Java Programming, 9e

Chapter 11

Advanced Inheritance Concepts

Objectives

- Create and use abstract classes
- Use dynamic method binding
- Create arrays of subclass objects
- Use the `Object` class and its methods
- Use inheritance to achieve good software design
- Create and use interfaces
- Describe anonymous inner classes and lambda expressions
- Describe packages

Creating and Using Abstract Classes (1 of 3)

- **Abstract class**
  - Cannot create any concrete objects
  - Can inherit
  - Usually has one or more empty abstract methods
- When declaring an abstract class:
  - Use the keyword `abstract`
  - Provide the superclass from which other objects can inherit
  - Example:

```
public abstract class Animal
```

Creating and Using Abstract Classes (2 of 3)

- An **abstract method** does not have:
  - A body
  - Curly braces
  - Method statements
- To create an abstract method:
  - Use the keyword `abstract`
  - The header must include the method type, name, and parameters
  - Include a semicolon at the end of the declaration
  - Example:

```
public abstract void speak();
```

Creating and Using Abstract Classes (3 of 3)

- Subclass of abstract class
  - Inherits the abstract method from its parent
    - Must provide the implementation for the inherited method or be abstract itself
  - Code a subclass method to override the empty superclass method

Using Dynamic Method Binding (1 of 3)

- Every subclass object "is a" superclass member
  - Convert subclass objects to superclass objects
  - Can create a reference to a superclass object
    - Create a variable name to hold the memory address
    - Store a concrete subclass object
    - Example:

```
Animal animalRef;
animalRef = new Cow();
```

Using Dynamic Method Binding (2 of 3)

- **Dynamic method binding**
  - Also called **late method binding**
  - An application's ability to select the correct subclass method
  - Makes programs flexible
- When an application executes, the correct method is attached (or bound) to the application based on current and changing (dynamic) context

Using Dynamic Method Binding (3 of 3)

```
public class AnimalReference
{
   public static void main(String[] args)
   {
      Animal animalRef;
      animalRef = new Cow();
      animalRef.speak();
      animalRef = new Dog();
      animalRef.speak();
   }
}
```

Figure 11-8 The AnimalReference application

Using a Superclass as a Method Parameter Type (1 of 3)

- Useful when you want to create a method that has one or more parameters that might be one of several types
- Use dynamic method binding

```
public static void talkingAnimal(Animal animal)
Dog dog = new Dog();
talkingAnimal(dog);
```

Using a Superclass as a Method Parameter Type (2 of 3)

```
public class TalkingAnimalDemo
{
   public static void main(String[] args)
   {
      Dog dog = new Dog();
      Cow cow = new Cow();
      dog.setName("Ginger");
      cow.setName("Molly");
      talkingAnimal(dog);
      talkingAnimal(cow);
   }
   public static void talkingAnimal(Animal animal)
   {
      System.out.println("Come one. Come all.");
      System.out.println
         ("See the amazing talking animal!");
      System.out.println(animal.getName() +
         " says");
      animal.speak();
      System.out.println("***************");
   }
}
```

This method can accept any object that descends from Animal.

Figure 11-10 The TalkingAnimalDemo class

Using a Superclass as a Method Parameter Type (3 of 3)

```
Come one. Come all.
See the amazing talking animal!
Ginger says
Woof!
****************
Come one. Come all.
See the amazing talking animal!
Molly says
Moo!
****************
```

Figure 11-11 Output of the `TalkingAnimalDemo` application

Creating Arrays of Subclass Objects

- Create a superclass reference
  - Treat subclass objects as superclass objects
    - Create an array of different objects that share the same ancestry
- Create an array of three `Animal` references

```
Animal[] animalRef = new Animal[3];
```

  - Reserve memory for three `Animal` object references

Using the `Object` Class and Its Methods (1 of 2)

- **`Object` class**
  - Every Java class is an extension of the `Object` class
  - Defined in the `java.lang` package
  - Imported automatically
  - Includes methods to use or override

Using the `Object` Class and Its Methods (2 of 2)

Table 11-1: Object Class Methods	
Method	**Description**
Object clone()	Creates and returns a copy of this object
boolean equals (Object obj)	Indicates whether some object is equal to the parameter object (this method is described in detail below)
void finalize()	Called by the garbage collector on an object when there are no more references to the object
Class<?> getClass()	Returns the class to which this object belongs at run time
int hashCode()	Returns a hash code value for the object (this method is described briefly below)
void notify()	Wakes up a single thread that is waiting on this object's monitor
void notifyAll()	Wakes up all threads that are waiting on this object's monitor
String toString()	Returns a string representation of the object (this method is described in detail below)
void wait()	Causes the current thread to wait until another thread invokes either the notify() method or the notifyAll() method for this object
void wait (long timeout)	Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed
void wait (long timeout, int nanos)	Causes the current thread to wait until another thread invokes the notify() or notifyAll() method for this obj1ect, or some other thread interrupts the current thread, or a certain amount of real time has elapsed

