Chapter 11

More Object-Oriented Programming Concepts

Chapter Introduction

In this chapter, you will learn about:

- Constructors
- Destructors
- Composition
- Inheritance
- GUI objects
- Exception handling
- The advantages of object-oriented programming

11-1 Understanding Constructors

In Chapter 10, you learned that you can create classes to encapsulate data and methods, and that you can instantiate objects from the classes you define. For example, you can create an `Employee` class that contains fields such as `lastName, hourlyWage`, and `weeklyPay`, and methods that set and return values for those fields. When you instantiate an object with a statement that uses the class type and an object identifier, such as `Employee chauffeur`, you are actually calling a method named `Employee()`. A method that has the same name as a class and that establishes an object is a constructor method, or more simply, a **constructor** (An automatically called method that establishes an object.) . A **default constructor** (A constructor that requires no arguments.) is one that requires no arguments. If a constructor requires arguments, it is a **nondefault constructor** (A constructor that requires at least one argument.) constructor. In object-oriented programming (OOP) languages, you can write both default and nondefault constructors, but if you do not create either, then a default constructor is created automatically by the compiler for every class you write.

The constructor for the `Employee` class establishes one `Employee` object. Depending on the programming language, the automatically supplied default constructor might provide initial values for the object's data fields; for example, in many languages all numeric fields are set to zero by default. If you do not want an object's fields to hold default values, or if you want to perform additional tasks when you create an instance of a class, you can write your own constructor. Any constructor you write must have the same name as the class it constructs, and cannot have a return type. Normally, you declare constructors to be public so that other classes can instantiate objects that belong to the class. You can create a constructor to accept one or more parameters; when you do not include parameters for a constructor you write, your constructor becomes the default constructor for the class and the automatically supplied version is no longer usable. In other words, a class can have three types of constructors:

- A class can contain a default (parameterless) constructor that is created automatically.
- A class can contain a default (parameterless) constructor that you create explicitly. A class with an explicitly created default constructor no longer contains the automatically supplied version, but it can coexist with a nondefault constructor.
- A class can contain a nondefault constructor (with one or more parameters), which must be explicitly created. A class with a nondefault constructor no longer contains the automatically supplied default version, but it can coexist with an explicitly created default constructor.

11-1a Default Constructors

For example, if you want every `Employee` object to have a starting hourly wage of $10.00 as well as the correct weekly pay for that wage, then you could write the default constructor for the `Employee` class that appears in Figure 11-1. Any `Employee` object instantiated will have an `hourlyWage` field equal to 10.00 and a `weeklyPay` field equal to 400.00. The `lastName` field will hold the default value for strings in the programming language in which this class is implemented because `lastName` is not assigned in the constructor.

Figure 11-1

`Employee` Class with a Default Constructor that Sets `hourlyWage` and `weeklyPay`

```
class Employee
   Declarations
      private string lastName
      private num hourlyWage
      private num weeklyPay

   public Employee()
      hourlyWage = 10.00
      calculateWeeklyPay()
   return

   public void setLastName(string name)
      lastName = name
   return

   public void setHourlyWage(num wage)
      hourlyWage = wage
      calculateWeeklyPay()
   return

   public string getLastName()
   return lastName

   public num getHourlyWage()
   return hourlyWage

   public num getWeeklyPay()
   return weeklyPay

   private void calculateWeeklyPay()
      Declarations
         num WORK_WEEK_HOURS = 40
      weeklyPay = hourlyWage * WORK_WEEK_HOURS
   return
endClass
```

The `Employee` constructor in Figure 11-1 calls the `calculateWeeklyPay()` method. You can write any statement you want in a constructor; it is just a method. Although you usually have no reason to do so, you could output a message from a constructor, accept input, declare local variables, or perform any other task. You can place a constructor anywhere inside the class, outside of any other method. Often, programmers list constructors first among the methods, because a constructor is the first method used when an object is created.

Figure 11-2 shows a program in which two `Employee` objects are declared and their `hourlyWage` values are displayed. In the output in Figure 11-3, you can see that even though the `setHourlyWage()` method is never called directly in the program, the `Employees` possess valid hourly wages as set by their constructors.

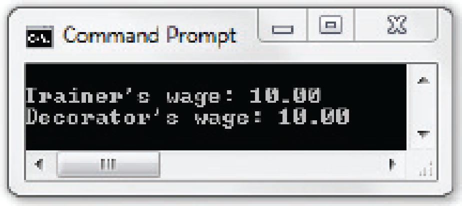
Figure 11-2

Program that Declares `Employee` Objects Using Class in Figure 11-1

```
start
   Declarations
      Employee myPersonalTrainer
      Employee myInteriorDecorator
   output "Trainer's wage: ",
      myPersonalTrainer.getHourlyWage()
   output "Decorator's wage: ",
      myInteriorDecorator.getHourlyWage()
stop
```

Figure 11-3

Output of Program in Figure 11-2



```
Trainer's wage: 10.00
Decorator's wage: 10.00
```

The `Employee` class in Figure 11-1 sets an `Employee`'s hourly wage to 10.00 at construction, but the class also contains a `setHourlyWage()` method that an application could use later to change the initial `hourlyWage` value. A potentially superior way to write the `Employee` class constructor is shown in Figure 11-4. In this version of the constructor, a named constant with the value 10.00 is passed to `setHourlyWage()`. Using this technique provides several advantages:

- The statement to call `calculateWeeklyPay()` is no longer required in the constructor because the constructor calls `setHourlyWage()`, which calls `calculateWeeklyPay()`.
- In the future, if restrictions should be imposed on `hourlyWage`, the code will need to be altered in only one location. For example, if `setHourlyWage()` is modified to disallow rates that are too high and too low, the code will change only in the `setHourlyWage()` method and will not have to be modified in the constructor. This reduces the amount of work required and reduces the possibility for error.

Figure 11-4

Alternate and Efficient Version of the `Employee` Class Constructor

```
public Employee()
   Declarations
      num DEFAULT_WAGE = 10.00
   setHourlyWage(DEFAULT_WAGE)
return
```

Of course, if different `hourlyWage` requirements are needed at initialization than are required when the value is set after construction, then different code statements will be written in the constructor than those written in the `setHourlyWage()` method.

11-1b Nondefault Constructors

You can write a nondefault constructor, which is one that accepts parameters. When you create a constructor, whether it is default or nondefault, the automatically supplied default constructor is no longer accessible.

For example, instead of forcing every `Employee` to be constructed with the same initial values, you might choose to create `Employee` objects that each have a unique `hourlyWage` by passing a numeric value for the wage to the constructor. Figure 11-5 shows an `Employee` constructor that receives an argument. With this constructor, an argument is passed using a declaration similar to one of the following:

```
Employee partTimeWorker(8.81)
Employee partTimeWorker(valueEnteredByUser)
```

When the constructor executes, the numeric value within the constructor call is passed to `Employee()`, where the parameter `rate` takes on the value of the argument. The value is assigned to `hourlyWage` within the constructor.

Figure 11-5

`Employee` Constructor that Accepts a Parameter

```
public Employee(num rate)
   hourlyWage = rate
   calculateWeeklyPay()
return
```

If you create an `Employee` class with a constructor such as the one shown in Figure 11-5, and it is the only constructor in the class, then every `Employee` object you create must use a numeric argument in its declaration. In other words, with this new version of the class that contains a single nondefault constructor, the following declaration no longer works:

```
Employee partTimeWorker
```

11-1c Overloading Methods and Constructors

In Chapter 9, you learned that you can overload methods by writing multiple versions of a method with the same name but different argument lists. In the same way, you can overload instance methods and constructors. For example, Figure 11-6 shows a version of the `Employee` class that contains two constructors. Recall that a method's signature is its name and list of argument types. The constructors in Figure 11-6 have different signatures —one version requires no argument and the other requires a numeric argument. In other words, this version of the class contains both a default constructor and a nondefault constructor.

Figure 11-6

`Employee` Class with Overloaded Constructors

When you use the version of the class shown in Figure 11-6, then you can make statements like the following:

```
Employee deliveryPerson
Employee myButler(25.85)
```

When you declare an `Employee` using the first statement, an `hourlyWage` of 10.00 is automatically set because the statement uses the parameterless version of the constructor. When you declare an `Employee` using the second statement, `hourlyWage` is set to the passed value. Any method or constructor in a class can be overloaded, and you can provide as many versions as you want as long as each version has a unique signature. For example, you could add a third constructor to the `Employee` class, as shown in Figure 11-7. This version can coexist with the other two because the parameter list is different from either existing version. With this version you can specify the hourly rate for the `Employee` as well as a name. If an application makes a statement similar to the following, then this two-parameter version would execute:

```
Employee myMaid(22.50, "Parker")
```

Figure 11-7

A Third Possible `Employee` Class Constructor

```
public Employee(num rate, string name)
   lastName = name
   hourlyWage = rate
   calculateWeeklyPay()
return
```

You might create an `Employee` class with several constructor versions to provide flexibility for client programs. For example, a particular client program might use only one version, and a different client might use another.

