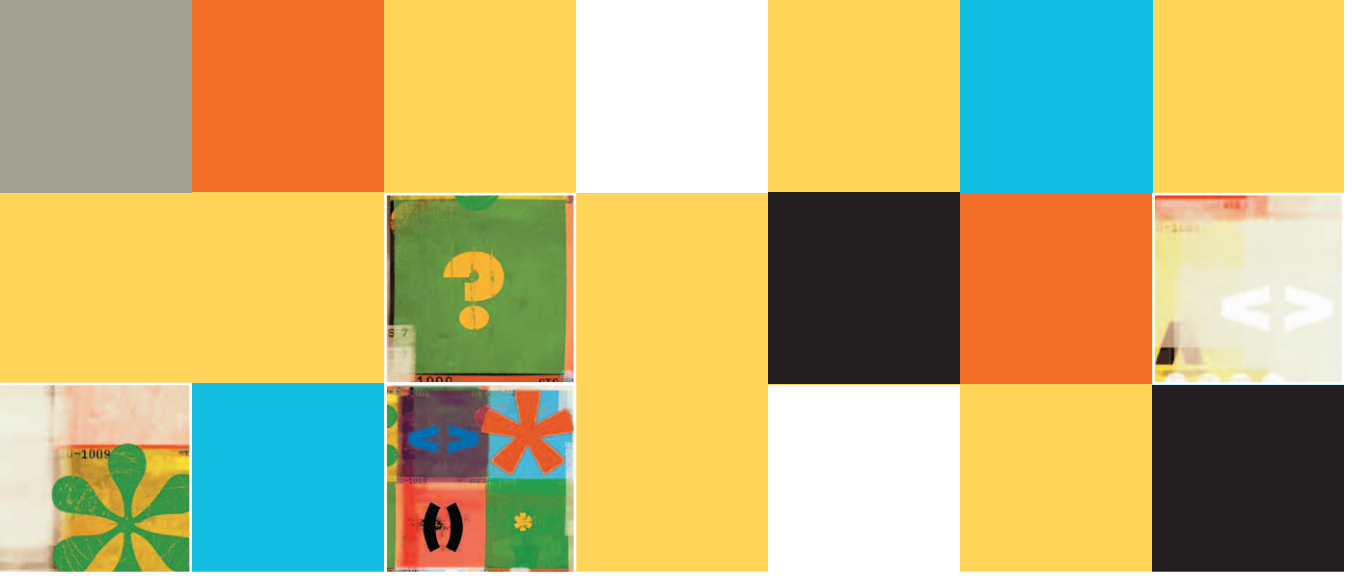


14 EVENT-DRIVEN PROGRAMMING WITH GRAPHICAL USER INTERFACES

After studying Chapter 14, you should be able to:

- Understand the principles of event-driven programming
- Describe user-initiated actions and GUI components
- Design graphical user interfaces
- Modify the attributes of GUI components
- List the steps to building an event-driven application
- Understand the disadvantages of traditional error-handling techniques
- Understand the advantages of the object-oriented technique of throwing exceptions

UNDERSTANDING EVENT-DRIVEN PROGRAMMING

From the 1950s, when people began to use computers to help them perform many jobs, right through the 1960s and 1970s, almost all interaction between human beings and computers was based on the command line. The **command line** is the location on your computer screen at which you type entries to communicate with the computer's operating system. An **operating system** is the software that you use to run a computer and manage its resources. Interacting with a computer operating system was difficult because the user had to know the exact syntax (that is, the correct sequence of words and symbols that form the operating system's command set) to use when typing commands, and had to spell and type those commands accurately.

TIP The command line also is called the **command prompt**. People who use the DOS operating system also call the command line the **DOS prompt**.

TIP If you use the Windows operating system on a PC, you can locate the command prompt by clicking Start, pointing to All Programs or Programs, pointing to Accessories, and then clicking Command Prompt.

Fortunately for today's computer users, operating system software is available that allows them to use a mouse or other pointing device to select pictures, or **icons**, on the screen. This type of environment is a **graphical user interface**, or **GUI**. Computer users can expect to see a standard interface in the GUI programs they use. Rather than memorizing difficult commands that must be typed at a command line, GUI users can select options from menus and click buttons to make their preferences known to a program. Users can select objects that look like their real-world counterparts and get the expected results. For example, users may select an icon that looks like a pencil when they want to write a memo, or they may drag an icon shaped like a folder to another icon that resembles a recycling bin when they want to delete the folder. Performing an operation on an icon (for example, clicking or dragging it) causes an **event**—an occurrence that generates a message sent to an object.