Using the `toString()` Method (1 of 3)

- **`toString()` method**
  - Converts an `Object` into a `String`
  - Contains information about the `Object`
  - Output:
    - Class name
    - @ sign
    - **Hash code (address in memory)**

Using the `toString()` Method (2 of 3)

- Write an overloaded version of the `toString()` method
  - Display some or all data field values for an object
  - Can be very useful in debugging a program
    - Display the `toString()` value
    - Examine its contents

Using the `toString()` Method (3 of 3)

```
public class TestBankAccount
{
   public static void main(String[] args)
   {
      BankAccount myAccount = new BankAccount(123, 4567.89);
      System.out.println(myAccount.toString());
   }
}
```

Figure 11-18 The `TestBankAccount` application

Using the `equals()` Method (1 of 4)

- **`equals()` method**
  - Takes a single argument
    - The same type as the invoking object
  - Returns a `boolean` value
    - Indicates whether objects are equal
  - Considers two objects of the same class to be equal only if they have the same hash code

Using the `equals()` Method (2 of 4)

- Example of the `equals()` method:

```
if(someObject.equals
  (someOtherObjectOfTheSameType))
  System.out.println("The objects are equal");
```

- To consider objects to be equal based on contents, you must write your own `equals()` method

Using the `equals()` Method (3 of 4)

- `Object` method `hashCode()`
  - Returns an integer representing the hash code
  - Whenever you override the `equals()` method:
    - You should override the `hashCode()` method as well
    - Equal objects should have equal hash codes

Using the `equals()` Method (4 of 4)

```
public class BankAccount
{
   private int acctNum;
   private double balance;
   public BankAccount(int num, double bal)
   {
      acctNum = num;
      balance = bal;
   }
   @Override
   public String toString()
   {
      String info = "BankAccount acctNum = " +
         acctNum + " Balance = $" + balance;
      return info;
   }
   public boolean equals(BankAccount secondAcct)
   {
      boolean result;
      if(acctNum == secondAcct.acctNum && balance == secondAcct.balance)
         result = true ;
      else
         result = false;
      return result;
   }
}
```

This `equals()` method overloads the one in the `Object` class. This method takes a `BankAccount` argument, but the one in the `Object` class takes an `Object` argument.

Figure 11-22 The `BankAccount` class containing its own `equals()` method

Using Inheritance to Achieve Good Software Design

- You can create powerful computer programs more easily if components are used either “as is” or with slight modifications
- Makes programming large systems more manageable
- Advantages of extendable superclasses
  - Save development time
    - Much code is already written
  - Save testing time
    - Superclass code is already tested
  - Programmers understand how a superclass works
  - A superclass maintains its integrity
    - The bytecode is not changed

Creating and Using Interfaces (1 of 8)

- **Multiple inheritance**
  - Inherit from more than one class
  - Prohibited in Java
  - Variables and methods in parent classes might have identical names
    - Creates conflict
    - Which class should `super` refer to when a child class has multiple parents?

Creating and Using Interfaces (2 of 8)

- **Interface**
  - An alternative to multiple inheritance
  - Looks like a class except all of its methods are implicitly `public` and `abstract`, and all of its data items are implicitly `public`, `abstract`, and `final`
  - A description of what a class does
  - Declares method headers

Creating and Using Interfaces (3 of 8)

```
public abstract class Animal
{
    private String name;
    public abstract void speak();
    public String getName()
    {
        return name;
    }
    public void setName(String animalName)
    {
        name = animalName;
    }
}
public class Dog extends Animal
{
    public void speak()
    {
        System.out.println("Woof!");
    }
}
```

Figure 11-26 The Animal and Dog classes

Creating and Using Interfaces (4 of 8)

```
public interface Worker
{
    public abstract void work();
}
```

Figure 11-27 The `Worker` interface

Creating and Using Interfaces (5 of 8)

```
public class WorkingDog extends Dog implements Worker
{
   private int hoursOfTraining;
   public void setHoursOfTraining(int hrs)
   {
      hoursOfTraining = hrs;
   }
   public int getHoursOfTraining()
   {
      return hoursOfTraining;
   }
   public void work()
   {
      speak();
      System.out.println("I am a dog who works");
      System.out.println("I have " + hoursOfTraining +
         " hours of professional training!");
   }
}
```

Figure 11-28 The WorkingDog class

Creating and Using Interfaces (6 of 8)

- Create an interface
  - Example:
    ```
    public interface Worker
    ```
- Implement an interface
  - Use the keyword `implements`
    - Requires the subclass to implement its own version of each method
  - Use the interface name in the class header
    - Requires class objects to include code

    ```
    public class WorkingDog extends Dog implements Worker
    ```

Creating and Using Interfaces (7 of 8)

- Abstract classes versus interfaces
  - You cannot instantiate concrete objects of either
  - Abstract classes
    - Can contain nonabstract methods
    - Provide data or methods that subclasses can inherit
      - Subclasses maintain the ability to override inherited methods
    - Can include methods that contain the actual behavior the object performs

