

# ADVANCED INHERITANCE CONCEPTS

**In this chapter, you will:**

- ◆ 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
- ◆ Create and use packages

Inheritance sure makes my programming job easier, you tell Lynn Greenbrier over a frosty lemonade at the Event Handlers Incorporated company picnic.

“So everything is going well, I take it?” says Lynn, smiling.

“It is,” you say as you smile back, “but I’m ready to learn more. What else can you tell me about inheritance?”

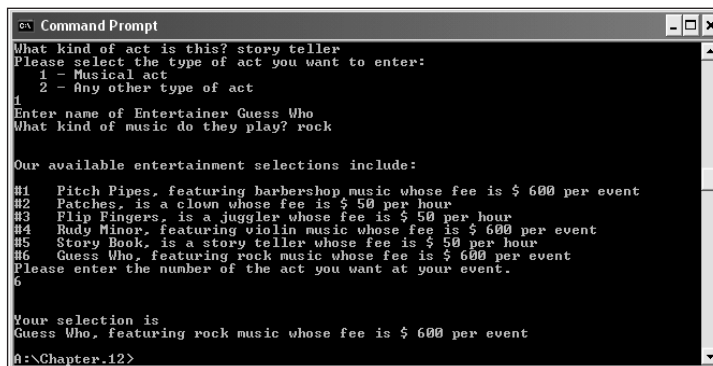
“Enjoy the picnic for today,” Lynn says. “On Monday morning I’ll teach you about superclass arrays that can use subclass methods, interfaces, and packages. Then you will be an inheritance pro.”

## PREVIEWING AN EXAMPLE OF USING AN ABSTRACT CLASS

Clients at Event Handlers Incorporated can choose many types of entertainment to feature at their events. Even though Event Handlers uses different class types to store different entertainment types, the different entertainment acts must be stored in a single entertainment database; this can be accomplished using methods introduced in this chapter. You can now use a completed version of the EntertainmentSelector program that is saved in the Chapter.12 folder on your Student Disk.

### To use the Chap12EntertainmentSelector class:

1. Go to the command prompt for the Chapter.12 folder on your Student Disk. Determine that the following four classes are in the folder: Chap12EntertainmentSelector, Entertainment, MusicalEntertainment, and OtherEntertainment. At the command prompt type **java Chap12EntertainmentSelector**, and then press **[Enter]**. This program allows you to supply data for six entertainment acts that are under contract to Event Handlers Incorporated. When you see the prompt, type **1** or **2** for a musical or non-musical weekend event, respectively.
2. If you indicated that this act is a musical act, the prompts will ask you for an act name and a style of music; if you indicated that this act is a non-musical act, the prompts will ask you for an act type. Supply any answers you want to these questions. After you enter information for six acts, the data you entered will echo to the screen and you will see the charge for each act.
3. Musical acts are paid by the event; non-musical acts are paid by the hour. Select the act you want to perform at your event. The last lines of a typical program run appear in Figure 12-1. (Yours may differ.) You will create a similar program in this chapter.



```

C:\> Command Prompt
What kind of act is this? story teller
Please select the type of act you want to enter:
    1 - Musical act
    2 - Any other type of act
1
Enter name of Entertainer Guess Who
What kind of music do they play? rock

Our available entertainment selections include:
#1 Pitch Pipes, featuring barbershop music whose fee is $ 600 per event
#2 Patchee, is a clown whose fee is $ 50 per hour
#3 Flip Fingers, is a juggler whose fee is $ 50 per hour
#4 Rudy Minor, featuring violin music whose fee is $ 600 per event
#5 Story Book, is a story teller whose fee is $ 50 per hour
#6 Guess Who, featuring rock music whose fee is $ 600 per event
Please enter the number of the act you want at your event.
6

Your selection is
Guess Who, featuring rock music whose fee is $ 600 per event
A:\Chapter.12>

```

Figure 12-1 Output of the Chap12EntertainmentSelector program

## CREATING AND USING ABSTRACT CLASSES

Creating new classes is easier after you understand the concept of inheritance. When you use a class as a basis from which to create extended child classes, the child classes are more specific than their parent. When you create a child class, it inherits all the general attributes you need; thus you must create only the new, more-specific attributes. For example, a `SalariedEmployee` and an `HourlyEmployee` are more specific than an `Employee`. They can inherit general attributes, such as an employee number, but they add specific attributes, such as pay-calculating methods.

Notice that a superclass contains the features that are shared by its subclasses. The subclasses are more-specific examples of the superclass type; they add additional features to the shared, general features. Conversely, when you examine a subclass, you see that its parent is more general and less specific. Sometimes a parent class is so general that you never intend to create any specific instances of the class. For example, you might never create “just” an `Employee`; each `Employee` is more specifically a `SalariedEmployee`, `HourlyEmployee`, or `ContractEmployee`. A class, such as `Employee`, that you create only to extend from, but not to instantiate from, is an abstract class. An **abstract class** is one from which you cannot create any concrete objects, but from which you can inherit. Abstract classes usually have one or more empty abstract methods. You use the keyword **`abstract`** when you declare an abstract class.



Nonabstract classes from which objects can be instantiated are called concrete classes.



In Chapter 11 you learned that you can create `final` classes if you do not want other classes to be able to extend them. Classes that you declare to be **`abstract`** are the opposite; your only purpose in creating them is to enable other classes to extend them.



In other programming languages, such as C++, abstract classes are known as virtual classes.

Abstract classes are like regular classes because they have data and methods but usually contain at least one abstract method. The difference is that you cannot create instances of abstract classes by using the **`new`** operator. You create abstract classes simply to provide a superclass from which other objects may inherit. Usually, abstract classes contain abstract methods. An **abstract method** is a method with no method statements. When you create an abstract method, you provide the keyword **`abstract`** and the intended method type, name, and arguments, but you do not provide any statements within the method. When you create a subclass that inherits an abstract method from a parent, you must provide the actions, or implementation, for the inherited method. It's important to

understand that you are required to code a subclass method to override the empty superclass method that is inherited. Programmers of abstract classes can include two method types: those that are implemented in the abstract class and simply inherited by its children, and those that are abstract and must be implemented by its children.



If you attempt to instantiate an object from an abstract class, you will receive an error message that you have committed an `InstantiationException`.



If you provide an empty method within an abstract class, the method is an abstract method even if you do not explicitly use the keyword `abstract` when defining the method.

Suppose you want to create classes to represent different animals, such as `Dog` and `Cow`. You can create a generic abstract class named `Animal` so you can provide generic data fields, such as the animal's name, only once. An `Animal` is generic, but all specific `Animals` make a sound. The actual sound differs from `Animal` to `Animal`. If you code an empty `speak()` method in the abstract `Animal` class, then you require all future `Animal` subclasses to code a `speak()` method that is specific to the subclass. Figure 12-2 shows an abstract `Animal` class containing a data field for the name, a constructor, a `getName()` method, and an abstract `speak()` method.

```
public abstract class Animal
{
    private String name;

    public Animal(String nm)
    {
        name = nm;
    }

    public String getName()
    {
        return(name);
    }

    public abstract void speak();
}
```

**Figure 12-2** Animal class

The `Animal` class in Figure 12-2 is declared as **abstract**. You cannot place a statement such as `Animal myPet = new Animal("Murphy");` within a program because the program will not execute. `Animal` is an abstract class, so no `Animal` objects can exist.



If you declare any method to be an abstract method, then you must also declare its class to be abstract.

You would create an abstract class such as `Animal` only so that you can extend it. For example, because a dog is an animal, you can create a `Dog` class as a child class of `Animal`. Figure 12-3 shows a `Dog` class; notice that it extends `Animal`. The `Animal` parent class in Figure 12-2 contains a constructor that requires a `String` holding the `Animal`'s name, so the child `Dog` class must also contain a constructor that passes a `String` to its superclass constructor.



You learned how child class and parent class constructors operate in Chapter 11.

```
public class Dog extends Animal
{
    public Dog(String nm)
    {
        super(nm);
    }
    public void speak()
    {
        System.out.println("Woof");
    }
}
```

**Figure 12-3** Dog class

The `speak()` method within the `Dog` class is required because the abstract, parent `Animal` class contains an abstract `speak()` method. You can code any statements you like within the `Dog speak()` method, but the `speak()` method must exist. Remember, you cannot instantiate an `Animal` object; however, instantiating a `Dog` object is perfectly legal because `Dog` is not an abstract class. When you code `Dog myPet = new Dog("Murphy");` you create a `Dog` object. Then when you code `myPet.speak();`, the correct `Dog speak()` method executes.



If you do not provide a subclass method to override a superclass abstract method, then you cannot instantiate any subclass objects. In this case, you also must declare the subclass itself to be abstract. Then you can extend the subclass into sub-subclasses where you write code for the method.

The classes in Figures 12-4 and 12-5 also inherit from the `Animal` class. When you create a `Cow` or a `Snake` object, each `Animal` will be able to use `speak()` appropriately.