Watch the video *Constructors*.

Two Truths & A Lie

Understanding Constructors

1. A constructor is a method that establishes an object.

   T F

2. A default constructor is defined as one that is created automatically.

   T F

3. Depending on the programming language, a default constructor might provide initial values for the object's data fields.

   T F

11-2 Understanding Destructors

A **destructor** (A is an automatically called method that contains the actions you require when an instance of a class is destroyed.) contains the actions you require when an instance of a class is destroyed. Most often, an instance of a class is destroyed when the object goes out of scope. As with constructors, if you do not explicitly create a destructor for a class, one is provided automatically.

The most common way to declare a destructor explicitly is to use an identifier that consists of a tilde (~) followed by the class name. You cannot provide parameters to a destructor; it must have an empty parameter list. As a consequence, destructors cannot be overloaded; a class can have one destructor at most. Like a constructor, a destructor has no return type.

The rules for creating and naming destructors vary among programming languages. For example, in Visual Basic classes, the destructor is called `Finalize`.

Figure 11-8 shows an `Employee` class that contains only one field (`idNumber`), a constructor, and a shaded destructor. Although it is unusual for a constructor or destructor to output anything, these display messages so you can see when the objects are created and destroyed. When you execute the program in Figure 11-9, you instantiate two `Employee` objects, each with its own `idNumber` value. When the program ends, the two `Employee` objects go out of scope, and the destructor for each object is called automatically. Figure 11-10 shows the output.

Figure 11-8

`Employee` Class with Destructor

```
class Employee
   Declarations
      private string idNumber
   public Employee(string empID)
      idNumber = empId
      output "Employee ", idNumber, " is created"
   return
   public ~Employee()
      output "Employee ", idNumber, " is destroyed"
   return
endClass
```

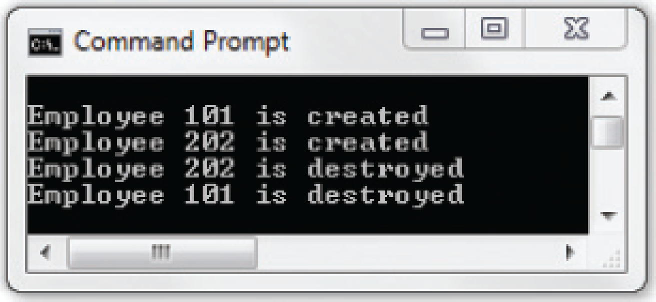
Figure 11-9

Program that Declares Two `Employee` Objects

```
start
   Declarations
      Employee aWorker("101")
      Employee anotherWorker("202")
stop
```

Figure 11-10

Output of Program in Figure 11-9



```
Employee 101 is created
Employee 202 is created
Employee 202 is destroyed
Employee 101 is destroyed
```

The program in Figure 11-9 never explicitly calls the `Employee` class destructor, yet you can see from the output that the destructor executes twice. Destructors are invoked automatically; you usually do not explicitly call one, although in some languages you can. Interestingly, you can see from the output in Figure 11-10 that the last object created is the first object destroyed; the same relationship would hold true no matter how many objects the program instantiated if the objects went out of scope at the same time.

An instance of a class becomes eligible for destruction when it is no longer possible for any code to use it—that is, when it goes out of scope. In many languages, the actual execution of an object's destructor might occur at any time after the object becomes eligible for destruction.

For now, you have little reason to create a destructor except to demonstrate how it is called automatically. Later, when you write more sophisticated programs that work with files, databases, or large quantities of computer memory, you might want to perform specific cleanup or close-down tasks when an object goes out of scope. Then you will place appropriate instructions within a destructor.

Two Truths & A Lie

Understanding Destructors

1. Unlike constructors, you must explicitly create a destructor if you want one for a class.

   T F

2. You cannot provide parameters to a destructor; it must have an empty argument list.

   T F

3. Destructors cannot be overloaded; a class can have one destructor at most.

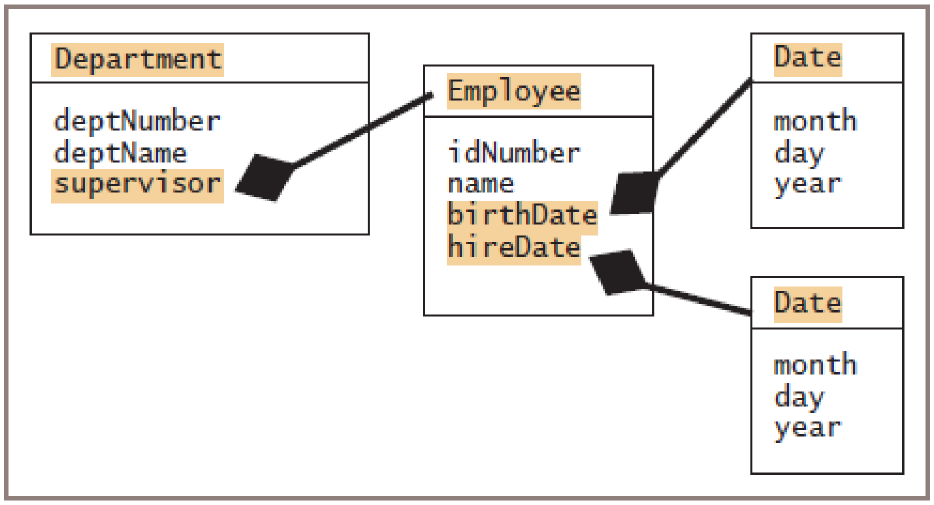
   T F

11-3 Understanding Composition

A class can contain objects of another class as data fields. For example, you might create a class named `Date` that contains a month, day, and year, and add two `Date` fields to an `Employee` class to hold the `Employee`'s birth date and hire date. Then you might create a class named `Department` that represents every department in a company, and create the `Department` class to contain a supervisor, who is an `Employee`. Figure 11-11 contains a diagram of these relationships. When a class contains objects of another class, the relationship is called a **whole-part relationship** (An association in which an object of one class is part of an object of a larger whole class.) or **composition** (The technique of placing an object within an object of another class.) . The relationship created is also called a **has-a relationship** (A whole-part relationship; the type of relationship that exists when using composition.) because one class “has an” instance of another.

Figure 11-11

Diagram of Typical Composition Relationships



Placing one or more objects within another object is often known as *composition* when the parts cease to exist if the whole ceases to exist, and *aggregation* when the parts can exist without the whole. For example, the relationship of a `Car` to its `Motor` might be called *composition*, but the relationship of a `UsedCarLot` to its `Cars` might be called *aggregation*. These terms are defined more precisely in Chapter 13.

When your classes contain objects that are members of other classes, your programming task becomes increasingly complex. For example, you sometimes must refer to a method by a very long name. Suppose you create a `Department` class that contains an array of `Employee` objects (those who work in the department), and a method named `getHighestPaidEmployee()` that returns a single `Employee` object. The `Employee` class contains a method named `getHireDate()` that returns a `Date` object—an `Employee`’s hire date. Further suppose the `Date` class contains a method that returns the year portion of the `Date`, and that you create a `Department` object named `sales`. An application might contain a statement such as the following, which outputs the year that the highest-paid employee in the sales department was hired:

```
output sales.getHighestPaidEmployee().getHireDate().getYear()
```

Additionally, when classes contain objects that are members of other classes, all the corresponding constructors and destructors execute in a specific order. As you work with object-oriented programming languages, you will learn to manage these complex issues.

Two Truths & A Lie

Understanding Composition

1. A class can contain objects of another class as data fields.

   T F

2. Composition occurs when you use an object as a field within another class.

   T F

3. Composition is called an is-a relationship because one class “is an” instance of another.

   T F

11-4 Understanding Inheritance

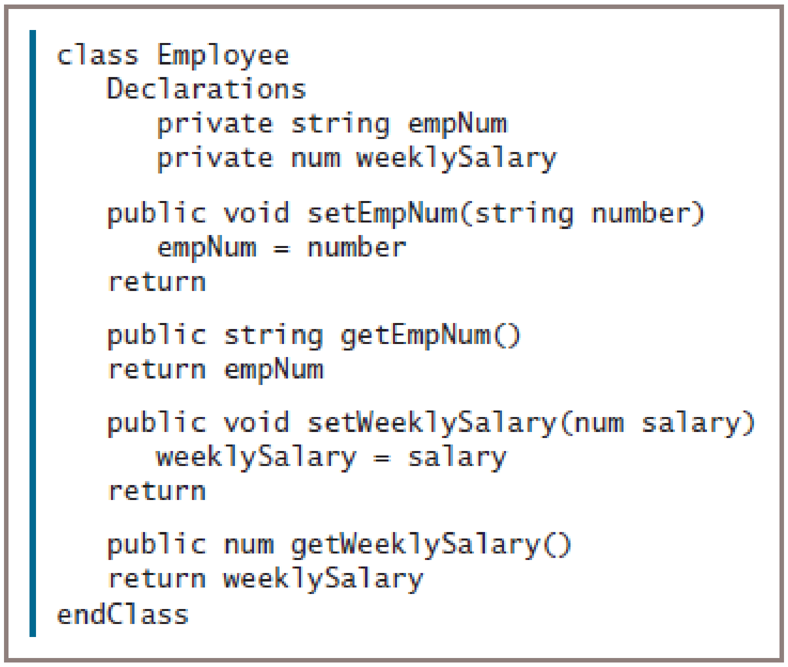
Understanding classes helps you organize objects in real life. Understanding inheritance helps you organize them more precisely. Inheritance enables you to apply your knowledge of a general category to more specific objects. When you use the term *inheritance*, you might think of genetic inheritance. You know from biology that your blood type and eye color are the products of inherited genes. You might choose to own plants and animals based on their inherited attributes. You plant impatiens next to your house because they thrive in the shade; you adopt a poodle because you know poodles don’t shed. Every plant and pet has slightly different characteristics, but within a species, you can count on many consistent inherited attributes and behaviors. In other words, you can reuse the knowledge you gain about general categories and apply it to more specific categories.