GUI programs are called **event-based** or **event-driven** because actions occur in response to user-initiated events such as clicking a mouse button. When you program with event-driven languages, the emphasis is on the objects that the user can manipulate, such as buttons and menus, and on the events that the user can initiate with those objects, such as clicking or double-clicking. The programmer writes instructions within modules that correspond to each type of event.

For the programmer, event-driven programs require unique considerations. The program logic you have developed so far for most of this book is procedural; each step occurs in the order the programmer determines. In a procedural program, if you issue a prompt and a statement to read the user's response, you have no control over how much time the user takes to enter a response, but you do control the sequence of events—the processing goes no further until the input is completed. In contrast, with event-driven programs, the user might initiate any number of events in any order. For example, if you use an event-driven word-processing program, you have dozens of choices at your disposal at any moment. You can type words, select text with the mouse, click a button to change text to bold or to italics, choose a menu item, and so on. With each word-processing document you create, you choose options in any order that seems appropriate at the time. The word-processing program must be ready to respond to any event you initiate.

Within an event-driven program, a component from which an event is generated is the **source** of the event. A button that a user can click is an example of a source; a text field that one can use to enter text is another source. An object that is "interested in" an event you want it to respond to is a **listener**. It "listens for" events so it knows when to respond. Not all objects can receive all events—you probably have used programs in which clicking on many areas of the screen has no effect at all. If you want an object, such as a button, to be a listener for an event, such as a mouse click, you must write the appropriate program statements.

Although event-based programming is relatively new, the instructions that programmers write to correspond to events are still simply sequences, selections, and loops. Event-driven programs still declare variables, use arrays, and contain all the attributes of their procedural-program ancestors. An event-based program may contain components with labels like "Sort Records," "Merge Files," or "Total Transactions." The programming logic you use when writing code for each of these processes is the same logic you have learned throughout this book. Writing event-driven programs simply involves thinking of possible events as the modules that constitute the program.

TIP In object-oriented languages, the procedural modules that depend on user-initiated events are often called *scripts*. The term "script" is also used to describe a relatively short computer program that performs one specific task. Scripts are commonly used to process user information from Web pages.

USER-INITIATED ACTIONS AND GUI COMPONENTS

To understand GUI programming, you need to have a clear picture of the possible events a user can initiate. These include the events listed in Table 14-1.

TABLE 14-1: COMMON USER-INITIATED EVENTS

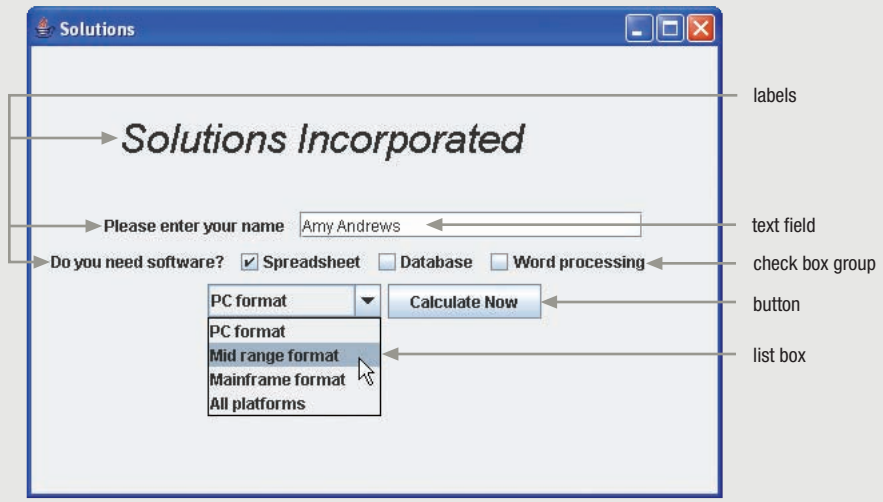
Event	Description
Key press	Pressing a key on the keyboard
Mouse point	Placing the mouse pointer over an area on the screen
Mouse click or left mouse click	Pressing the left mouse button
Right mouse click	Pressing the right mouse button
Mouse double-click	Pressing the left mouse button two times in rapid sequence
Mouse drag	Holding the left mouse button down while moving the mouse over the desk surface

You also need to be able to picture common GUI components. Some are listed in Table 14-2. Figure 14-1 shows a screen that contains several common GUI components.