In Chapter 11, you learned that using the same method name to indicate different implementations is called polymorphism. Using polymorphism, one method name causes different actions for different types of objects.

```
public class Cow extends Animal
{
    public Cow(String nm)
    {
        super(nm);
    }
    public void speak()
    {
        System.out.println("Moo");
    }
}
```

**Figure 12-4**    Cow class

```
public class Snake extends Animal
{
    public Snake(String nm)
    {
        super(nm);
    }
    public void speak()
    {
        System.out.println("Sss");
    }
}
```

**Figure 12-5**    Snake class

Next you will create an abstract Entertainment class for Event Handlers Incorporated. The Entertainment class holds data about entertainment acts that customers can hire for their events. The class includes fields for the name of the act and for the fee charged for providing the act. Entertainment is an abstract class. You will create two subclasses, MusicalEntertainment and OtherEntertainment. The more-specific classes include different methods for calculating the entertainment act's fee (musical acts are paid by the performance; other acts are paid by the hour), as well as different methods for displaying data.

**To create an abstract Entertainment class:**

1. Open a new file in your text editor and enter the following first two lines of an abstract Entertainment class:

```
public abstract class Entertainment
{
```

2. Define the two data fields that hold the entertainer's name and fee as **protected** rather than **private** because you want child classes to be able to access the fields when the fee is set and when the fields are shown on the screen. Define the fields as follows:

```
protected String entertainer;  
protected int fee;
```

3. The Entertainment constructor calls two methods. The first method accepts the entertainer's name from the keyboard. The second method sets the entertainer's fee. Because the first method accepts keyboard data entry, you must include the phrase **throws Exception** in the following constructor method header:

```
public Entertainment() throws Exception  
{  
    setEntertainerName();  
    setEntertainmentFee();  
}
```

4. Include the following two get methods that return the values for the entertainer's name and the act's fee:

```
public String getEntertainerName()  
{  
    return entertainer;  
}  
public double getEntertainmentFee()  
{  
    return fee;  
}
```

5. Enter the following **setEntertainerName()** method, which is similar to other data-entry methods you have coded in previous chapters. It prompts the user for the name of an entertainment act and assigns the characters to the entertainer field.

```
public void setEntertainerName() throws Exception  
{  
    String inputString = new String();  
    char newChar;  
    System.out.print("Enter name of entertainer ");  
    newChar = (char)System.in.read();  
    while(newChar >= 'A' && newChar <= 'z' || newChar == ' ' ||  
          newChar >= '0' && newChar <= '9')  
    {  
        inputString = inputString + newChar;  
        newChar = (char)System.in.read();  
    }  
    System.in.read();  
    entertainer = inputString;  
}
```

6. The `setEntertainmentFee()` method is an abstract method. Each subclass you eventually create that represents different entertainment types will have a different fee schedule. Type the abstract method definition and the closing curly brace for the class:

```
public abstract void setEntertainmentFee();
}
```

7. Save the file as **Entertainment.java** in the Chapter.12 folder on your Student Disk. At the command prompt, compile the file using the **javac** command.

You just created an abstract class, but you cannot instantiate any objects from this class. Rather, you must extend this class to be able to create any Entertainment-related objects. Next you will create a `MusicalEntertainment` class that extends the `Entertainment` class. This new class will be concrete; that is, you can create actual `MusicalEntertainment` class objects.

#### To create the `MusicalEntertainment` class:

1. Open a new file in your text editor, and then type the following header and opening brace for a `MusicalEntertainment` class that is a child of the `Entertainment` class:

```
public class MusicalEntertainment extends Entertainment
{
```

2. Add the definition of a music type field that is specific to musical entertainment by typing: **private String typeOfMusic;**
3. The `MusicalEntertainment` constructor must call its parent's constructor. It must also use the following method that sets the music type in which the entertainer specializes:

```
public MusicalEntertainment() throws Exception
{
    super();
    setTypeOfMusic();
}
```

4. Enter the following `setTypeOfMusic` method, which asks for user input:

```
public void setTypeOfMusic() throws Exception
{
    String inputString = new String();
    char newChar;
    System.out.print("What kind of music do they play? ");
    newChar = (char)System.in.read();
    while(newChar >= 'A' && newChar <= 'z' || newChar == ' ')
    {
        inputString = inputString + newChar;
        newChar = (char)System.in.read();
    }
    System.in.read();
    typeOfMusic = inputString;
}
```



5. Event Handlers Incorporated charges a flat rate of \$600 per event for musical entertainment. Add the following `setEntertainmentFee()` method to your program:

```
public void setEntertainmentFee()
{
    fee = 600;
}
```

6. Add the following `toString()` method that you can use when you want to convert the details of a `MusicalEntertainment` object into a `String` so you can easily and efficiently display the contents of the object. Add the closing curly brace for the class.



In Chapter 7, you first used the automatically included `toString()` method that converts objects to `Strings`. Now you are overriding that method for this class by writing your own version. You will learn more about the `toString()` method later in this chapter.

```
public String toString()
{
    return(entertainer + ", featuring " + typeOfMusic
    + " music whose fee is $ " + fee + " per event!");
}
```

7. Save the file as **MusicalEntertainment.java** in the Chapter.12 folder on your Student Disk, and then compile the file.

Event Handlers Incorporated classifies all non-musical entertainment acts, such as clowns, jugglers, and stand-up comics, as `OtherEntertainment`. The `OtherEntertainment` class inherits from `Entertainment`, just as the `MusicalEntertainment` class does. Whereas the `MusicalEntertainment` class requires a data field to hold the type of music played by the act, the `OtherEntertainment` class requires a field for the type of act. Other differences lie in the content of the prompt within the `setTypeOfAct()` method, and in the handling of fees. Event Handlers Incorporated charges \$50 per hour for non-musical acts, so both the `setEntertainmentFee()` and `toString()` methods differ from those in the `MusicalEntertainment` class.

Next you will create an `OtherEntertainment` class to implement the abstract method `setEntertainmentFee()`.

#### To create the `OtherEntertainment` class file:

1. Open a new file in your text editor, and then type the following first lines of the `OtherEntertainment` class:

```
public class OtherEntertainment extends Entertainment
{
```

2. Create the following String variable to hold the type of entertainment act (such as comedian): **private String typeOfAct;**
3. Enter the following code so the OtherEntertainment class constructor calls the parent constructor, then calls its own method to set the act type:

```
public OtherEntertainment() throws Exception
{
    super();
    setTypeOfAct();
}
```

4. Enter the following setTypeOfAct() method:

```
public void setTypeOfAct() throws Exception
{
    String inputString = new String();
    char newChar;
    System.out.print("What type of act is this? ");
    newChar = (char)System.in.read();
    while(newChar >='A' && newChar <= 'z' || newChar == ' ')
    {
        inputString = inputString + newChar;
        newChar = (char)System.in.read();
    }
    System.in.read();
    typeOfAct = inputString;
}
```

5. The fee for non-musical acts is \$50 per hour, so add the following setEntertainmentFee() method:

```
public void setEntertainmentFee()
{
    fee = 50;
}
```

6. Enter the following toString() method and add the closing curly brace for the class:

```
public String toString()
{
    return(entertainer + ", is a " + typeOfAct +
           " whose fee is $ " + fee + " per hour.");
}
```

7. Save the file as **OtherEntertainment.java** in the Chapter.12 folder on your Student Disk, and then compile the file.

Finally, you will create a program that instantiates concrete objects that belong to each of the two child classes.

To create a program that demonstrates using the `MusicalEntertainment` and `OtherEntertainment` classes:

1. Open a new file in your text editor, and then enter the `DemoEntertainment` class header, opening brace, `main()` method header, and its opening brace as follows:

```
public class DemoEntertainment
{
    public static void main(String[] args) throws Exception
    {
```

2. Enter the following statement that prompts the user to enter a musical act description. Then instantiate a `MusicalEntertainment` object.

```
System.out.println("Create a musical act description:");
MusicalEntertainment anAct = new MusicalEntertainment();
```

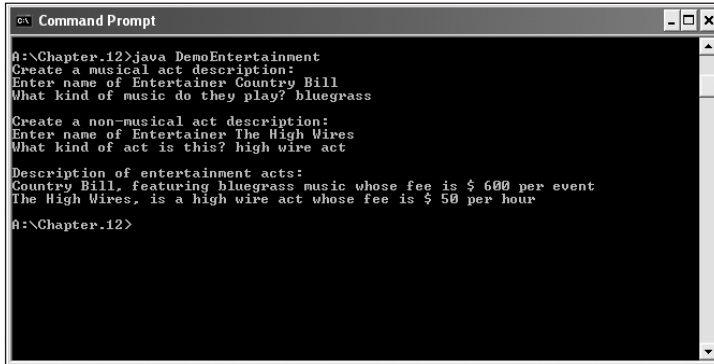
3. Enter the following similar statements for a non-musical act using the newline character `\n` to print on the next line:

```
System.out.println
    ("\nCreate a non-musical act description:");
OtherEntertainment anotherAct = new OtherEntertainment();
```

4. Enter the following lines to display the contents of the two objects and add the closing curly brace for the `main()` method and for the class:

```
System.out.println
    ("\nDescription of entertainment acts:");
System.out.println(anAct.toString());
System.out.println(anotherAct.toString());
}
```

5. Save the file as **DemoEntertainment.java** in the Chapter.12 folder on your Student Disk, and then compile the file. After the file compiles with no errors, run this program using the `java DemoEntertainment` command. When the program prompts you to do so, enter the name of a musical act, a type of music, the name of a non-musical act, and the type of act. Figure 12-6 shows a sample program run.



```

C:\>A:\Chapter.12>java DemoEntertainment
Create a musical act description:
Enter name of Entertainer Country Bill
What kind of music do they play? bluegrass

Create a non-musical act description:
Enter name of Entertainer The High Wires
What kind of act is this? high wire act

Description of entertainment acts:
Country Bill, featuring bluegrass music whose fee is $ 600 per event
The High Wires, is a high wire act whose fee is $ 50 per hour
A:\Chapter.12>

```

Figure 12-6 Output of the DemoEntertainment program

## USING DYNAMIC METHOD BINDING

When you create a superclass and one or more subclasses, each object of the subclass “is a” superclass object. Every `SalariedEmployee` “is an” `Employee`; every `Dog` “is an” `Animal`. (The opposite is not true. Superclass objects are not members of any of their subclasses. An `Employee` is not a `SalariedEmployee`. An `Animal` is not a `Dog`.) Because every subclass object “is a” superclass member, you can convert subclass objects to superclass objects.