Similarly, the classes you create in object-oriented programming languages can inherit data and methods from existing classes. When you create a class by making it inherit from another class, the new class contains fields and methods automatically, allowing you to reuse fields and methods that are already written and tested.

You already know how to create classes and how to instantiate objects that are members of those classes. For example, consider the `Employee` class in Figure 11-12. The class contains two data fields, `empNum` and `weeklySalary`, as well as methods that get and set each field.

Figure 11-12

An Employee Class



```
class Employee
   Declarations
      private string empNum
      private num weeklySalary

   public void setEmpNum(string number)
      empNum = number
   return

   public string getEmpNum()
   return empNum

   public void setWeeklySalary(num salary)
      weeklySalary = salary
   return

   public num getWeeklySalary()
   return weeklySalary
endClass
```

Suppose that you hire a new type of `Employee` who earns a commission as well as a weekly salary. You can create a class with a name such as `CommissionEmployee`, and provide this class with three fields (`empNum`, `weeklySalary`, and `commissionRate`) and six methods (to get and set each of the three fields). However, this work would duplicate much of the work that you already have done for the `Employee` class. The wise and efficient alternative is to create the `CommissionEmployee` class so it inherits all the attributes and methods of `Employee`. Then, you can add just the single field and two methods (the get and set methods for the new field) that are additions within the new class. Figure 11-13 depicts these relationships. The complete `CommissionEmployee` class is shown in Figure 11-14.

Figure 11-13

Commissionemployee Inherits from Employee

Figure 11-14

`commissionemployee` Class

```
class CommissionEmployee inheritsFrom Employee
   Declarations
      private num commissionRate

   public void setCommissionRate(num rate)
      commissionRate = rate
   return

   public num getCommissionRate()
   return commissionRate
endClass
```

Recall that a plus sign in a class diagram indicates public access and a minus sign indicates private access. Figure 11-13 and several other figures in this chapter are examples of UML diagrams. Chapter 13 describes UML diagrams in more detail.

The class in Figure 11-14 uses the phrase `inheritsFrom Employee` (see shading) to indicate inheritance. Each programming language uses its own syntax. For example, using Java you would write `extends`, in Visual Basic you would write `inherits`, and in C++ and C# you would use a colon between the new class name and the one from which it inherits.

When you use inheritance to create the `CommissionEmployee` class, you acquire the following benefits:

- You save time, because you need not re-create the `Employee` fields and methods.
- You reduce the chance of errors, because the `Employee` methods have already been used and tested.
- You make it easier for anyone who has used the `Employee` class to understand the `CommissionEmployee` class, because such users can concentrate on the new features only.
- You reduce the chance for errors and inconsistencies in shared fields. For example, if your company decides to change employee ID numbers from four digits to five, and you have code in the `Employee` class constructor that ensures valid ID numbers, then you can simply change the code in the `Employee` class; every `CommissionEmployee` object will automatically acquire the change. Without inheritance, not only would you make the change in multiple places, but the likelihood would increase that you would forget to make the change in one of the classes.

The ability to use inheritance makes programs easier to write, easier to understand, and less prone to errors. Imagine that besides `CommissionEmployee`, you want to create several other more specific `Employee` classes (perhaps `PartTimeEmployee`, including a field for hours worked, or `DismissedEmployee`, including a reason for dismissal). By using inheritance, you can develop each new class correctly and more quickly.

In part, the concept of class inheritance is useful because it makes class code reusable. However, you do not use inheritance simply to save work. When properly used, inheritance always involves a general-to-specific relationship.

11-4a Understanding Inheritance Terminology

A class that is used as a basis for inheritance, like `Employee`, is called a **base class** (A class that is used as a basis for inheritance.) . When you create a class that inherits from a base class (such as `CommissionEmployee`), it is a **derived class** (An extended class.) or **extended class** (A derived class.) . When two classes have a base-derived relationship, you can distinguish the classes by using them in a sentence with the phrase is *a*. A derived class always "is a" case or instance of the more general base class. For example, a `Tree` class may be a base class to an `Evergreen` class. Every `Evergreen` is *a* `Tree`; however, it is not true that every `Tree` is an `Evergreen`. Thus, `Tree` is the base class and `Evergreen` is the derived class. Similarly, a `CommissionEmployee` *is an* `Employee`—not always the other way around —so `Employee` is the base class and `CommissionEmployee` is derived.

You can use the terms **superclass** (A base class.) and **subclass** (A derived class.) as synonyms for base class and derived class. Thus, `Evergreen` can be called a subclass of the `Tree` superclass. You also can use the terms **parent class** (A base class.) and **child class** (A derived class.) . A `CommissionEmployee` is a child to the `Employee` parent.

As an alternative way to discover which of two classes is the base class and which is the derived class, you can try saying the two class names together, although this technique might not work with every base-subclass pair. When people say their names in the English language, they state the more specific name before the all-encompassing family name, such as *Mary Johnson*. Similarly, with classes, the order that "makes more sense" is the child-parent order. Thus, because "Evergreen Tree" makes more sense than "Tree Evergreen," you can deduce that `Evergreen` is the child class. It also is convenient to think of a derived class as building upon its base class by providing the "adjectives" or additional descriptive terms for the "noun." Frequently, the names of derived classes are formed in this way, as in `CommissionEmployee` or `EvergreenTree`.

Finally, you usually can distinguish base classes from their derived classes by size. Although it is not required, a derived class is generally larger than its base class, in the sense that it usually has additional fields and methods. A subclass description may look small, but any subclass contains all of its base class's fields and methods as well as its own more specific fields and methods.

Do not think of a subclass as a *subset* of another class—in other words, as possessing only parts of its base class. In fact, a derived class usually contains more than its parent.

A derived class can be further extended. In other words, a subclass can have a child of its own. For example, after you create a `Tree` class and derive `Evergreen`, you might derive a `Spruce` class from `Evergreen`. Similarly, a `Poodle` class might derive from `Dog`, `Dog` from `DomesticPet`, and `DomesticPet` from `Animal`. The entire list of parent classes from which a child class is derived constitutes the **ancestors** (The entire list of parent classes from which a class is derived.) of the subclass.

After you create the `Spruce` class, you might be ready to create `Spruce` objects. For example, you might create `theTreeInMyBackYard`, or you might create an array of 1000 `Spruce` objects for a tree farm. On the other hand, before you are ready to create objects, you might first want to create even more specific child classes such as `ColoradoSpruce` and `NorwaySpruce`.