TABLE 14-2: COMMON GUI COMPONENTS

GUI components	Description
Label	A rectangular area that displays text
Text field	A rectangular area into which the user can type a line of text
Button	A rectangular object you can click; usually it appears to press inward like a push button
Check box	A label positioned beside a square; you can click the square to display or remove a check mark—allows the user to turn an option on or off
Option buttons	A group of check-box-type objects in which the options are mutually exclusive; when the user selects any one option, the others are turned off—when the objects are square, they are often called a check box group, whereas when they are round, they are often called a set of radio buttons
List box	A menu of options that appears when the user clicks a list arrow; when the user selects an option from the list, the selected item replaces the original item in the display—all other items are unselected (with some list boxes, the user can make multiple selections)
Toolbar	A strip of icons that activate menu items

FIGURE 14-1: ILLUSTRATION OF COMMON GUI COMPONENTS



When you program in a language that supports event-driven logic, typically you do not create the GUI components you need from scratch. Instead, you call prewritten routines or methods that draw the GUI components on the screen for you. The components themselves are existing objects complete with names, attributes, and methods. In some programming

languages, you write statements that call the methods that create the GUI objects; in others, you can drag GUI objects onto your screen from a toolbar. Either way, you do not worry about the details of constructing the components. Instead, you concentrate on the actions that you want to take place when a user initiates an event from one of the components. Thus, GUI components are excellent examples of the best principles of object-oriented programming—they represent objects with attributes and methods that operate like black boxes, making them easy for you to use.

TIP

GUI components are often referred to as *widgets*. Some sources claim that the term is short for *window gadgets* or *Web gadgets*. The term "widgets" also is used in business textbooks to refer to a product whose specific identity or function is irrelevant; it was first used in this context in a 1924 play, "Beggar on Horseback." In computing, "widget" also is used to refer to a small, specialized desktop application such as a calendar or calculator.

When you use existing GUI components, you are instantiating objects that belong to a prewritten class. For example, you might use a `Button` class object when you want the user to be able to click a button to make a selection. Depending on the programming language you use, the `Button` class might contain attributes or properties such as `color` and `text` and methods such as `setText()`, in which you define the words that appear on the `Button`'s surface, and `click()`, in which you define the actions that will take place when a user clicks the `Button` object. To create a `Button` object, you might write a statement similar to `Button myProgramButton`, in which `Button` represents the type and `myProgramButton` represents the object you create.

DESIGNING GRAPHICAL USER INTERFACES

You should consider several general design principles when creating a program that will use a GUI:

- The interface should be natural and predictable.
- The screen design should be attractive and user-friendly.
- It's helpful if the user can customize your applications.
- The program should be forgiving.
- The GUI is only a means to an end.

THE INTERFACE SHOULD BE NATURAL AND PREDICTABLE

The GUI program interface should represent objects with icons that are like their real-world counterparts. In other words, it makes sense to use an icon that looks like a recycling bin when you want to allow a user to drag files or other components to the bin to delete them. Using a recycling bin icon is "natural" in that people use one in real life when they want to discard real-life items; dragging files to the bin is also "natural" because that's what people do with real-life items they discard. Using a recycling bin for discarded items is also predictable, because a number of other programs with which users are already familiar employ the recycling bin icon. Some icons may be natural, but if they are not predictable as well, then they are not as effective. An icon that depicts a recycling truck is just as "natural" as far as corresponding to the real world, but because other programs do not use a truck icon for this purpose, it is not as predictable.

Graphical user interfaces should also be predictable in their layout. For example, with most GUI programs, you use a menu bar at the top of the screen, and the first menu item is almost always *File*. If you design a program interface in which the menu bar runs vertically down the right side of the screen, or in which *File* is the last menu option instead of the first, you will confuse the people who use your program. Either they will make mistakes when using it, or they may give up using it entirely. It doesn't matter if you can prove that your layout plan is more efficient than the standard one—if you do not use a predictable layout, your program will meet rejection from users in the marketplace.

TIP

Many studies have proven that the Dvorak keyboard layout is more efficient for typists than the Qwerty keyboard layout that most of us use. The Qwerty keyboard layout gets its name from the first six letter keys in the top row. With the Dvorak layout, which gets its name from its inventor, the most frequently used keys are in the home row, allowing typists to complete many more keystrokes per minute. However, the Dvorak keyboard has not caught on with the computer-buying public because it is not predictable to users trained on the Qwerty keyboard.

TIP

Stovetops often have an unnatural interface, making unfamiliar stoves more difficult for you to use. Many stovetops have four burners arranged in two rows, but the knobs that control the burners frequently are placed in a single horizontal row. Because there is not a natural correlation between the placement of a burner and its control, you are more likely to select the wrong knob when adjusting the burner's flame or heating element.

THE SCREEN DESIGN SHOULD BE ATTRACTIVE AND USER-FRIENDLY

If your interface is attractive, people are more likely to use it. If it is easy to read, users are less likely to make mistakes and more likely to want to use it. And if the interface is easy to read, it will more likely be considered attractive. When it comes to GUI design, fancy fonts and weird color combinations are signs of amateur designers. In addition, you should make sure that unavailable screen options are either sufficiently disabled or removed, so the user does not waste time clicking components that aren't functional.