As you are aware, when a superclass is abstract, you cannot instantiate objects of the superclass; however, you can indirectly create a reference to a superclass abstract method. A reference is not an object, but points to a memory address. When you create a reference, you do not use the keyword `new` to create a concrete object; you create a variable name in a subclass in which you can hold the memory address of a subclass concrete object that “is a” superclass member.



You learned how to create a reference in Chapter 4. When you code `someClass someObject;`, you are creating a reference. If you later code `someObject = new someClass();`, then you actually set aside memory for `someObject`.

If you create an `Animal` class, as shown in Figure 12-2, and various subclasses, as shown in Figures 12-3 through 12-5, then you create a new public class with a generic `Animal` reference variable into which you can assign any of the concrete `Animal` child objects. Figure 12-7 shows a new `AnimalReference` class, and Figure 12-8 shows its output. The variable `ref` is a type of `Animal`. No superclass `Animal` object is created, but instead, `Dog`, `Cow`, and `Snake` objects are created using the `new` keyword from classes extended from the `Animal` class. When the `Cow` object is assigned to the `Animal` reference, the `ref.speak()` method call results in “Moo”; when the `Dog` object is assigned to the `Animal` reference, the method call results in “Woof”.

```
public class AnimalReference
{
    public static void main(String[] args)
    {

        Animal ref;
        Cow aCow = new Cow("Mabel");
        Dog aDog = new Dog("Rover");
        Snake aSnake = new Snake("Siskal");

        ref = aCow;
        ref.speak();

        ref = aDog;
        ref.speak();

        ref = aSnake;
        ref.speak();

    }
}
```

Figure 12-7 AnimalReference class



Figure 12-8 Output of the AnimalReference program



Recall from Chapter 2 that when you assign a variable or constant of one type to a variable of another type, as in `doubleVar = intVar;`, the behavior is called casting.

The program in Figure 12-7 demonstrates polymorphic behavior. The same statement, `ref.speak();`, repeats after `ref` is set to each new animal type. Each call to the `speak()` method results in different output. Each reference “chooses” the correct `speak()` method based on the type of animal referenced. The program’s ability to select the correct subclass method is known as **dynamic method binding**. When the program executes, the

correct method is attached (or bound) to the program based on the current, changing context (dynamically).



Dynamic method binding is also called late binding.

---

## CREATING ARRAYS OF SUBCLASS OBJECTS

You might want to create a superclass reference and treat subclass objects as superclass objects so that you can create an array of different objects that share the same ancestry. For example, even though every `Employee` object is a `SalariedEmployee` or an `HourlyEmployee` subclass object, it can be convenient to create an array of generic `Employee` objects. Likewise, an `Animal` array might contain individual elements that are `Dog`, `Cow`, or `Snake` objects. As long as every `Employee` subclass has access to a `calculatePay()` method, or every `Animal` subclass has access to a `speak()` method, you can manipulate an array of superclass objects by invoking the appropriate method for each subclass member.



In Chapter 8 you learned that all elements in a single array must be of the same type.

The statement `Animal[] ref = new Animal[3];` creates an array of three `Animal` references. The statement reserves enough computer memory for three `Animal` objects named `ref[0]`, `ref[1]`, and `ref[2]`. The statement does not actually instantiate `Animals`; `Animals` are abstract and cannot be instantiated. The statement simply reserves memory for three `Animal` object references. If you instantiate three `Animal` subclass objects, you can place references to those objects in the `Animal` array, as Figure 12-9 illustrates.



Recall from Chapter 8 that when you create an array of any type of objects, concrete or abstract, you are not actually constructing those objects. Instead, you are creating space for references not yet instantiated.

Once the objects are in the array, you can manipulate them like any other array objects. For example, you can use a loop and a subscript to get each individual reference to `speak()`. Figure 12-10 shows the output of the `AnimalArray` program. The output is identical to that of Figure 12-8, except that an array of references is used, instead of a single reference.

```
public class AnimalArray
{
    public static void main(String[] args)
    {
        Animal[] ref = new Animal[3];

        Cow aCow = new Cow("Mabel");

        Dog aDog = new Dog("Rover");

        Snake aSnake = new Snake("Siskal");

        ref[0] = aCow;

        ref[1] = aDog;

        ref[2] = aSnake;

        for (int x = 0; x < 3; ++x)
            ref[x].speak();

    }
}
```

**Figure 12-9** AnimalArray class

Next you will write an Event Handlers Incorporated program in which you create an array of Entertainment references. Within the program, you will assign MusicalEntertainment objects and OtherEntertainment objects to the same array. Then, because the different object types are stored in the same array, you can easily manipulate them by using a **for** loop.



```
Command Prompt
A:\Chapter.12>java AnimalArray
Moo
Woof
Sss
A:\Chapter.12>_
```

**Figure 12-10** Output of the AnimalArray program

To write a program that uses an **Entertainment** array:

1. Open a new file in your text editor, and then enter the following first few lines of the **EntertainmentDataBase** program:

```
public class EntertainmentDataBase
{
    public static void main(String[] args) throws Exception
    {
```

2. Create the following array of six **Entertainment** references and an integer subscript to use with the array:

```
Entertainment[] actArray = new Entertainment[6];
int x;
```

3. Enter the following **for** loop that prompts you to select whether you will enter a musical or non-musical entertainment act. Based on user input, instantiate either a **MusicalEntertainment** or an **OtherEntertainment** object.

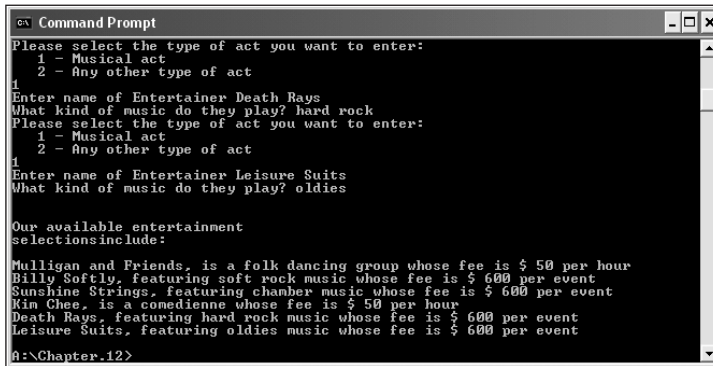
```
for(x = 0; x < actArray.length; ++x)
{
    char selection;
    System.out.print("Please select the type of ");
    System.out.println("act you want to enter:");
    System.out.println("  1 - Musical act");
    System.out.println("  2 - Any other type of act");
    selection = (char)System.in.read();
    System.in.read(); System.in.read();
    if(selection == '1')
        actArray[x] = new MusicalEntertainment();
    else
        actArray[x] = new OtherEntertainment();
}
```

4. After entering the information for all the acts, display the array contents by typing the following code, and by typing the closing curly braces for the **main()** method and for the class:

```
System.out.print("\n\nOur available entertainment ");
System.out.println("selections include:\n");
for(x = 0; x < actArray.length; ++x)
    System.out.println(actArray[x].toString());
}
}
```

5. Save the file as **EntertainmentDataBase.java** in the **Chapter.12** folder on your Student Disk, and then compile it. Run the program, enter some appropriate data (use Figure 12-11 as a guide, if you want), and then compare your results to the output shown in Figure 12-11.





```

C:\> Command Prompt
Please select the type of act you want to enter:
1 - Musical act
2 - Any other type of act
1
Enter name of Entertainer: Death Rays
What kind of music do they play? hard rock
Please select the type of act you want to enter:
1 - Musical act
2 - Any other type of act
1
Enter name of Entertainer: Leisure Suits
What kind of music do they play? oldies

Our available entertainment
selections include:
Mulligan and Friends, is a folk dancing group whose fee is $ 50 per hour
Billy Softly, featuring soft rock music whose fee is $ 600 per event
Sunshine Strings, featuring chamber music whose fee is $ 600 per event
Kin Chee, is a comedienne whose fee is $ 50 per hour
Death Rays, featuring hard rock music whose fee is $ 600 per event
Leisure Suits, featuring oldies music whose fee is $ 600 per event
A:\Chapter.12>

```

Figure 12-11 Final lines of output of the EntertainmentDataBase program

## USING THE OBJECT CLASS AND ITS METHODS

Every class in Java is actually a subclass, except one. When you define a class, if you do not explicitly extend another class, then your class is an extension of the `Object` class. The `Object` class is defined in the `java.lang` package, which is imported automatically every time you write a program. It includes methods that you can use or override, as you see fit.