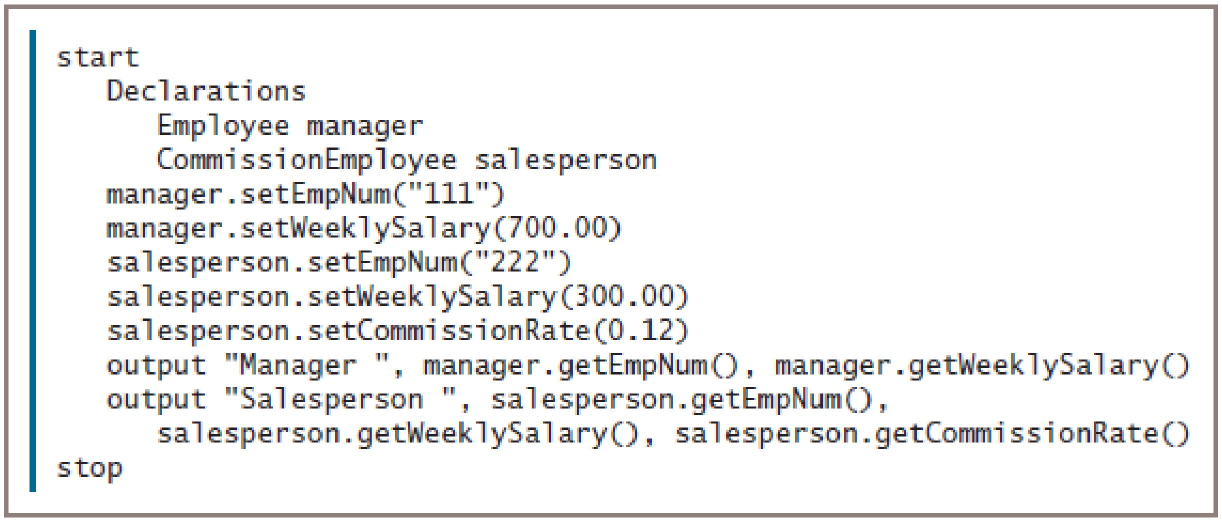
A child inherits all the data fields and methods of all its ancestors. In other words, when you declare a `Spruce` object, it contains all the attributes and methods of both an `Evergreen` and a `Tree`, and a `CommissionEmployee` contains all the attributes and methods of an `Employee`. In other words, the components of `Employee` and `CommissionEmployee` are as follows:

- `Employee` contains two fields and four methods, as shown in Figure 11-12.
- `CommissionEmployee` contains three fields and six methods, even though you do not see all of them in Figure 11-14.

Although a child class contains all the data fields and methods of its parent, a parent class does not gain any child class data or methods. Therefore, when `Employee` and `CommissionEmployee` classes are defined as in Figures 11-12 and 11-14, the statements in Figure 11-15 are all valid in an application. The `salesperson` object can use all the methods of its parent, and it can use its own `setCommissionRate()` and `getCommissionRate()` methods. Figure 11-16 shows the output of the program as it would appear in a commandline environment.

Figure 11-15

`Employeedemo` Application that Declares Two `Employee` Objects



```
start
   Declarations
      Employee manager
      CommissionEmployee salesperson
   manager.setEmpNum("111")
   manager.setWeeklySalary(700.00)
   salesperson.setEmpNum("222")
   salesperson.setWeeklySalary(300.00)
   salesperson.setCommissionRate(0.12)
   output "Manager ", manager.getEmpNum(), manager.getWeeklySalary()
   output "Salesperson ", salesperson.getEmpNum(),
      salesperson.getWeeklySalary(), salesperson.getCommissionRate()
stop
```

Figure 11-16

Output of the Program in Figure 11-15

The following statements would not be allowed in the `EmployeeDemo` application in Figure 11-15 because `manager`, as an `Employee` object, does not have access to the methods of the `CommissionEmployee` child class:

When you create your own inheritance chains, you want to place fields and methods at their most general level. In other words, a method named `grow()` rightfully belongs in a `Tree` class, whereas `leavesTurnColor()` does not because the method applies to only some of the `Tree` child classes. Similarly, a `leavesTurnColor()` method would be better located in a `DeciduousTree` class than separately within the `Oak` or `Maple` child classes.

It makes sense that a parent class object does not have access to its child's data and methods. When you create the parent class, you do not know how many future child classes might be created, or what their data or methods might look like. In addition, derived classes are more specific, so parent class objects cannot use them. For example, a `Cardiologist` class and an `Obstetrician` class are children of a `Doctor` class. You do not expect all members of the general parent class `Doctor` to have the `Cardiologist`'s `repairHeartValve()` method or the `Obstetrician`'s `performCaesarianSection()` method. However, `Cardiologist` and `Obstetrician` objects have access to the more general `Doctor` methods `takeBloodPressure()` and `billPatients()`. As with subclasses of doctors, it is convenient to think of derived classes as *specialists*. That is, their fields and methods are more specialized than those of the parent class.

In some programming languages, such as C#, Visual Basic, and Java, every class you create is a child of one ultimate base class, often called the `Object` class. The `Object` class usually provides basic functionality that is inherited by all the classes you create—for example, the ability to show its memory location and name.

11-4b Accessing Private Fields and Methods of a Parent Class

In Chapter 10 you learned that when you create classes, the most common scenario is for methods to be public but for data to be private. Making data private is an important concept in object-oriented programming. By making data fields private and allowing access to them only through a class's methods, you protect the ways in which data items can be altered and used.

When a data field within a class is private, no outside class can use it—including a child class. The principle of data hiding would be lost if you could access a class's private data merely by creating a child class. However, it can be inconvenient when the methods of a child class cannot directly access its own inherited data.

Watch the video *Inheritance*.

For example, suppose that some employees do not earn a weekly salary as defined in the `Employee` class, but are paid by the hour. You might create an `HourlyEmployee` class that descends from `Employee`, as shown in Figure 11-17. The class contains two new fields, `hoursWorked` and `hourlyRate`, and a get and set method for each.

Figure 11-17

Class Diagram for `HourlyEmployee` Class

You can implement the new class as shown in Figure 11-18. Whenever you set either `hoursWorked` or `hourlyRate`, you want to modify `weeklySalary` based on the product of the hours and rate. The logic makes sense, but the code does not compile. The two shaded statements show that the `HourlyEmployee` class is attempting to modify the `weeklySalary` field. Although every `HourlyEmployee` *has* a `weeklySalary` field by virtue of being a child of Employee, the HourlyEmployee class methods do not have access to the `weeklySalary` field, because `weeklySalary` is private within the `Employee` class. In this case, the private `weeklySalary` field is **inaccessible** (Describes any field or method that cannot be reached because of a logical error.) to any class other than the one in which it is defined.

Figure 11-18

Implementation of `HourlyEmployee` Class that Attempts to Access `weeklySalary`

One solution to this problem would be to make `weeklySalary` public in the parent `Employee` class. Then the child class could use it. However, that action would violate the important object-oriented principle of data hiding. Good object-oriented style dictates that your data should be altered only by the methods you choose and only in ways that you can control. If outside classes could alter an `Employee`'s private fields, then the fields could be assigned values that the `Employee` class could not control. In such a case, the principle of data hiding would be destroyed, causing the behavior of the object to be unpredictable.

Therefore, OOP languages allow a medium-security access specifier that is more restrictive than public but less restrictive than private. The **protected access specifier** (A specifier used when outside classes should not be able to use a data field unless they are children of the original class.) is used when you want no outside classes to be able to use a data field, except classes that are children of the original class. Figure 11-19 shows a rewritten `Employee` class that uses the `protected` access specifier on one of its data fields (see shading). When this modified class is used as a base class for another class, such as `HourlyEmployee`, the child class's methods will be able to access any protected items (fields or methods) originally defined in the parent class. When the `Employee` class is defined with a protected `weeklySalary` field, as shown in Figure 11-19, the code in the `HourlyEmployee` class in Figure 11-18 works correctly.

Figure 11-19

`Employee` Class with a Protected Field

Figure 11-20 contains the class diagram for the version of the `Employee` class shown in Figure 11-19. Notice that the `weeklySalary` field is preceded with an octothorpe (#)—the character that conventionally is used in class diagrams to indicate protected class members.

Figure 11-20

`Employee` Class with Protected Member

If `weeklySalary` is defined as protected instead of private in the `Employee` class, then either the creator of the class knew that a child class would want to access the field or the class was revised after it became known the child class would need access to the field.

If the `Employee` class's creator did not foresee that a field would need to be accessible, or if it is not preferable to revise the class, then `weeklySalary` will remain private. It is still possible to correctly set an `HourlyEmployee`'s weekly pay—the `HourlyEmployee` is just required to use the same means as any other class would. That is, the `HourlyEmployee` class can use the public method `setWeeklySalary()` that already exists in the parent class. Any class, including a child, can use a public field or method of the base class. So, assuming that `weeklySalary` remains private in `Employee`, Figure 11-21 shows how `HourlyEmployee` could be written to correctly set `weeklySalary`.

Figure 11-21

The `HourlyEmployee` Class When `weeklySalary` Remains Private

In the version of `HourlyEmployee` in Figure 11-21, the shaded statements within `setHoursWorked()` and `setHourlyRate()` assign a value to the corresponding child class field (`hoursWorked` or `hourlyRate`, respectively). Each method then calls the public parent class method `setWeeklySalary()`. In this example, no `protected` access specifiers are needed for any fields in the parent class, and the creators of the parent class did not have to foresee that a child class would eventually need to access any of its fields. Instead, any child classes of `Employee` simply follow the same access rules as any other outside class would. As an added benefit, if the parent class method `setWeeklySalary()` contained additional code (for example, to require a minimum base weekly pay for all employees), then that code would be enforced even for `HourlyEmployees`.

