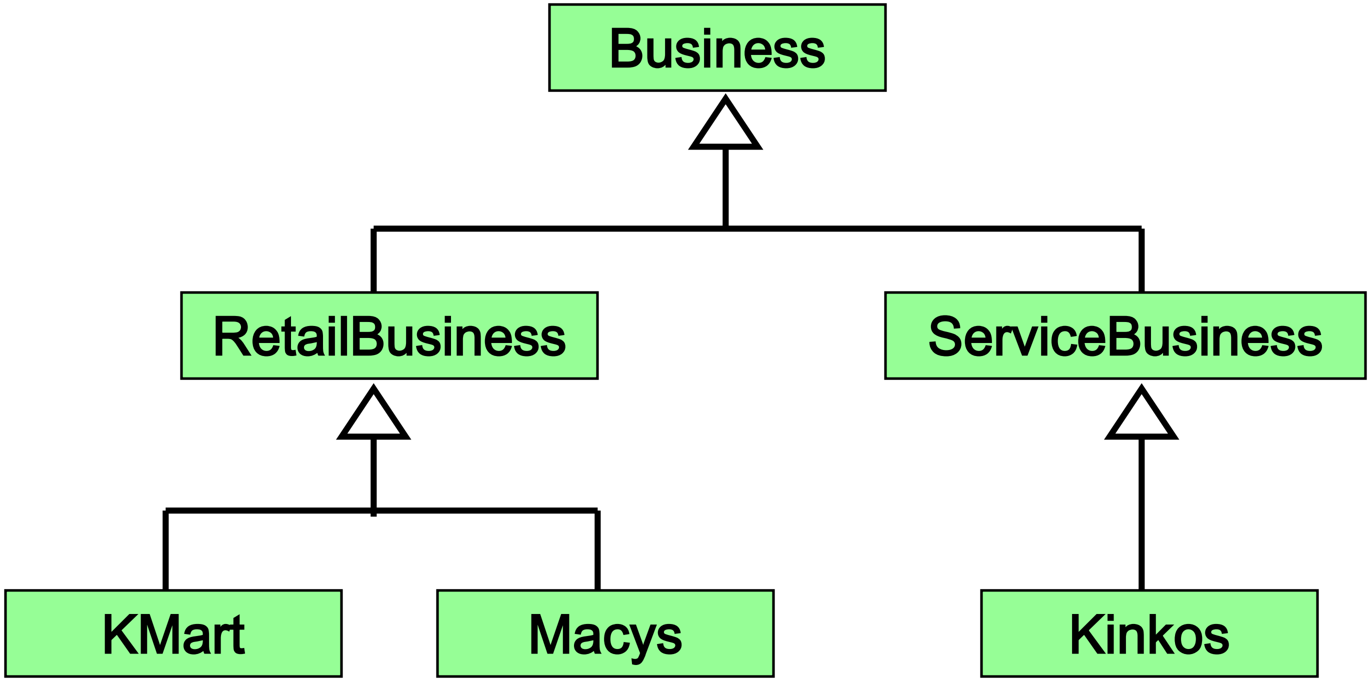
Designing for Inheritance

Inheritance and GUIs

The `Timer` Class

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**

try to put the most common on as far back as possible

- **An inherited member is passed continually down the line**
- **Therefore, a child class inherits from all its ancestor classes**
- **There is no single class hierarchy that is appropriate for all situations**

The Object Class

- **A class called `Object` is defined in the `java.lang` package of the Java standard class library**

- **All classes are derived from the `Object` class**

- **If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class**

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

The Object Class

- **The `Object` class contains a few useful methods, which are inherited by all classes**
- **For example, the `toString` method is defined in the `Object` class**
- **Every time we define the `toString` method, we are actually overriding an inherited definition**
- **The `toString` method in the `Object` class is defined to return a string that contains the name of the object's class along with some other information**

The Object Class

- **The `equals` method of the `Object` class returns true if two references are aliases**

- **We can override `equals` in any class to define equality in some more appropriate way**

- **As we've seen, the `String` class defines the `equals` method to return true if two `String` objects contain the same characters**

- **The designers of the `String` class have overridden the `equals` method inherited from `Object` in favor of a more useful version**