### The `toString()` Method

You override the `Object` class `toString()` method in the steps you used to create the `MusicalEntertainment` and `OtherEntertainment` classes. If you do not create a `toString()` method for a class, then you can use the superclass version of the `toString()` method. For example, review the `Dog` class shown in Figure 12-3. Notice that it does not contain a `toString()` method and that it extends the `Animal` class. Examine the `Animal` class shown in Figure 12-2. Notice that it also does not define a `toString()` method. Yet, when you write the program that prints a `Dog` object, as shown in Figure 12-12, the program compiles correctly, converts the `Dog` object to a `String`, and produces the output shown in Figure 12-13. The output is not very useful, however. It consists of the class name of which the object is an instance, the at sign (`@`), and a hexadecimal (base 16) number that represents the object.



The hexadecimal number, which is expressed as a series of digits and letters, represents a computer memory address that can change every time you run the program, and basically is of no use to you.

It is usually better to write your own `toString()` method that displays some or all of the data field values, instead of using the automatic `toString()` method with your classes.



A good `toString()` method can be very useful in debugging a program. If you do not understand why a class is behaving as it is, you can print the `toString()` value and examine its contents. You can also use a `toString()` method to return text to a method statement call. In Chapter 10, you used the `toString()` method to return a `String` representation of the screen resolution and screen size using a combination of text, method calls, and catenation.

```
public class DogString
{
    public static void main(String[] args)
    {
        Dog myDog = new Dog("Murphy");

        System.out.println(myDog);
    }
}
```

Figure 12-12 DogString program

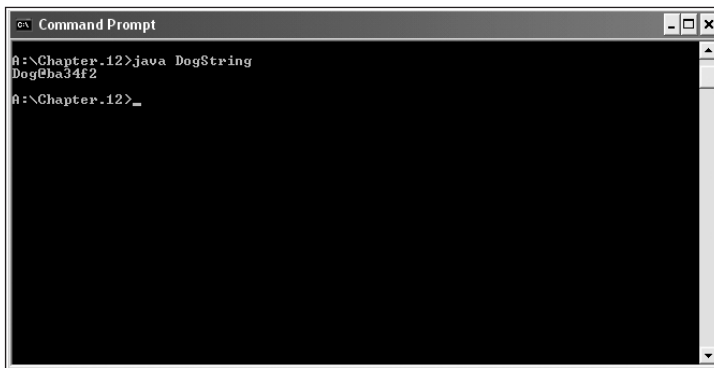


Figure 12-13 Output of the DogString program

## The equals() Method

The `Object` class also contains an `equals()` method that takes a single argument, which must be the same type as the type of the invoking method, as in the following example:

```
someObject.equals(someOtherObjectOfTheSameType)
```



Other classes, such as the `String` class, also have their own `equals()` methods. You first used the `equals()` method to compare `String` objects in Chapter 7. Two `String` objects were considered equal only if their `String` contents were identical.

The `Object` class `equals()` method returns a Boolean value indicating whether the objects are equal. This `equals()` method considers two objects of the same class to be equal only if they have the same memory address; in other words, they are equal only if one is a reference to the other. If you want to consider two objects to be equal only when one is a reference to the other, you can use the `Object` class `equals()` method. However, if you want to consider objects to be equal based on their contents, then you must write your own `equals()` method for your classes.

The program shown in Figure 12-14 instantiates three `Dog` objects: a `BlackLab` named Murphy, a `Collie` named Colleen, and a `Schnauzer` named Murphy. The `Dog` class does not include its own `equals()` method, so it does not override the `Object` `equals()` method. Thus, the program in Figure 12-14 produces the output in Figure 12-15. Even though two of the `Dog` objects have the same name, none of the `Dogs` are equal because they do not have the same memory address.

```
public class DogCompare
{
    public static void main(String[] args)
    {
        Dog aBlackLab = new Dog("Murphy");
        Dog aCollie = new Dog("Colleen");
        Dog aSchnauzer = new Dog("Murphy");

        System.out.print("The black lab and collie are ");

        if (aBlackLab.equals(aCollie))
            System.out.println("equal");
        else
            System.out.println("not equal");

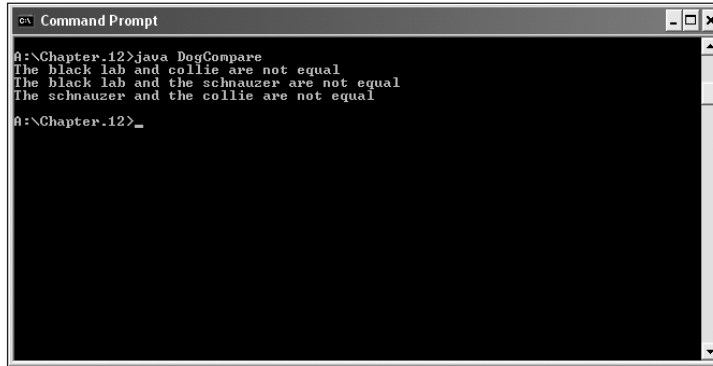
        System.out.print("The black lab and the schnauzer are ");

        if (aBlackLab.equals(aSchnauzer))
            System.out.println("equal");
        else
            System.out.println("not equal");

        System.out.print("The schnauzer and the collie are ");

        if (aSchnauzer.equals(aCollie))
            System.out.println("equal");
        else
            System.out.println("not equal");
    }
}
```

**Figure 12-14** DogCompare program



```
Command Prompt
A:\Chapter.12>java DogCompare
The black lab and collie are not equal
The black lab and the schnauzer are not equal
The schnauzer and the collie are not equal
A:\Chapter.12>_
```

**Figure 12-15** Output of the DogCompare program when Dog does not override equals()

Next you will add an equals() method to the Dog class to override the equals() method in the DogCompare program.

**To override the equals() method in the DogCompare program:**

1. Open the **Dog.java** program in your text editor, and change the class name to **Dog2**. Change the name of the Dog constructor to **Dog2**.
2. Add the equals() method shown in Figure 12-16 to the Dog2 class. You can add the equals() method to the Dog2 class file anywhere within the file as long as it is not within any other method. The best location is after the closing curly brace for the Dog2 class constructor and before the method header for speak(). This way, the equals() method appears in alphabetical order among the methods.
3. Save the file as **Dog2.java** in the Chapter.12 folder on your Student Disk, and then compile it.
4. Open the **DogCompare.java** program in your text editor, and change the class name to **DogCompare2**.
5. Change each occurrence of Dog that instantiates a new Dog object to **Dog2**.
6. Save the file as **DogCompare2.java** in the Chapter.12 folder on your Student Disk, and then compile it. Run the program and observe that the output (as shown in Figure 12-17) now indicates that two Dog objects with the same name are equal.

The equals() method in Figure 12-16 returns a Boolean value. When you call the method, you use the name of one Dog2 object, a dot, and the name of another Dog2 object as an argument within parentheses, as in `aBlackLab.equals(aCollie)`. Therefore, the equals() method header shows that it receives a Dog2 object, which has the local name `anotherDog`. When you compare the name of the calling Dog2 with the argument `anotherDog` by using the String equals() method, you determine whether the two Dog2s are to be considered equal.

```
boolean equals(Dog2 anotherDog)
{
    boolean result;

    if(getName().equals(anotherDog.getName()))

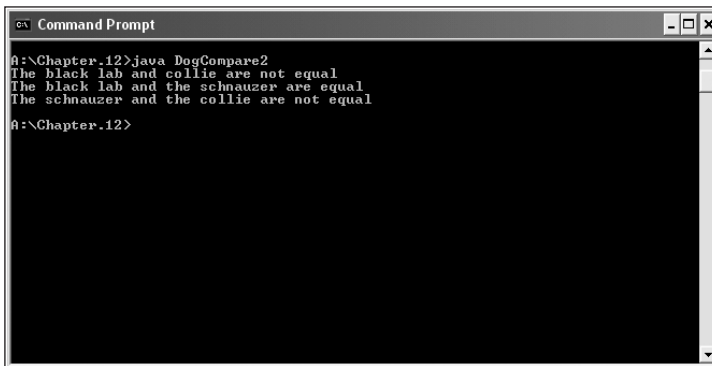
        result = true;

    else

        result = false;

    return result;
}
```

**Figure 12-16** Dog2 equals() method



**Figure 12-17** Output of the DogCompare2 program when Dog2 overrides equals()



Tip

Recall from Chapter 4 that when you use an instance method for a class object, the method receives a `this` reference to the calling object. Therefore, the call to the `getName()` method retrieves the name for `this` Dog. You could replace the expression `getName().equals(anotherDog.getName())` with `this.getName().equals(anotherDog.getName())`.