So, in summary, when a child class must access a private field of its parent's class, you can take one of several approaches:

- You can modify the parent class to make the field public. Usually, this is not advised because it violates the principle of data hiding.
- You can modify the parent class to make the field protected so that child classes have access to it, but other outside classes do not. This approach is necessary if you do not want public methods to be able to access the parent class field. Be aware that some programmers oppose making any data fields nonprivate. They feel that public methods should always control data access, even by a class's children.
- The child class can use a public method within the parent class that modifies the field, just as any other outside class would. This is frequently, but not always, the best

option.

Using the `protected` access specifier for a field can be convenient, and it improves program performance a little by using a field directly instead of "going through" another method. Also, using the `protected` access specifier is occasionally necessary when no existing public method accesses a field in a way required by the child class. However, protected data members should be used sparingly. Whenever possible, the principle of data hiding should be observed, and even child classes should have to go through methods to "get to" their parent's private data.

The likelihood of future errors increases when child classes are allowed direct access to a parent's fields. For example, if the company decides to add a bonus to every `Employee`'s weekly salary, you might make a change in the `setWeeklySalary()` method. If a child class is allowed direct access to the `Employee` field `weeklySalary` without using the `setWeeklySalary()` method, then any child class objects will not receive the bonus. Classes that depend on field names from parent classes are said to be **fragile** (Describes classes that depend on field names from parent classes and are prone to errors.) because they are prone to errors—that is, they are easy to "break."

Some OOP languages, such as C++, allow a subclass to inherit from more than one parent class. For example, you might create an `InsuredItem` class that contains data fields such as value and purchase date for each insured possession, and an `Automobile` class with appropriate data fields (for example, vehicle identification number, make, model, and year). When you create an `InsuredAutomobile` class for a car rental agency, you might want to include information and methods for `Automobiles` and `InsuredItems`, so you might want to inherit from both. The capability to inherit from more than one class is called **multiple inheritance** (The ability to inherit from more than one class.) .

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 an object that is "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 objects from, is an abstract class. An **abstract class** (An abstract class is one from which you cannot create concrete objects, but from which you can inherit.) is one from which you cannot create any concrete objects, but from which you can inherit.

11-4c Using Inheritance to Achieve Good Software Design

When an automobile company designs a new car model, it does not build every component of the new car from scratch. The company might design a new feature; for example, at some point a carmaker 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, existing 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 makes your job easier because you don't have to create every part of a new class from scratch. Professional programmers constantly create new class libraries for use with OOP languages. Having these classes available to use and extend makes programming large systems more manageable. 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** (The feature of modular programs that ensures a module has been tested and proven to function correctly.) .
- 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, neither the superclass source code nor the translated superclass code is changed. The superclass maintains its integrity.

When you consider classes, you must think about their commonalities, 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.

Two Truths & A Lie

Understanding Inheritance

1. When you create a class by making it inherit from another class, you save time because you need not re-create the base class fields and methods.

   T F

2. A class that is used as a basis for inheritance is called a base class, derived class, or extended class.

   T F

3. When a data field within a class is private, no outside class can use it—including a child class.

   T F

Chapter 11: More Object-Oriented Programming Concepts: 11-5 An Example of Using Predefined Classes: Creating GUI Objects
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)
© 2013 ,

11-5 An Example of Using Predefined Classes: Creating GUI Objects

When you purchase or download a compiler for an object-oriented programming language, it comes packaged with many predefined, built-in classes. The classes are stored in **libraries** (Stored collections of classes that serve related purposes.) or **packages** (Another name for libraries in some languages.) —collections of classes that serve related purposes. Some of the most helpful are the classes you can use to create graphical user interface (GUI) objects such as frames, buttons, labels, and text boxes. You place these GUI components within interactive programs so that users can manipulate them using input devices—most frequently a keyboard and a mouse. For example, if you want to place a clickable button on the screen using a language that supports GUI applications, you instantiate an object that belongs to an existing class with a name similar to `Button`. You then create objects with names such as `yesButton` or `buyProductNowButton`. The `Button` class contains private data fields such as `text` and `height` and public methods such as `setText()` and `setHeight()` that allow you to alter the objects' fields. For example, you might write a statement such as the following to change the text on a `Button` object:

If no predefined GUI object classes existed, you could create your own. However, this would present several disadvantages:

- It would be a lot of work. Creating graphical objects requires a substantial amount of code and at least a modicum of artistic talent.
- It would be repetitious work. Almost all GUI programs require standard components such as buttons and labels. If each programmer created the classes for these components from scratch, much of this work would be repeated unnecessarily.
- The components would look different in various applications. If each programmer created his or her own component classes, objects like buttons would look different and operate in slightly different ways. Users prefer standardization in their components—title bars on windows that are a uniform height, buttons that appear to be pressed when clicked, frames and windows that contain maximize and minimize buttons in predictable locations, and so on. By using standard component classes, programmers are assured that the GUI components in their programs have the same look and feel as those in other programs.

Programming languages that supply existing GUI classes often provide a **visual development environment** (A programming environment in which programs are created by dragging components such as buttons and labels onto a screen and arranging them visually.) in which you can create programs by dragging components such as buttons and labels onto a screen and arranging them visually. (In several languages, the visual development environment is known by the acronym **IDE** (The acronym for Integrated Development Environment, which is the visual development environment in some programming languages.) , which stands for *integrated development environment*.) Then you write programming statements to control the actions that take place when a user manipulates the controls—by clicking them using a mouse, for example. Many programmers never create classes of their own from which they will instantiate objects, but only write application classes that use built-in GUI component classes. Some languages—for example, Visual Basic and C#—lend themselves very well to this type of programming. In Chapter 12, you will learn more about creating programs that use GUI objects.

Two Truths & A Lie

An Example of Using Predefined Classes: Creating GUI Objects

1. Collections of classes that serve related purposes are called annals.

   T F

2. GUI components are placed within interactive programs so that users can manipulate them using input devices.

   T F

3. By using standard component classes, programmers are assured that the GUI components in their programs have the same look and feel as those in other programs.

   T F

11-6 Understanding Exception Handling

A great deal of the effort that goes into writing programs involves checking data items to make sure they are valid and reasonable. Professional data-entry operators who create the files used in business applications spend their entire working day entering facts and figures, so operators can and do make typing errors. When programs depend on data entered by average users who are not trained typists, the likelihood of errors is even greater.

Programmers use the acronym *GIGO* to describe what happens when worthless or invalid input causes inaccurate or unrealistic results. GIGO is an acronym for “garbage in, garbage out.”

In procedural programs, programmers handled errors in various ways that were effective, but the techniques had some drawbacks. The introduction of object-oriented programming has led to a new model called exception handling.

11-6a Drawbacks to Traditional Error-Handling Techniques

In traditional programming, probably the most common error-handling outcome was to terminate the program, or at least to terminate the method in which the offending statement occurred. For example, suppose that a program prompts a user to enter an insurance premium type from the keyboard, and that the entered value should be *A* or *H* for *Auto* or *Health*. Figure 11-22 shows a segment of pseudocode that causes the `determinePremium()` method to end if `policyType` is invalid; in the shaded `if` statement, the method ends abruptly when `policyType` is not *A* or *H*. This method of handling an error is not only unforgiving, it isn't even structured. Recall that a structured method should have one entry point and one exit point. The method in Figure 11-22 contains two exit points at the two `return` statements.

Figure 11-22

A Method that Handles an Error in an Unstructured Manner

In the example in Figure 11-22, if `policyType` is an invalid value, the method in which the code appears is terminated. The client program might continue with an invalid value or it might stop working. If the program that contains this method is part of a business program or a game, the user may be annoyed. However, an early termination in a program that monitors a hospital patient's vital signs or navigates an airplane might have far more serious consequences.

Rather than ending a method prematurely just because it encounters a piece of invalid data, a more elegant solution involves looping until the data item becomes valid, as shown in the highlighted portion of Figure 11-23. As long as the value of `policyType` is invalid, the user is prompted continuously to enter a new value. Only when `policyType` is *A* or *H* does the method continue.

Figure 11-23

Method that Handles an Error Using a Loop

The error-handling logic shown in Figure 11-23 has at least two shortcomings:

- The method is not as reusable as it could be.
- The method is not as flexible as it might be.

One of the principles of modular and object-oriented programming is reusability. The method in Figure 11-23 is only reusable under limited conditions. The `determinePremium()` method allows the user to reenter policy data any number of times, but other programs in the insurance system may need to limit the number of chances the user gets to enter correct data, or may allow no second chance at all. A more flexible `determinePremium()` method would simply calculate the premium amount without deciding what to do about data errors. The `determinePremium()` method will be most flexible if it can detect an error and then notify the calling program or method that an error has occurred. Each client that uses the `determinePremium()` method then can handle the mistake appropriately for the current application.

The other drawback to forcing the user to reenter data is that the technique works only with interactive programs. A more flexible program accepts any kind of input, including data stored on a disk. Program errors can occur as a result of many factors—for example, a disk drive might not be ready, a file might not exist on the disk, or stored data items might be invalid. You cannot continue to reprompt a disk file for valid data the way you can reprompt a user in an interactive program; if stored data is invalid, it remains invalid.