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:

```
public abstract class Product
{
   // 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`**

- **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 generic to instantiate**

Interface Hierarchies

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

Outline

Creating Subclasses

Overriding Methods

Class Hierarchies

➡ Inheritance and Visibility

Designing for Inheritance

Inheritance and GUIs

The `Timer` Class

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

the variable is inheritated, but there is no direct access to it. you will have to use a public method to access the private things

- **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` (page 460)**
- **See `FoodItem.java` (page 461)**
- **See `Pizza.java` (page 462)**

end 11/2/17

Outline

Creating Subclasses

Overriding Methods

Class Hierarchies

Inheritance and Visibility

⇨ **Designing for Inheritance**

Inheritance and GUIs

The `Timer` Class

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, maintainabilty, and reuse of the software**
- **Let's summarize some of the issues regarding inheritance that relate to a good software design**

Inheritance Design Issues

- **Every derivation should be an is-a relationship**
- **Think about the potential future of a class hierarchy, and design classes to be reusable and flexible**
- **Find common characteristics of classes and push them as high in the class hierarchy as appropriate**
- **Override methods as appropriate to tailor or change the functionality of a child**
- **Add new variables to children, but don't redefine (shadow) inherited variables**

Inheritance Design Issues

- Allow each class to manage its own data; use the `super` reference to invoke the parent's constructor to set up its data
- Even if there are no current uses for them, override general methods such as `toString` and `equals` with appropriate definitions
- Use abstract classes to represent general concepts that lower classes have in common
- Use visibility modifiers carefully to provide needed access without violating encapsulation

Restricting Inheritance

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

Outline

Creating Subclasses

Overriding Methods

Class Hierarchies

Inheritance and Visibility

Designing for Inheritance

➡ **Inheritance and GUIs**

The `Timer` Class

The Component Class Hierarchy

- **The Java classes that define GUI components are part of a class hierarchy**
- **Swing GUI components typically are derived from the `JComponent` class which is derived from the `Container` class which is derived from the `Component` class**
- **Many Swing components can serve as (limited) containers, because they are derived from the `Container` class**
- **For example, a `JLabel` object can contain an `ImageIcon`**

The Component Class Hierarchy

- An applet is a good example of inheritance
- Recall that when we define an applet, we extend the `Applet` class or the `JApplet` class
- The `Applet` and `JApplet` classes already handle all the details about applet creation and execution, including:
  - interaction with a Web browser
  - accepting applet parameters through HTML
  - enforcing security restrictions

The Component Class Hierarchy

- **Our applet classes only have to deal with issues that specifically relate to what our particular applet will do**
- **When we define `paintComponent` method of an applet, we are actually overriding a method defined originally in the `JComponent` class and inherited by the `JApplet` class**

Event Adapter Classes

- **Inheritance also gives us a alternate technique for creating listener classes**
- **We've seen that listener classes can be created by implementing a particular interface, such as `MouseListener`**
- **We can also create a listener class by extending an *event adapter class***
- **Each listener interface that has more than one method has a corresponding adapter class, such as the `MouseAdapter` class**

Event Adapter Classes

- Each adapter class implements the corresponding listener and provides empty method definitions
- When you derive a listener class from an adapter class, you only need to override the event methods that pertain to the program
- Empty definitions for unused event methods do not need to be defined because they are provided via inheritance
- See `OffCenter.java` (page 467)
- See `OffCenterPanel.java` (page 468)

Outline

Creating Subclasses

Overriding Methods

Class Hierarchies

Inheritance and Visibility

Designing for Inheritance

Inheritance and GUIs

➡ The `Timer` Class

The Timer Class

- The `Timer` class of the `javax.swing` package is a GUI component, but it has no visual representation
- A `Timer` object generates an action event at specified intervals
- Timers can be used to manage any events that are based on a timed interval, such as an animation
- To create the illusion of movement, we use a timer to change the scene after an appropriate delay

The Timer Class

- The `start` and `stop` methods of the `Timer` class start and stop the timer

- The delay can be set using the `Timer` constructor or using the `setDelay` method

- See `Rebound.java` (page 472)
- See `ReboundPanel.java` (page 473)

Summary

- **Chapter 8 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 GUI component class hierarchy**
  - **extending listener adapter classes**
  - **the `Timer` class**