Tip

If you change a class, such as adding the `equals()` method to the Dog class, not only must you recompile the Dog class, but to make use of the newly added method, you must also recompile a client program such as DogCompare.

If there were more fields in the `Dog` class, you could base equality on a large number of comparisons. Rather than simply comparing names, within the `equals()` method of the `Dog` class you could substitute a more-detailed comparison, such as the following:

```
if(getName().equals(anotherDog.getName()) &&
    getAge() == anotherDog.getAge() &&
    getGender() == anotherDog.getGender() &&
    getBreed().equals(anotherDog.getBreed()))
    result = true;
else
    result = false;
```

Using this code, two `Dog` objects are considered equal only if they have the same name, age, gender, and breed.

Next you will add an `equals()` method to the Event Handlers Incorporated Entertainment class. Then you will use the `equals()` method in the `EntertainmentDataBase` program to compare each new Entertainment act to every act residing in the database. Your improved program will not allow two acts to have the same name.

#### To add an `equals()` method to the Entertainment class:

1. Open the **Entertainment.java** file in your text editor, and change the constructor and the class name to **Entertainment2**.
2. Position your insertion point after the closing curly brace of the `Entertainment` constructor, and then press **[Enter]** to start a new line.
3. Type the `equals()` method as follows:

```
public boolean equals(Entertainment2 act)
{
    boolean result;
    if(entertainer.equals(act.entertainer))
        result = true;
    else
        result = false;
    return result;
}
```

4. Save the file as **Entertainment2.java**, and then compile it using the **javac** command.

Next you will modify the `EntertainmentDataBase` program so the user cannot enter Entertainment objects with the same entertainer names.

#### To modify the EntertainmentDataBase class:

1. Open the **EntertainmentDataBase.java** file in your text editor, and then save it as **EntertainmentNoDuplicates.java**. Change the class name in the first line of the class file from `EntertainmentDataBase` to **EntertainmentNoDuplicates**.

2. Change the `Entertainment [] actArray = new Entertainment [6];` statement to `Entertainment2[] actArray = new Entertainment2[6];` because the `Entertainment` class used by the `EntertainmentDataBase` was updated in a previous example.
3. Change `actArray[x] = new MusicalEntertainment ();` and `actArray[x] = new OtherEntertainment ();` to `actArray[x] = new MusicalEntertainment2();` and `actArray[x] = new OtherEntertainment2();`, respectively, and then save the file.
4. Open the `MusicalEntertainment` file, and then save it as `MusicalEntertainment2.java`. Change the name of the class and the name of the constructor so each becomes `MusicalEntertainment2`. Change the class that this class extends to `Entertainment2`. Save the file, and then compile it.
5. Open the `OtherEntertainment` file, and then save it as `OtherEntertainment2.java`. Change the name of the class and the name of the constructor so each become `OtherEntertainment2`. Change the class that this class extends to `Entertainment2`. Save the file, and then compile it.



You must change the names of the `MusicalEntertainment` and `OtherEntertainment` files so that the new files `MusicalEntertainment2` and `OtherEntertainment2` now extend `Entertainment2`. When the `equals()` method in the `Entertainment2` class receives an object to compare, it must be of the same type as `Entertainment2`.

6. Open the `EntertainmentNoDuplicates` file if necessary, and position your insertion point at the end of the line that reads `actArray[x] = new OtherEntertainment2();`. Press **[Enter]** to start a new line, and then add the following `for` loop that compares the most recently entered `actArray` element against all previously entered `actArray` elements. If the new element equals any previously entered `Entertainment` act, then issue an error message and reduce the subscript by one. Reducing the subscript ensures that the next act you enter overwrites the duplicate act.

```
for(int y = 0; y < x; ++y)
    if(actArray[x].equals(actArray[y]))
    {
        System.out.println
            ("Sorry, you entered a duplicate act");
        --x;
    }
```

7. Save the file, compile it using the `javac` command, and then execute the program. When you see the prompts, enter any appropriate data. Make sure that you repeat an entertainer's name for several of the prompts. Each time you repeat a name, you will see an error message and get another opportunity to enter an act. The program will not end until you enter six acts with unique names.

---

## USING INHERITANCE TO ACHIEVE GOOD SOFTWARE DESIGN

When an automobile company designs a new car model, the company does not build every component of the new car from scratch. The company might design a new feature completely from scratch; for example, at some point someone designed the first air bag. However, many of a new car's features are simply modifications of existing features. The manufacturer might create a larger gas tank or more comfortable seats, but even these new features still possess many properties of their predecessors in the older models. Most features of new car models are not even modified; instead, preexisting components, such as air filters and windshield wipers, are included on the new model without any changes.

Similarly, you can create powerful computer programs more easily if many of their components are used either “as-is” or with slight modifications. Inheritance does not give you the ability to write any programs that you could not write without it because you could create every part of a program from scratch. Inheritance simply makes your job easier. Professional programmers constantly create new class libraries for use with Java programs. Having these classes available makes programming large systems more manageable.

You have already used many “as-is” classes, such as `String` and `JApplet`. In these cases, your programs were easier to write than if you had to write these classes yourself. Now that you have learned about inheritance, you have gained the ability to modify existing classes. When you create a useful, extendable superclass, you and other future programmers gain several advantages:

- Subclass creators save development time because much of the code needed for the class has already been written.
- Subclass creators save testing time because the superclass code has already been tested and probably used in a variety of situations. In other words, the superclass code is reliable.
- Programmers who create or use new subclasses already understand how the superclass works, so the time it takes to learn the new class features is reduced.
- When you create a new subclass in Java, neither the superclass source code nor the superclass bytecode is changed. The superclass maintains its integrity.

When you consider classes, you must think about the commonalities between them, and then you can create superclasses from which to inherit. You might be rewarded professionally when you see your own superclasses extended by others in the future.

---

## CREATING AND USING INTERFACES

Many object-oriented programming languages, such as C++, allow a subclass to inherit from more than one parent class. For example, you might create an `Employee` class that contains data fields pertaining to each employee in your organization. You might also create a `Product` class that holds information about each product your organization produces.



When you create a Patent class for each product for which your company holds a patent, you might want to include product information, as well as information about your company's employee who was responsible for the invention. It would be convenient to inherit fields and methods from both the Product and the Employee classes. The capability to inherit from more than one class is called **multiple inheritance**.

Multiple inheritance is a difficult concept, and when programmers use it, they encounter many problems. Programmers have to deal with the possibility that variables and methods in the parent classes may have identical names, which creates conflict when the child class uses one of the names. Also, you have already learned that a child class constructor must call its parent class constructor. When there are two or more parents, this task becomes more complicated. To which class should `super()` refer when a child class has multiple parents?

For all of these reasons, multiple inheritance is prohibited in the Java programming language. Java, however, does provide an alternative to multiple inheritance—an interface. An **interface** looks much like a class, except all of its methods must be abstract and all of its data (if any) must be **static final**. When you create a class that uses an interface, you include the keyword **implements** and the interface name in the class header. This notation requires class objects to include code for all the methods in the interfaces that have been implemented. Whereas **extends** exposes elements of the superclass program to the user of subclasses, **implements** exposes elements of your programs to the user without exposing the program source code.

As an example, you can create a Working interface to use with the Animal subclasses. For simplicity, give the Working interface a single method named `work()`. Figure 12-18 shows the Working program.

```
public interface Working
{
    public void work();
}
```

**Figure 12-18** Working interface

When any class implements Working, it must also include a `work()` method. The WorkingDog class in Figure 12-19 extends Dog and implements Working; its `work()` method calls the Dog `speak()` method, and then produces a line of output.

When you create a program that instantiates a WorkingDog object, as in Figure 12-20, you can use the `work()` method. The program output appears in Figure 12-21. You can also create WorkingCow and WorkingHorse classes that implement Working. In addition, if you decide to create a Playing interface, any class that implements Working can also implement Playing.