In the next section, you will learn object-oriented exception-handling techniques that overcome the limitations of traditional error handling.

11-6b The Object-Oriented Exception-Handling Model

Object-oriented programs employ a group of techniques for handling errors called

exception handling (The techniques for managing errors in objectoriented programs.) . The generic name used for errors in object-oriented languages is **exceptions** (The generic term used for an error in object-oriented languages. Presumably, errors are not usual occurrences; they are the “exceptions” to the rule.) because errors are not usual occurrences; they are the “exceptions” to the rule.

In object-oriented terminology, you **try** (To execute code that might throw an exception.) some code that might **throw an exception** (To pass an exception out of a block where it occurs, usually to a block that can handle it.) . If an exception is thrown, it is passed to a block of code that can **catch the exception** (To receive an exception from a throw so it can be handled.) , which means to receive it in a way similar to how a parameter is received by a method. In some languages, the exception object that is thrown can be any data type—a number, a string, or a programmer-created object. Even when a language permits any data type to be thrown, most programmers throw an object of the built-in class `Exception`, or they derive a class from a built-in `Exception` class. For example, Figure 11-24 shows a `determinePremium()` method that throws an exception only if `policyType` is neither *H* nor *A*. If `policyType` is invalid, an object of type `Exception` named `mistake` is instantiated and thrown from the method by a `throw` statement. A **throw statement** (A programming statement that sends an `Exception` object out of a method or code block to be handled elsewhere.) is one that sends an `Exception` object out of the current code block or method so it can be handled elsewhere. If `policyType` is *H* or *A*, the method continues, the premium is calculated, and the method ends naturally.

Figure 11-24

A Method that Creates and Throws an `Exception` Object

When you create a segment of code in which something might go wrong, you place the code in a **try block** (A block of code that attempts to execute while acknowledging that an exception might occur.) , which is a block of code you attempt to execute while acknowledging that an exception might occur. A `try` block consists of the keyword `try` followed by any number of statements, including some that might cause an exception to be thrown. If a statement in the block causes an exception, the remaining statements in the `try` block do not execute and the `try` block is abandoned. For pseudocode purposes, you can end a `try` block with a sentinel such as `endtry`.

You almost always code at least one `catch` block immediately following a `try` block. A **catch block** (A segment of code written to handle an exception that might be thrown by the `try` block that precedes it.) is a segment of code written to handle an exception that might be thrown by the `try` block that precedes it. Each `catch` block "catches" one type of exception—in many languages the caught object must be of type `Exception` or one of its child classes. You create a `catch` block using the following pseudocode elements:

- The keyword `catch`, followed by parentheses that contain an `Exception` type and an identifier
- Statements that take the action to handle the error condition
- An `endcatch` statement to indicate the end of the `catch` block in the pseudocode

Figure 11-25 shows a client program that calls the `determinePremium()` method. Because `determinePremium()` has the potential to throw an exception, the call to the method is contained in a `try` block. If `determinePremium()` throws an exception, the `catch` block in the program executes; if all goes well and `determinePremium()` does not throw an exception, the `catch` block is bypassed. A `catch` block looks like a method named `catch()` that takes an argument that is some type of `Exception`. However, it is not a method; it has no `return` type, and you can't call it directly.

Figure 11-25

A Program that Contains a `try`…`catch` Pair

In the program in Figure 11-25, a message is displayed when the exception is thrown. Another application might take different actions. For example, you might write an application in which the `catch` block forces the `policyType` to *H* or to *A*, or reprompts the user for a valid value. Various programs can use the `determinePremium()` method and handle an error in the way that is considered most appropriate.

In the method in Figure 11-25, the variable `mistake` in the `catch` block is an object of type `Exception`. The object is not used within the `catch` block, but it could be. For example, depending on the language, the `Exception` class might contain a method named `getMessage()` that returns a string explaining the cause of the error. In that case, you could place a statement such as `output mistake.getMessage()` in the `catch` block.

Even when a program uses a method that throws an exception, the exceptions are created and thrown only occasionally, when something goes wrong. Programmers sometimes refer to a situation in which nothing goes wrong as the **sunny day case** (A program execution in which nothing goes wrong.) .

The general principle of exception handling in object-oriented programming is that a method that uses data should be able to detect errors, but not be required to handle them. The handling should be left to the application that uses the object, so that each application can use each method appropriately.

Watch the video *Exception Handling*.

11-6c Using Built-in Exceptions and Creating Your Own Exceptions

Many OOP languages provide built-in `Exception` types. For example, Java, Visual Basic, and C# each provide dozens of categories of `Exceptions` that you can use in your programs. Every object-oriented language has an automatically created exception with a name like `ArrayOutOfBoundsException` that is thrown when you attempt to use an invalid subscript with an array. Similarly, an exception with a name like `DivideByZeroException` might be generated automatically if a program attempts to divide a number by zero.

Although some actions, such as dividing by zero, are errors in every programming situation, the built-in `Exceptions` in a programming language cannot cover *every* condition that might be an `Exception` in your applications. For example, you might want to declare an `Exception` when your bank balance is negative or when an outside party attempts to access your e-mail account. Most organizations have specific rules for exceptional data; for example, an employee number must not exceed three digits, or an hourly salary must not be less than the legal minimum wage. You can handle these potential error situations with `if` statements, but you also can create your own `Exceptions`.

To create your own throwable `Exception`, you usually extend a built-in `Exception` class. For example, you might create a class named `NegativeBankBalanceException` or `EmployeeNumberTooLargeException`. (When you create a class that derives from `Exception`, it is conventional, but not required, to use `Exception` in the name.) By inheriting from the `Exception` class, you gain access to methods contained in the parent class, such as those that display a default message describing the `Exception`.

Depending on the language you are using, you might be able to extend from other throwable classes as well as `Exception`.

In most object-oriented programming languages, a method can throw any number of exceptions. A `catch` block must be available for each type of exception that might be thrown.

Two Truths & A Lie

Understanding Exception Handling

1. In object-oriented terminology, you try some code that might throw an exception. The exception can then be caught and handled.

   T F

2. A `catch` block is a segment of code that can handle an exception that might be thrown by the `try` block preceding it.

   T F

3. The general principle of exception handling in object-oriented programming is that a method that uses data should be able to detect and handle most common errors.

   T F

11-7 Reviewing the Advantages of Object-Oriented Programming

In Chapter 10 and this chapter, you have been exposed to many concepts and features of object-oriented programming, which provide extensive benefits as you develop programs. Whether you instantiate objects from classes you have created or from those created by others, you save development time because each object automatically includes appropriate, reliable methods and attributes. When using inheritance, you can develop new classes more quickly by extending classes that already exist and work; you need to concentrate only on new features added by the new class. When using existing objects, you need to concentrate only on the interface to those objects, not on the internal instructions that make them work. By using polymorphism, you can use reasonable, easy-to-remember names for methods and concentrate on their purpose rather than on memorizing different method names.

Two Truths & A Lie

Reviewing the Advantages of Object-Oriented Programming

1. When you instantiate objects in programs, you save development time because each object automatically includes appropriate, reliable methods and attributes.

   T F

2. When using inheritance, you can develop new classes more quickly by extending existing classes that already work.

   T F

3. By using polymorphism, you can avoid the strict rules of procedural programming and take advantage of more flexible object-oriented

methods.

T F

11-8 Chapter Review

11-8a Chapter Summary

- A constructor is a method that establishes an object. A default constructor is one that requires no arguments; in OOP languages, a default constructor is created automatically by the compiler for every class you write. If you want to perform specific tasks when you create an instance of a class, then you can write your own constructor. Any constructor you write must have the same name as the class it constructs, and cannot have a return type. Once you write a constructor for a class, you no longer receive the automatically written default constructor.
- A destructor contains the actions you require when an instance of a class is destroyed, most often when the instance goes out of scope. As with constructors, if you do not explicitly create a destructor for a class, one is automatically provided. The most common way to declare a destructor explicitly is to use an identifier that consists of a tilde (~) followed by the class name. You cannot provide parameters to a destructor; as a consequence, destructors cannot be overloaded. Like a constructor, a destructor has no return type.
- A class can contain objects of another class as data fields. Creating whole-part relationships is known as composition or aggregation.
- When you create a class by making it inherit from another class, you are provided with prewritten and tested data fields and methods automatically. Using inheritance helps you save time, reduces the chance of errors and inconsistencies, and makes it easier for readers to understand your classes. A class that is used as a basis for inheritance is called a base class. A class that inherits from a base class is a derived class or extended class. The terms *superclass* and *parent class* are synonyms for *base class*. The terms *subclass* and *child class* are synonyms for *derived class*.
- Some of the most useful classes packaged in language libraries are used to create graphical user interface (GUI) objects such as frames, buttons, labels, and text boxes.