Creating and Using Interfaces (8 of 8)

- Abstract classes versus interfaces (cont'd.)
  - Interfaces
    - Methods must be abstract
    - Programmers know what actions to include
    - Every implementing class defines the behavior that must occur when the method executes
    - A class can implement behavior from more than one parent

Creating Interfaces to Store Related Constants (1 of 2)

- Interfaces can contain data fields
  - Data fields must be `public`, `static`, and `final`
- Interfaces contain constants
  - Provide a set of data that many classes can use without having to redeclare values

Creating Interfaces to Store Related Constants (2 of 2)

```
public interface PizzaConstants
{
    public static final int SMALL_DIAMETER = 12;
    public static final int LARGE_DIAMETER = 16;
    public static final double TAX_RATE = 0.07;
    public static final String COMPANY = "Antonio's Pizzeria";
}
```

Figure 11-31 The PizzaConstants interface

Using Anonymous Inner Classes and Lambda Expressions (1 of 4)

- **Anonymous inner class**
  - Unnamed
  - Defined inside another class

```
public interface Worker
{
    public abstract void work();
}
```

- **Effectively final variable**
  - Value assigned only once

Using Anonymous Inner Classes and Lambda Expressions (2 of 4)

```
import java.util.Scanner;
public class DemoAnonymousClass
{
   public static void main(String[] args)
   {
      int pounds;
      Scanner input = new Scanner(System.in);
      System.out.print("Enter capacity for washing machine" +
         " in pounds of laundry >> ");
      pounds = input.nextInt();
      Worker washingMachine = new Worker()
      {
         public void work()
         {
            System.out.println("I get clothes clean");
            System.out.println(" and can handle " + pounds +
               " pounds of laundry at a time.");
         }
      };
      washingMachine.work();
   }
}
```

Figure 11-36 The DemoAnonymousClass program

Using Anonymous Inner Classes and Lambda Expressions (3 of 4)

- **Lambda Expression**
  - Creates an object that implements a function interface
- **Lambda Operator**
  - **“->”**

Using Anonymous Inner Classes and Lambda Expressions (4 of 4)

```
import java.util.Scanner;
public class DemoLambda
{
   public static void main(String[] args)
   {
      int pounds;
      Scanner input = new Scanner(System.in);
      System.out.print("Enter capacity for washing machine" +
         " in pounds of laundry >> ");
      pounds = input.nextInt();
      Worker washingMachine = () ->
      {
          System.out.println("I get clothes clean");
          System.out.println(" and can handle " + pounds +
             " pounds of laundry at a time.");
      };
      washingMachine.work();
   }
}
```

Figure 11-38 The DemoLambda program

Creating and Using Packages (1 of 4)

- Package
  - A named collection of classes
  - Easily imports related classes into new programs
  - Encourages other programmers to reuse software
  - Helps avoid naming conflicts or collisions
  - Gives every package a unique name

Creating and Using Packages (2 of 4)

- Create classes for others to use
  - Protect your work
    - Do not provide users with source code in files with .java extensions
    - Provide users with compiled files with .class extensions
  - Include the `package` statement at the beginning of the class file
    - Place compiled code into the indicated folder
- If you do not specify a package the class is placed in an unnamed **default package**

Creating and Using Packages (3 of 4)

- Compile the file to place in a package
  - Use a compiler option with the `javac` command
    - The `-d` option places the generated .class file in a folder
- Package-naming convention
  - Use your Internet domain name in reverse order
- Collisions
  - Class naming conflicts

Creating and Using Packages (4 of 4)

- **Java ARchive (JAR) file**
  - A package or class library is delivered to users as a JAR file
  - Compresses and stores data
    - Reduces the size of archived class files
  - Based on the Zip file format

Don’t Do It

- Don’t write a body for an abstract method
- Don’t forget to end an abstract method header with a semicolon
- Don’t forget to override any abstract methods in any subclasses you derive
- Don’t mistakenly overload an abstract method instead of overriding it
- Don’t try to instantiate an abstract class object
- Don’t forget to override all the methods in an interface that you implement
- Don’t use the wildcard format to import multiple classes when creating your own packages

Summary (1 of 2)

- Abstract class
  - A class that you create only to extend from, but not to instantiate from
  - Usually contains abstract methods
    - Methods with no method statements
- Can convert subclass objects to superclass objects
- Dynamic method binding
  - Create a method that has one or more parameters that might be one of several types
  - Create an array of superclass object references but store subclass instances in it

Summary (2 of 2)

- Interface
  - Similar to a class
  - All methods are implicitly `public` and `abstract`
  - All of its data fields are implicitly `public`, `static`, and `final`
  - To create a class that uses an interface, include the keyword `implements` and the interface name in the class header
- Anonymous inner class
  - Has no name
  - Defined inside another class
  - Lambda expression creates an object that implements a functional interface
- Place classes in packages
  - Convention uses Internet domain names in reverse order