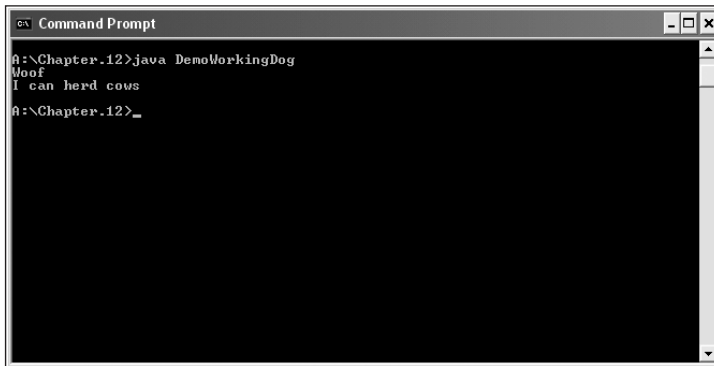
```
public class WorkingDog extends Dog implements Working
{
    public WorkingDog(String nm)
    {
        super(nm);
    }

    public void work()
    {
        speak();
        System.out.println("I can herd cows");
    }
}
```

**Figure 12-19** WorkingDog class

```
public class DemoWorkingDog
{
    public static void main(String[] args)
    {
        WorkingDog mySheltie = new WorkingDog("Simon");

        mySheltie.work();
    }
}
```

**Figure 12-20** DemoWorkingDog programA screenshot of a Windows Command Prompt window. The title bar reads "Command Prompt". The command prompt shows the command `h:\Chapter.12>java DemoWorkingDog` being executed. The output is `Woof` followed by `I can herd cows` on the next line. The prompt then shows `h:\Chapter.12>_` indicating the command has completed and the cursor is ready for the next input.

```
h:\Chapter.12>java DemoWorkingDog
Woof
I can herd cows
h:\Chapter.12>_
```

**Figure 12-21** Output of the DemoWorkingDog program

Abstract classes and interfaces are similar in that you cannot instantiate concrete objects from either one. Abstract classes differ from interfaces because abstract classes can contain nonabstract methods, but all methods within an interface must be abstract. A class can inherit from only one abstract superclass, but it can implement any number of interfaces.

Beginning programmers sometimes find it difficult to decide when to create an abstract superclass and when to create an interface. Remember, you create an abstract class when you want to provide data or methods that subclasses can inherit, but at the same time these subclasses maintain the ability to override the inherited methods.

Suppose you create a `CardGame` class to use as a base class for different card games. It contains four methods named `shuffle()`, `deal()`, `listRules()`, and `keepScore()`. The `shuffle()` method works the same way for every `CardGame`, so you write the statements for `shuffle()` within the superclass, and any `CardGame` objects you create later inherit `shuffle()`. The methods `deal()`, `displayRules()`, and `keepScore()` operate differently for every subclass, so you force `CardGame` children to contain instructions for those methods by leaving them empty in the superclass. When you write classes named `Hearts`, `Solitaire`, and `Poker`, you extend the `CardGame` parent class, inherit the `shuffle()` method, and implement `deal()`, `displayRules()`, and `keepScore()` methods for each specific child.

You create an interface when you know what actions you want to include, but you also want every user to separately define the behavior that must occur when the method executes. Suppose you create a `MusicalInstrument` class to use as a base for different musical instrument object classes such as `Piano`, `Violin`, and `Drum`. The parent `MusicalInstrument` class contains methods such as `playNote()` and `outputSound()` that apply to every instrument, but you want to implement these methods differently for each type of instrument. By making `MusicalInstrument` an interface, you require every subclass to code all the methods.



An interface specifies only the messages to which an object can respond; an abstract class can include methods that contain the actual behavior the object performs when those messages are received.

12

You also create an interface when you want a class to implement behavior from more than one parent. A `NameThatInstrument` card game that requires players to identify instrument sounds they hear by clicking cards, for example, could not extend from two classes, but it could extend from `CardGame` and implement `MusicalInstrument`.



You used prewritten interfaces earlier in this book. For example, you implemented the `ActionListener` interface in applets you wrote in Chapters 9 and 10. By implementing `ActionListener`, you provided your applet the means to respond to `ActionEvents`. You will use more interfaces in future chapters.

---

## CREATING AND USING PACKAGES

Throughout most of this book, you have imported packages into your programs. You learned in Chapter 4 that the `java.lang` package is automatically imported into every program you write. You have explicitly imported packages such as `java.util` and `javax.Swing`. When you create your own classes, you can place them in packages so that you or other

programmers can easily import related classes into new programs. When you create a number of classes that inherit from each other, you will often find it convenient to place these classes in a package.



Creating packages encourages others to reuse software because it makes it convenient to import many related classes at once.

When you create classes for others to use, you most often do not want to provide the users with your source code in the files with .java extensions. You expend significant effort developing workable code for your programs, and you do not want other programmers to be able to copy your programs, make minor changes, and market the new product themselves. Rather, you want to provide users with the compiled files with the .class extensions. These are the files the user needs to run the program you have developed. Likewise, when other programmers use the classes you have developed, they need only the completed compiled code to import into their programs. The .class files are the files you place in a package so other programmers can import them.

You can include a package statement at the beginning of your class file to place the compiled code into the indicated folder. For example, when it appears at the beginning of a class file, the statement **package com.course.animals;** indicates that the compiled file should be placed in a folder named com.course.animals. That is, the compiled file will be stored in the animals subfolder inside the course subfolder inside the com subfolder (or com\course\animals). The pathname can contain as many levels as you want. The package statement must appear outside the class definition.



The package statement, import statements, and comments are the only statements that appear outside class definitions in Java program files.

When you compile a file that you want to place in a package, you must use a compiler option with the **javac** command. The **-d** option indicates that you want to place the generated .class file in a folder. For example, the command **javac -d c:\Animal.java** indicates that the compiled Animal.java file should be placed in the root directory of drive C. If the Animal class file contains the statement **package com.course.animals;**, then the Animal.class file will be placed in C:\com\course\animals. If any of these subfolders do not exist, Java will create them. If you similarly package the compiled files for Dog.java, Cow.java, and so on, future programs only need to use the statement **import com.course.animals.\*** to be able to use all the related classes. Alternately, you can list each class separately, as in **import com.course.Dog;** and **import com.course.Cow;**. Usually, if you want to use only one or two classes in a package, you use separate import statements for each class. If you want to use many classes in a package, it is easier to import the entire package, even if there are some classes you will not use.



The *d* in the `-d` compiler option stands for directory, which is another name for folder.



You cannot import more than one package in one statement; for example, `import com.*` does not work.

Because the Java programming language is used extensively on the Internet, it is important to give every package a unique name. Sun Microsystems, the creator of the Java programming language, has defined a package-naming convention in which you use your Internet domain name in reverse order. For example, if your domain name is `course.com`, then you begin all of your package names with `com.course`. Subsequently, you organize your packages into reasonable subfolders. Using this convention ensures that your package names will not conflict with those of any other Java code providers.

Next you will place some of the Event Handlers Incorporated classes into a package. Because Event Handlers Incorporated sponsors a Web site at *eventhandlers.com*, you will use the `com.eventhandlers` package.

### To place three of your classes for Event Handlers Incorporated into a package:

1. Open the **Entertainment.java** file in your text editor.
2. For the first line in the file, insert the following statement:  
**`package com.eventhandlers.entertainment;`**
3. Save the file as **a:\Entertainment.java**. Notice that you are saving this file in the root directory of your disk, and not in the Chapter.12 folder. Because Java uses the dot (period) to separate folder names for packages, you cannot use a dot within a folder name.



Note that your drive letter may vary.

4. At the command line for the root directory on your Student disk, compile the file using the command **`javac -d a:\ Entertainment.java`**. (Make sure that you type a space between the drive name and the class name `Entertainment.java`.) Java will create a folder named `com\eventhandlers\entertainment` on your Student Disk. The compiled `Entertainment.class` file will be placed within this folder.



If you see a list of compile options when you try to compile the file, then you did not type a space between `a:\` and `Entertainment.java`. Repeat Step 4 to compile again.



To change to the command prompt for the root directory on your Student Disk, type `cd\` at the command prompt.

5. Examine the folders on your Student Disk, using any operating system program with which you are familiar. For example, if you are compiling at the DOS command line, type `dir a:\` at the command-line prompt to view the folders stored in the root directory. You can see that Java created a folder named `com`. Within the `com` folder is an `eventhandlers` folder, and within `eventhandlers` is an `entertainment` folder. The `Entertainment.class` file is within the `entertainment` subfolder, and not in the same folder as the `.java` source file where it ordinarily would be placed.



If Java did not create a `com` folder on your Student Disk, then you probably did not compile the file at the command prompt for the root directory. Repeat Steps 4 and 5, but make sure that you first change to the command prompt for the root directory.

6. Delete the copy of the **Entertainment.java** file from the root directory of your Student Disk. There is no further need for this source file because the compiled `.class` file is stored in the `com\eventhandlers\entertainment` folder.



If you don't want to delete the `Entertainment.java` file, you can move it to the `Chapter.12` folder and overwrite the existing file.

7. Open the **MusicalEntertainment.java** file in your text editor. For the first line in the file, insert the following statement: **`package com.eventhandlers.entertainment;`**
8. Save the file in the root directory as **`a:\MusicalEntertainment.java`**. At the command line for the root directory on your Student Disk, compile the file using the command **`javac -d a:\ MusicalEntertainment.java`**. (Make sure that you type a space between the drive name and the `MusicalEntertainment.java` filename.) Then delete the **MusicalEntertainment.java** source file from the root directory of your Student Disk.
9. Open the **OtherEntertainment.java** file in your text editor. For the first line in the file, insert the following statement: **`package com.eventhandlers.entertainment;`**. Save the file in the root directory of your Student Disk as **`a:\OtherEntertainment.java`**. At the command line for the root directory of your Student Disk, compile the file using the command: **`javac -d a:\ OtherEntertainment.java`**. Then delete the **OtherEntertainment.java** source file from the root directory.