Programming languages that supply existing GUI classes often provide a visual development environment in which you can create programs by dragging components such as buttons and labels onto a screen and arranging them visually.

- Exception-handling techniques are used to handle errors in object-oriented programs. When you try a block of code, you attempt to use it, and if an exception occurs, it is thrown. A `catch` block of the correct type can receive the thrown exception and handle it. Many OOP languages provide built-in `Exception` types, and you can create your own types by extending the `Exception` class.
- When you use object-oriented programming techniques, you save development time because each object automatically includes appropriate, reliable methods and attributes. Efficiency is achieved through both inheritance and polymorphism.

Chapter Review

11-8b Key Terms

constructor (An automatically called method that establishes an object.)

default constructor (A constructor that requires no arguments.)

nondefault constructor (A constructor that requires at least one argument.)

destructor (A is an automatically called method that contains the actions you require when an instance of a class is destroyed.)

whole-part relationship (An association in which an object of one class is part of an object of a larger whole class.)

composition (The technique of placing an object within an object of another class.)

has-a relationship (A whole-part relationship; the type of relationship that exists when using composition.)

base class (A class that is used as a basis for inheritance.)

derived class (An extended class.)

extended class (A derived class.)

superclass (A base class.)

subclass (A derived class.)

parent class (A base class.)

child class (A derived class.)

ancestors (The entire list of parent classes from which a class is derived.)

inaccessible (Describes any field or method that cannot be reached because of a logical error.)

protected access specifier (A specifier used when outside classes should not be able to use a data field unless they are children of the original class.)

fragile (Describes classes that depend on field names from parent classes and are prone to errors.)

multiple inheritance (The ability to inherit from more than one class.)

abstract class (An abstract class is one from which you cannot create concrete objects, but from which you can inherit.)

reliable (The feature of modular programs that ensures a module has been tested and proven to function correctly.)

libraries (Stored collections of classes that serve related purposes.)

packages (Another name for libraries in some languages.)

visual development environment (A programming environment in which programs are created by dragging components such as buttons and labels onto a screen and arranging them visually.)

IDE (The acronym for Integrated Development Environment, which is the visual development environment in some programming languages.)

exception handling (The techniques for managing errors in objectoriented programs.)

exceptions (The generic term used for an error in object-oriented languages. Presumably, errors are not usual occurrences; they are the “exceptions” to the rule.)

try (To execute code that might throw an exception.)

throw an exception (To pass an exception out of a block where it occurs, usually to a block that can handle it.)

catch the exception (To receive an exception from a throw so it can be handled.)

throw statement (A programming statement that sends an `Exception` object out of a method or code block to be handled elsewhere.)

try block (A block of code that attempts to execute while acknowledging that an exception might occur.)

catch block (A segment of code written to handle an exception that might be thrown by the `try` block that precedes it.)

sunny day case (A program execution in which nothing goes wrong.)

Chapter Review

11-8c Review Questions

1. When you instantiate an object, the automatically created method that is called is a _____ .
    a. creator
    b. initiator
    c. constructor
    d. architect
2. Every class has _____ .
    a. exactly one constructor
    b. at least one constructor
    c. at least two constructors
    d. a default constructor and a programmer-written constructor
3. Which of the following can be overloaded?
    a. constructors
    b. instance methods
    c. both of the above
    d. none of the above
4. A default constructor is _____ .

   a. another name that is used only for a class's automatically created constructor
   b. a constructor that requires no arguments
   c. a constructor that sets a value for every field in a class
   d. the only constructor that is explicitly written in a class
5. When you write a constructor that receives a parameter, ____ .
   a. the parameter must be numeric
   b. the parameter must be used to set a data field
   c. the automatically created default constructor no longer exists
   d. the constructor body must be empty
6. When you write a constructor that receives no parameters, ____ .
   a. the automatically created constructor no longer exists
   b. it becomes known as the default constructor
   c. both of the above
   d. none of the above
7. Most often, a destructor is called when ____ .
   a. an object is created
   b. an object goes out of scope
   c. you make an explicit call to it
   d. a value is returned from a class method
8. Which of the following is not a similarity between constructors and destructors?
   a. Both can be called automatically.
   b. Both have the same name as the class.
   c. Both have no return type.
   d. Both can be overloaded.

9. Advantages of creating a class that inherits from another include all of the following *except*:

    a. You save time because subclasses are created automatically from those that come built in as part of a programming language.

    b. You save time because you need not re-create the fields and methods in the original class.

    c. You reduce the chance of errors because the original class's methods have already been used and tested.

    d. You make it easier for anyone who has used the original class to understand the new class.

10. Employing inheritance reduces errors because ____ .

    a. the new classes have access to fewer data fields

    b. the new classes have access to fewer methods

    c. you can copy and paste methods that you already created

    d. many of the methods you need have already been used and tested

11. A class that is used as a basis for inheritance is called a ____ .

    a. derived class

    b. subclass

    c. child class

    d. base class

12. Which of the following is another name for a derived class?

    a. base class

    b. child class

    c. superclass

    d. parent class

13. Which of the following is *not* another name for a derived class?

   a. extended class
   b. superclass
   c. child class
   d. subclass

14. Which of the following is true?
    a. A base class usually has more fields than its descendent.
    b. A child class can also be a parent class.
    c. A class's ancestors consist of its entire list of children.
    d. To be considered object oriented, a class must have a child.

15. Which of the following is true?
    a. A derived class inherits all the data and methods of its ancestors.
    b. A derived class inherits only the public data and methods of its ancestors.
    c. A derived class inherits only the private data and methods of its ancestors.
    d. A derived class inherits none of the data and methods of its ancestors.

16. Which of the following is true?
    a. A class's data fields usually are public.
    b. A class's methods usually are public.
    c. both of the above
    d. none of the above

17. A ____ is a collection of predefined, built-in classes that you can use when writing programs.
    a. vault
    b. black box
    c. library

   d. store

18. An environment in which you can develop GUI programs by dragging components to their desired positions is a(n) ____ .

    a. visual development environment

    b. integrated compiler

    c. text-based editor

    d. GUI formatter

19. In object-oriented programs, errors are known as ____ .

    a. faults

    b. gaffes

    c. exceptions

    d. omissions

20. The general principle of exception handling in object-oriented programming is that a method that uses data should ____ .

    a. be able to detect errors, but not be required to handle them

    b. be able to handle errors, but not detect them

    c. be able to handle and detect errors

    d. not be able to detect or handle errors

Chapter Review

11-8d Exercises

1. Complete the following tasks:

a. Design a class named `Circle` with fields named `radius, area,` and `diameter`. Include a constructor that sets the radius to 1. Include get methods for each field, but include a set method only for the radius. When the radius is set, do not allow it to be zero or a negative number. When the radius is set, calculate the diameter (twice the radius) and the area (the radius squared times pi, which is approximately 3.14). Create the class diagram and write the pseudocode that defines the class.

b. Design an application that declares two `Circles`. Set the radius of one manually, but allow the other to use the default value supplied by the constructor. Then, display each `Circle`'s values.

2. Complete the following tasks:

   a. Design a class named `PhoneCall` with four fields: two strings that hold the 10- digit phone numbers that originated and received the call, and two numeric fields that hold the length of the call in minutes and the cost of the call. Include a constructor that sets the phone numbers to *X*s and the numeric fields to 0. Include get and set methods for the phone number and call length fields, but do not include a set method for the cost field. When the call length is set, calculate the cost of the call at three cents per minute for the first 10 minutes, and two cents per subsequent minute. Create the class diagram and write the pseudocode that defines the class.

   b. Design an application that declares three `PhoneCalls`. Set the length of one `PhoneCall` to 10 minutes, another to 11 minutes, and allow the third object to use the default value supplied by the constructor. Then, display each `PhoneCall`'s values.

   c. Create a child class named `InternationalPhoneCall`. Override the parent class method that sets the call length to calculate the cost of the call at 40 cents per minute.

   d. Create the logic for an application that instantiates a `PhoneCall` object and an `InternationalPhoneCall` object and displays the costs for both.

3. Complete the following tasks:

   a. The Rockford *Daily Clarion* wants you to design a class named `Issue`. Fields include the issue number, total number of advertisements sold in the issue, and total advertising revenue. Include get and set methods for each field. Include a static method that displays the newspaper's motto ("Everything you need to know"). Include three overloaded constructors as follows:

- A default constructor that sets the issue number to 1 and the other fields to 0
- A constructor that allows you to pass values for all three fields
- A constructor that allows you to pass an issue number and a number of advertisements sold, but sets the advertising revenue to $50 per ad

Create the class diagram and write the pseudocode that defines the class.

b. Design an application that declares three `Issue` objects using a different constructor version with each object. Display each `Issue`’s values and then display the motto.