10. Open the **EntertainmentDataBase.java** file in your text editor. For the first line in the file, insert the following statement: **import com.eventhandlers.entertainment.\*;** Save the file as **a:\EntertainmentDataBase.java**. Compile the file at the **A:\>** prompt using the **javac EntertainmentDataBase.java** command, and then run the program at the **A:\>** prompt using the **java EntertainmentDataBase** command. The program's output should be the same as it was before you added the import statement.



Instead of using the wildcard **import com.eventhandlers.entertainment.\*;**, you can use three separate import statements that name the classes individually, such as **import com.eventhandlers.entertainment.Entertainment;**

11. Examine again the contents of your Student Disk. The only .class file in the root directory of your Student Disk is the **EntertainmentDataBase.class** file. Because this file imports the class files from the **com.eventhandlers.entertainment** package, your program recognizes the **Entertainment**, **MusicalEntertainment**, and **OtherEntertainment** classes, even though neither their .java files nor their .class files are in the same folder with the **EntertainmentDataBase**.

Placing the Entertainment-related class files in a folder is not required for the **EntertainmentDataBase** program to execute correctly; you ran it in exactly the same manner before you learned about creating packages. The first time you executed the **EntertainmentDataBase**, all the files you used (source files as well as .class compiled files) were in the **Chapter.12** folder on your Student Disk. If you distribute that folder to clients, they have access to all the code you have written.

After placing the class files in a package, you could import the package into the **EntertainmentDataBase** program, and run the **EntertainmentDataBase** program from a separate folder. The folder with the three .class files is the only folder you would want to distribute to programmers who use your Entertainment classes to write programs similar to **EntertainmentDataBase**. Placing classes in packages gives you the ability to more easily isolate and distribute files.

## CHAPTER SUMMARY

- A class that you create only to extend from, but not to instantiate from, is an abstract class. An abstract class is one from which you cannot create any concrete objects, but from which you can inherit. You use the keyword **abstract** when you declare an abstract class.

- You cannot create instances of abstract classes using the **new** operator. Usually, abstract classes contain abstract methods. An abstract method is a method with no method statements. When you create an abstract method, you provide the keyword **abstract** and the intended method type, name, and arguments, but you do not provide any statements within the method. You must code a subclass method to override any inherited abstract superclass method.
- When you create a superclass and one or more subclasses, each object of the subclass “is a” superclass object. Because every subclass object “is a” superclass member, you can convert subclass objects to superclass objects.
- You can create a reference to a superclass. When you create a reference, you do not use the keyword **new** to create a concrete object. You create a variable name in which you can hold the memory address of a subclass concrete object that “is a” superclass member.
- The ability of a program to select the correct subclass method is known as dynamic method binding.
- You might want to create a superclass reference and treat subclass objects as superclass objects so you can create an array of different objects that share the same ancestry. You can manipulate an array of superclass objects by invoking the appropriate method for each subclass.
- When you create a useful, extendable superclass, you save development time because much of the code needed for the class has already been written. In addition, you save testing time and, because the superclass code is reliable, you reduce the time it takes to learn the new class features. You also maintain superclass integrity.
- An interface is similar to a class, but all of its methods must be abstract, and all of its data (if any) must be **static final**. When you create a class that uses an interface, you include the keyword **implements** and the interface name in the class header. This notation serves to require class objects to include code for all the methods in the interface.
- Abstract classes and interfaces are similar in that you cannot instantiate concrete objects from either. Abstract classes differ from interfaces because abstract classes can contain nonabstract methods, but all methods within an interface must be abstract. A class can inherit from only one abstract superclass, but it can implement any number of interfaces.
- You can place classes in packages so you or other programmers can easily import related classes into new programs. When you create a number of classes that inherit from each other, you often will find it convenient to place them in a package.
- You can include a package statement at the beginning of your class file to place compiled code in the indicated folder. The package statement must appear outside the class definition. When you compile a file that you want to place in a package, you must use the **-d** compiler option with the **javac** command.



- The convention for naming packages uses Internet domain names in reverse order to ensure that your package names will not conflict with those of any other Internet users.

---

## REVIEW QUESTIONS

1. Parent classes are \_\_\_\_\_ than their child classes.
  - a. smaller
  - b. more specific
  - c. easier to understand
  - d. more cryptic
2. Abstract classes differ from regular classes in that you \_\_\_\_\_.
  - a. must not code any methods within them
  - b. must instantiate objects from them
  - c. cannot instantiate objects from them
  - d. cannot have data fields within them
3. Abstract classes can contain \_\_\_\_\_.
  - a. abstract methods
  - b. nonabstract methods
  - c. both of the above
  - d. none of the above
4. An abstract class `Product` has two subclasses, `Perishable` and `NonPerishable`. None of the constructors for these classes requires any arguments. Which of the following statements is legal?
  - a. `Product myProduct = new Product();`
  - b. `Perishable myProduct = new Product();`
  - c. `NonPerishable myProduct = new NonPerishable();`
  - d. none of the above
5. An abstract class `Employee` has two subclasses, `Permanent` and `Temporary`. The `Employee` class contains an abstract method named `setType()`. Before you can instantiate `Permanent` or `Temporary` objects, which of the following statements must be true?
  - a. You must code statements for the `setType()` method within the `Permanent` class.
  - b. You must code statements for the `setType()` method within both the `Permanent` and the `Temporary` classes.

- c. You must not code statements for the `setType()` method within either the Permanent or Temporary classes.
  - d. You may code statements for the `setType()` method within the Permanent class or the Temporary class, but not both.
6. When you create a superclass and one or more subclasses, each object of the subclass \_\_\_\_\_ superclass object.
- a. overrides the
  - b. “is a”
  - c. “is not a”
  - d. is a new
7. Which of the following statements are false?
- a. Subclass objects are members of their superclass.
  - b. Superclass objects can contain abstract methods.
  - c. You can convert subclass objects to superclass objects.
  - d. Two of the above statements are false.
8. When you create a \_\_\_\_\_, you create a variable name in which you can hold the memory address of an object.
- a. class
  - b. superclass
  - c. subclass
  - d. reference
9. The program’s ability to select the correct subclass method to execute is known as \_\_\_\_\_ method binding.
- a. polymorphic
  - b. dynamic
  - c. early
  - d. intelligent
10. The statement \_\_\_\_\_ creates an array of five reference objects of an abstract class named `Currency`.
- a. `Currency[] = new Currency[5];`
  - b. `Currency[] currencyref= new Currency[5];`
  - c. `Currency[5] currencyref = new Currency[5];`
  - d. `Currency[5] = new Currency[5];`

11. You \_\_\_\_\_ override the `toString()` method in any class you create.
- a. cannot
  - b. can
  - c. must
  - d. must implement `StringListener` to
12. The `Object` class `equals()` method takes \_\_\_\_\_.
- a. no arguments
  - b. one argument
  - c. two arguments
  - d. as many arguments as you need
13. The following statement appears in a Java program:  
`if(thing.equals(anotherThing)) x = 1;` You know that \_\_\_\_\_.
- a. `thing` is an object of the `Object` class
  - b. `anotherThing` is the same type as `thing`
  - c. both of the above are correct
  - d. none of the above are correct
14. The `Object` class `equals()` method considers two objects of the same class to be equal if they have the same \_\_\_\_\_.
- a. value in all data fields
  - b. value in any data field
  - c. data type
  - d. memory address
15. Java subclasses have the ability to inherit from \_\_\_\_\_ parent class(es).
- a. one
  - b. two
  - c. multiple
  - d. no
16. The alternative to multiple inheritance in Java is known as a(n) \_\_\_\_\_.
- a. superobject
  - b. abstract class
  - c. interface
  - d. none of the above

17. When you create a class that uses an interface, you include the keyword \_\_\_\_\_ and the interface's name in the class header.
- a. `interface`
  - b. `implements`
  - c. `accouterments`
  - d. `listener`
18. You cannot instantiate concrete objects from a(n) \_\_\_\_\_.
- a. abstract class
  - b. interface
  - c. either a or b
  - d. neither a nor b
19. In Java, a class can \_\_\_\_\_.
- a. inherit from only one abstract superclass
  - b. implement only one interface
  - c. both a and b
  - d. neither a nor b
20. When you want to provide some data or class that subclasses can inherit, but you want the subclasses to override some specific methods, you should write a(n) \_\_\_\_\_.
- a. abstract class
  - b. interface
  - c. final superclass
  - d. concrete object

---

## EXERCISES

1. a. Create an abstract class named `Book`. Include a `String` field for the book's title and a `double` field for the book's price. Within the class, include a constructor that requires the book title and two get methods—one that returns the title and one that returns the price. Also include an abstract method named `setPrice()`. Create two child classes of `Book`: `Fiction` and `NonFiction`. Within the constructors for the `Fiction` and `NonFiction` classes, call `setPrice` so all `Fiction` Books cost \$24.99 and all `NonFiction` Books cost \$37.99. Finally, write a program that demonstrates that you can create both a `Fiction` and a `NonFiction` Book and display their fields. Save the **`Book.java`**, **`Fiction.java`**, **`NonFiction.java`**, and **`UseBook.java`** programs in the `Chapter.12` folder on your Student Disk.