4. Complete the following tasks:

    a. Create a class named `BankAccount` that includes two numeric variables: a bank balance and an interest rate. Also create two overloaded methods named `computeInterest()`. The first method takes two numeric arguments representing balance and rate, multiplies them, and then displays the results. The second method takes a single argument representing balance. When this method is called, the interest rate is assumed to be 1.5 percent and the results are displayed.

    b. Create an application that declares two `BankAccount` objects and demonstrates how both method versions can be called.

5. Complete the following tasks:

    a. Create a class named `Pay` that includes five numeric variables: hours worked, hourly pay rate, withholding rate, gross pay, and net pay. Also create three overloaded `computeNetPay()` methods. When `computeNetPay()` receives values for hours, pay rate, and withholding rate, it computes the gross pay and reduces it by the appropriate withholding amount to produce the net pay. (Gross pay is computed as hours worked multiplied by hourly pay rate.) When `computeNetPay()` receives two arguments, they represent the hours and pay rate, and the withholding rate is assumed to be 15 percent. When `computeNetPay()` receives one argument, it represents the number of hours worked; the withholding rate is assumed to be 15 percent and the hourly rate is assumed to be 6.50.

   b. Create an application that demonstrates all the methods.

6. Complete the following tasks:

   a. Design a class named `Book` that holds a stock number, author, title, price, and number of pages. Include methods to set and get the values for each data field. Also include a `displayInfo()` method that displays each of the Book's data fields with explanations.

   b. Design a class named `TextBook` that is a child class of Book. Include a new data field for the grade level of the book. Override the Book class `displayInfo()` method to accommodate the new grade-level field.

   c. Design an application that instantiates an object of each type and demonstrates all the methods.

7. Complete the following tasks:

   a. Design a class named `Player` that holds a player number and name for a sports team participant. Include methods to set the values for each data field and output the values for each data field.

   b. Design two classes named `BaseballPlayer` and `BasketballPlayer` that are child classes of `Player`. Include a new data field in each class for the player's position. Include an additional field in the `BaseballPlayer` class for batting average. Include a new field in the `BasketballPlayer` class for free-throw percentage. Override the `Player` class methods that set and output the data so that you accommodate the new fields.

   c. Design an application that instantiates an object of each type and demonstrates all the methods.

8. Complete the following tasks:

   a. Create a class for a cell phone service named `Message` that includes a field for the price of the message. Create get and set methods for the field.

   b. Derive three subclasses—`VoiceMessage`, `TextMessage`, and `PictureMessage`. The `VoiceMessage` class includes a numeric field to hold the length of the message in minutes and a get and set method for the field. When a `VoiceMessage`'s length value is set, the price is calculated at 4 cents per minute. The `TextMessage` class includes a numeric field to hold the length of the message in words and a get and set method for the field. When a `TextMessage`'s length value is set, the price is calculated at 2 cents per word. The `PictureMessage` class includes a numeric field that holds the size of the picture in kilobytes and get and set methods for the field. When a `PictureMessage`'s length value is set, the price is calculated at 1 cent per kilobyte.

   c. Design a program that instantiates one object of each of the three classes, and demonstrate using all the methods defined for each class.

9. Complete the following tasks:

   a. Create a class named `Order` that performs order processing of a single item. The class has four fields: customer name, customer number, quantity ordered, and unit price. Include set and get methods for each field. The set methods prompt the user for values for each field. This class also needs a `computePrice()` method to compute the total price (quantity multiplied by unit price) and a method to display the field values.

   b. Create a subclass named `ShippedOrder` that overrides `computePrice()` by adding a shipping and handling charge of $4.00.

   c. Create the logic for an application that instantiates an object of each of these two classes. Prompt the user for data for the `Order` object and display the results; then prompt the user for data for the `ShippedOrder` object and display the results.

   d. Create the logic for an application that continuously prompts for order information until the user enters *ZZZ* for the customer name or 10 orders have been taken, whichever comes first. Ask the user whether each order will be shipped, and create an `Order` or a `ShippedOrder` appropriately. Store each order in an array. When the user finishes entering data, display all the order information taken as well as the total price that was computed for each order.

10. Complete the following tasks:

   a. Design a method that calculates the cost of a weekly cleaning job for Molly's Maid Service. Variables include a job location code of *B* for business, which costs $200, or R for residential, which costs $140. The method should throw an exception if the location code is invalid.

   b. Write a method that calls the method designed in Exercise 10a. If the method throws an exception, force the price of the job to 0.

   c. Write a method that calls the method designed in Exercise 10a. If the method throws an exception, require the user to reenter the location code.

   d. Write a method that calls the method designed in Exercise 10a. If the method throws an exception, force the location code to *R* and the price to $140.

11. Design a method that calculates the monthly cost to rent a roadside billboard. Variables include the size of the billboard (*S*, *M*, or *L* for small, medium, or large) and its location (*H*, *M*, or *L* for high-, medium-, or low-traffic areas). The method should throw an exception if the size or location code is invalid. The monthly rental cost is shown in Table 11-1.

Table 11-1

Monthly Billboard Rental Rates

	High Traffic	**Medium Traffic**	**Low Traffic**
Small size	100.00	65.00	35.00
Medium size	150.00	95.00	48.00
Large size	210.00	130.00	60.00

Find the Bugs

12. Your downloadable files for Chapter 11 include DEBUG11-01.txt, DEBUG11-02.txt, and DEBUG11-03.txt. Each file starts with some comments that describe the problem. Comments are lines that begin with two slashes (//). Following the comments, each file contains pseudocode that has one or more bugs you

must find and correct.

Game Zone

13. Computer games often contain different characters or creatures. For example, you might design a game in which alien beings possess specific characteristics such as color, number of eyes, or number of lives. Create an `Alien` class. Include at least three data fields of your choice. Include a constructor that requires a value for each data field and a method named `toString()` that returns a string containing a complete description of the `Alien`.

    Create two classes—`Martian` and `Jupiterian`—that descend from `Alien`. Supply each with a constructor that sets the `Alien` data fields with values you choose. For example, you can decide that a `Martian` has four eyes but a `Jupiterian` has only two.

    Create an application that instantiates one `Martian` and one `Jupiterian`. Call the `toString()` method with each object and display the results.

14. In Chapter 2, you learned that in many programming languages you can generate a random number between 1 and a limiting value named `LIMIT` by using a statement similar to `randomNumber = random(LIMIT)`. In Chapter 4 and 5, you created and fine-tuned the logic for a guessing game in which the application generates a random number and the player tries to guess it. As written, the game should work as long as the player enters numeric guesses. However, if the player enters a letter or other nonnumeric character, the game throws an automatically generated exception. Improve the game by handling any exception so that the user is informed of the error and allowed to enter data again.

15. a. In Chapter 10, you developed a `Card` class that contains a string data field to hold a suit and a numeric data field for a value from 1 to 13. Now extend the class to create a class called BlackjackCard. In the game of Blackjack, each card has a point value as well as a face value. These two values match for cards with values of 2 through 10, and the point value is 10 for jacks, queens, and kings (face values 11 through 13). For a simplified version of the game, assume that the value of the ace is 11. (In the official version of Blackjack, the player chooses whether each ace is worth 1 or 11 points.)

b. Randomly assign values to 10 `BlackjackCard` objects, then design an application that plays a modified version of Blackjack. The objective is to accumulate cards whose total value equals 21, or whose value is closer to 21 than the opponent's total value without exceeding 21. Deal five `BlackjackCards` each to the player and the computer. Make sure that each `BlackjackCard` is unique. For example, a deck cannot contain more than one queen of spades.

   Determine the winner as follows:

   - If the player's first two, first three, first four, or all five cards have a total value of exactly 21, the player wins, even if the computer also achieves a total of 21.
   - If the player's first two cards do not total exactly 21, sum as many as needed to achieve the highest possible total that does not exceed 21. For example, suppose that the player's five cards are valued as follows: 10, 4, 5, 9, 2. In that case, the player's total for the first three cards is 19; counting any more cards would cause the total to exceed 21.
   - After you have determined the player's total, sum the computer's cards in sequence. For example, suppose that the computer's cards are 10, 10, 5, 6, 7. The first two cards total 20; you would not use the third card because it would cause the total to exceed 21.
   - The winner has the highest total among the cards used. For example, if the player's total using the first three cards is 19 and the computer's total using the first two cards is 20, the computer wins.

   Display a message that indicates whether the game ended in a tie, the computer won, or the player won.

Up for Discussion

16. Many programmers think object-oriented programming is a superior approach to procedural programming. Others think it adds a level of complexity that is not needed in many scenarios. Find and summarize arguments on both sides. With which side do you agree?
17. Many object-oriented programmers are opposed to using multiple

inheritance. Find out why and decide whether you agree with this stance.

18. If you are completing all the programming exercises in this book, you can see how much work goes into planning a full-blown professional program. How would you feel if someone copied your work without compensating you? Investigate the magnitude of software piracy in our society. What are the penalties for illegally copying software? Are there circumstances in which it is acceptable to copy a program? If a friend asked you to copy a program for him, would you? What should we do about this problem, if anything?