- b. Write a program named **BookArray** in which you create an array that holds 10 Books, some Fiction and some NonFiction. Using a **for** loop, display details about all 10 books. Save the **BookArray.java** program in the Chapter.12 folder on your Student Disk.
2. a. Create an abstract class named **Account** for a bank. Include an integer field for the account number and a double field for the account balance. Also include a constructor that requires an account number and sets the balance to 0.0. Include a set method for the balance. Also include two abstract get methods—one for each field. Create two child classes of **Account**: **Checking** and **Savings**. Within the **Checking** class, the get method displays the String “Checking Account Information”, the account number, and the balance. Within the **Savings** class, add a field to hold the interest rate, and require the **Savings** constructor to accept an argument for the value of the interest rate. The **Savings** get method displays the String “Savings Account Information”, the account number, the balance, and the interest rate. Save the **Account.java**, **Checking.java**, and **Savings.java** programs in the Chapter.12 folder on your Student Disk.
- b. Write a program named **AccountArray** in which you enter data for a mix of 10 **Checking** and **Savings** accounts. Use a **for** loop to display the data. Save the **AccountArray.java** program in the Chapter.12 folder on your Student Disk.
3. Create an abstract **Auto** class with fields for the car make and price. Include get and set methods for these fields; the **setPrice()** method is abstract. Create two subclasses for individual automobile makers (for example, **Ford** or **Chevy**) and include appropriate **setPrice()** methods in each subclass. Finally, write a program that uses the **Auto** class and subclasses to display information about different cars. Save the **Auto.java**, **Ford.java**, **Chevy.java**, and **UseAuto.java** programs in the Chapter.12 folder on your Student Disk.
4. Create an abstract class **Division** with fields for a company’s division name and account number, and corresponding get and set methods. Use a constructor in the superclass. Create at least two subclasses for divisions such as **Accounting** or **Human Resources**. Write a program that uses the classes and displays information about them. Save the **Division.java**, **HumanResources.java**, **Accounting.java**, and **UseDivision.java** programs in the Chapter.12 folder on your Student Disk.
5. Write a program named **UseChildren** that uses an abstract **Child** class, and **Male** and **Female** subclasses, to display the name, gender, and age of two or more children. Use constructors with appropriate arguments in each of the classes. Include get and set methods, at least one of which is abstract. Save the **Child.java**, **Male.java**, **Female.java**, and **UseChildren.java** programs in the Chapter.12 folder on your Student Disk.
6. Create a class named **NewsPaperSubscriber** with fields for a subscriber’s street address and the subscription rate. Include get and set methods for the subscriber’s street address, and get and set methods for the subscription rate. The set method for the rate is abstract. Include an **equals()** method that indicates two **Subscribers** are equal if they have the same street address. Create child classes named **SevenDaySubscriber**, **WeekdaySubscriber**, and **WeekendSubscriber**. Each child class constructor sets the rate

as follows: SevenDaySubscribers pay \$4.50 per week, WeekdaySubscribers pay \$3.50 per week, and WeekendSubscribers pay \$2.00 per week. Each child class should include a `toString()` method that returns the street address, rate, and service type. Write a program named `Subscribers` that prompts the user for the subscriber's street address and requested service, and creates the appropriate object based on the service type. Do not let the user enter more than one subscription type for any given street address. Save the **NewspaperSubscriber.java**, **WeekdaySubscriber.java**, **WeekEndSubscriber.java**, and **Subscriber.java** programs in the Chapter.12 folder on your Student Disk.

7. a. Create an interface named `Turning`, with a single method named `turn()`. Create a class named `Leaf` that implements `turn()` to print "Changing colors". Create a class named `Page` that implements `turn()` to print "Going to the next page". Create a class named `Pancake` that implements `turn()` to print "Flipping". Write a program named `Turners` that creates one object of each of these class types and demonstrates the `turn()` method for each class. Save the **Turning.java**, **Leaf.java**, **Pager.java**, **Pancake.java**, and **Turners.java** programs in the Chapter.12 folder on your Student Disk.
- b. Think of two more objects that use `turn()`, create classes for them, and then add objects to the `Turners` program. Save the programs, using the names of new objects that use `turn()`, in the Chapter.12 folder on your Student Disk.
8. Write a program that uses an abstract class named `Drug`, and subclasses for two specific drugs to display a drug, its purpose, and the number of times per day it should be taken. Use constructors in each class, with appropriate arguments. Include get and set methods, at least one of which is abstract. Prompt the user for the drug to be displayed, and then create the appropriate object. Save the programs, using the name of each drug for the program name, in the Chapter.12 folder on your Student Disk.
9. Write a program named `UseInsurance` that uses an abstract `Insurance` class and `Health` and `Life` subclasses to display different types of insurance policies and the cost per month. Use constructors in each class, with appropriate arguments. Include get and set methods, at least one of which is abstract. Prompt the user for the type to be displayed, and then create the appropriate object. Also create an interface for a `print()` method and use this interface with both subclasses. Save the **Life.java**, **Health.java**, **Insurance.java**, **Print.java**, and **UseInsurance.java** programs in the Chapter.12 folder on your Student Disk.
10. Write a program named `UseLoan` that uses an abstract class named `Loans` and subclasses to display different types of loans and the cost per month (home, car, and so on). Use constructors in each of the classes with appropriate arguments. Include get and set methods, at least one of which is abstract. Prompt the user for the type to be displayed, and then create the appropriate object. Also create an interface with at least one method that you use with your subclasses. Save the **Loans.java**, **Car.java**, **Home.java**, **Print.java**, and **UseLoan.java** programs in the Chapter.12 folder on your Student Disk.

11. Create an abstract class called `GeometricFigure`. Each figure includes a height, a width, a figure type, and an area. Include an abstract method to determine the area of the figure. Create two subclasses called `Square` and `Triangle`. Create a program that demonstrates the use of both subclasses, and create them using an array. Save the **`GeometricFigure.java`**, **`Square.java`**, **`Triangle.java`**, and **`UseGeometric.java`** programs in the Chapter.12 folder on your Student Disk.
12. Create an interface called `Playing`. The interface has an abstract method called `play()`. Create classes called `Child`, `Musician` and `Actor` that all implement `Playing`. Create a program that demonstrates the use of the classes. Save the **`Playing.java`**, **`Child.java`**, **`Actor.java`**, **`Musician.java`**, and **`UsePlaying.java`** programs in the Chapter.12 folder on your Student Disk.
13. Create an abstract class called `Student`. The student class includes a name and a Boolean value representing full-time status. Include an abstract method to determine the tuition, with full-time students paying \$2000, and part-time students paying \$200 per credit hour. Create two subclasses called `FullTime` and `PartTime`. Create a program which demonstrates the use of both subclasses. Save the **`Student.java`**, **`Fulltime.java`**, **`PartTime.java`**, and **`UseStudent.java`** programs in the Chapter.12 folder on your Student Disk.
14. Modify Exercise 11 with the abstract class called `GeometricFigure`. Add an interface called `Sides` with a method `printSides()`. Modify the subclasses to include the use of the interface to print the number of sides of the figure. Create a program which demonstrates the use of both subclasses. Save the **`GeometricFigure2.java`**, **`Square2.java`**, **`Triangle2.java`**, **`Sides.java`**, and **`UseGeometric2.java`** programs in the Chapter.12 folder on your Student Disk.
15. Create a package named `com.course.animals` using the `Animal`, `Dog`, `Cow`, and `Snake` classes. Demonstrate that your package produces identical results with the output of the `AnimalArray` class. Save the necessary files in the Chapter.12 folder on your Student Disk.
16. Each of the following files in the Chapter.12 folder on your Student Disk has syntax and/or logical errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with `Fix`. For example, `DebugTwelve1.java` will become `FixDebugTwelve1.java`.
  - a. `DebugTwelve1.java`
  - b. `DebugTwelve2.java`
  - c. `DebugTwelve3.java`
  - d. `DebugTwelve4.java`

## CASE PROJECT



Sanchez Construction Loan Co. makes small loans for construction projects up to a maximum of \$10,000.00. The current cost for a \$10,000 loan is based on the following fee structure that has a maximum loan payoff time of 24 months:

Time	Fee
6 months	\$800.00
12 months	\$1,800.00
18 months	\$3,000.00
24 months (max)	\$4,000.00

You have been asked to write a program that will track a starting and ending date (due date) for all new construction loans. The program must also calculate the amount of the original loan and the total amount owed at the due date (original loan amount + loan fee). The program should include four classes, as shown in the table below:

Class	Type
Loan	public class
LoanInterface	public interface
AnnualLoan	public class extends Loan implements LoanInterface
DemoLoan	Test program

The Loan Interface requires a CalculateFee() method. This method must be implemented in the AnnualLoan class. The DemoLoan test program should instantiate at least two AnnualLoan objects that output the loan amount of \$10,000, the loan fee for the period, the beginning date of the loan, the ending date of the loan, and the balance due at the loan due date. Save the programs as **Loan.java**, **LoanInterface.java**, **AnnualLoan.java**, and **DemoLoan.java** in the Chapter.12 folder on your Student Disk.