TIP

Disabling a component is frequently indicated by **dimming** or **graying** the component—that is, muting or softening its appearance. Disabling a component provides another example of predictability—users with computer experience do not expect to be able to use a dimmed component.

Screen designs should not be distracting. When there are too many components on a screen, users can't find what they're looking for. When a text field or button is no longer needed, it should be removed from the interface. You also want to avoid distracting users with overly creative design elements. When users click a button to open a file, they might be amused the first time a file name dances across the screen, or the speakers play a tune. But after one or two experiences with your creative additions, users find that intruding design elements simply hamper the actual work of the program.

TIP

GUI programmers sometimes refer to screen space as *real estate*. Just as a plot of real estate becomes unattractive when it supports no open space, your screen becomes unattractive when you fill the limited space with too many components.

IT'S HELPFUL IF THE USER CAN CUSTOMIZE YOUR APPLICATIONS

Every user works in his or her own way. If you are designing an application that will use numerous menus and toolbars, it's helpful if users can position the components in an order that's convenient for them. Users appreciate being able to change features such as color schemes. Allowing a user to change the background color in your application may seem frivolous to you, but to users who are color-blind or visually impaired, it might make the difference in whether they use your application at all.

TIP The screen design issues that make programs easier to use for people with physical limitations are known as **accessibility** issues.

THE PROGRAM SHOULD BE FORGIVING

Perhaps you have had the inconvenience of accessing a voice mail system in which you selected several sequential options, only to find yourself at a dead end with no recourse but to hang up and redial the number. Good program design avoids such problems. You should always provide an escape route to accommodate users who have made bad choices or changed their minds. By providing a Back button or functional Escape key, you provide more functionality to your users.

THE GUI IS ONLY A MEANS TO AN END

The most important principle of GUI design is to always remember that any GUI is only an interface. Using a mouse to click items and drag them around is not the point of any business program (except one that trains people how to use a mouse). Instead, the point of a graphical interface is to help people be more productive. To that end, the design should help the user see what options are available, allow the use of components in the ordinary way, and not force the user to concentrate on how to interact with your application. The real work of any GUI program is done after the user clicks a button or makes a list box selection.

MODIFYING THE ATTRIBUTES OF GUI COMPONENTS

When you design a program with premade or preprogrammed graphical components, you will want to change their appearance to customize them for the current application. Each programming language provides its own means of changing the appearance of components, but all involve changing the values stored in the components' attribute fields. Some common changes include setting the following items:

- Setting the size of the component
- Setting the color of the component
- Setting the screen location of the component
- Setting the font for any text in the component
- Setting the component to be visible or invisible
- Setting the component to be dimmed or undimmed, sometimes called enabled or disabled

You must learn the exact names of the methods and what type of arguments you are allowed to use in each programming language you learn, but all languages that support creating event-driven applications allow you to set components' attributes. With some languages, you set attributes by coding assignment statements, such as `myButton.text = "Push here"`. With other languages, you might change an attribute by calling a method and sending an argument, using a statement such as `myButton.setText("Push here")`. With other languages, you can access a properties list for every GUI object you create, and simply type the needed text into a table of attributes.

THE STEPS TO DEVELOPING AN EVENT-DRIVEN APPLICATION

In Chapter 1, you first learned the steps to developing a computer program. They are:

1. Understand the problem.
2. Plan the logic.
3. Code the program.
4. Translate the program into machine language.
5. Test the program.
6. Put the program into production.

Developing an event-driven application is more complicated than developing a standard procedural program. You can include three new steps between understanding the problem and developing the logic. The complete list of development steps for an event-driven application is as follows:

1. Understand the problem.
2. Create storyboards.
3. Define the objects.
4. Define the connections between the screens the user will see.
5. Plan the logic.
6. Code the program.
7. Translate the program into machine language.
8. Test the program.
9. Put the program into production.

The three new steps involve elements of object-oriented, GUI design—creating storyboards, defining objects, and defining the connections between user screens. As with procedural programming, you cannot write an event-driven program unless you first understand the problem.

UNDERSTANDING THE PROBLEM

Suppose you want to create a simple, interactive program that determines premiums for prospective insurance customers. The users should be able to use a graphical interface to select a policy type—health or auto. Next, the users answer pertinent questions, such as how old they are, whether they smoke, and what their driving records are like. Although most insurance premium amounts would be based on more characteristics than these, assume that policy rates are determined using the factors shown in Table 14-3. The final output of the program is a second screen that shows the semiannual premium amount for the chosen policy.

TABLE 14-3: INSURANCE PREMIUMS BASED ON CUSTOMER CHARACTERISTICS

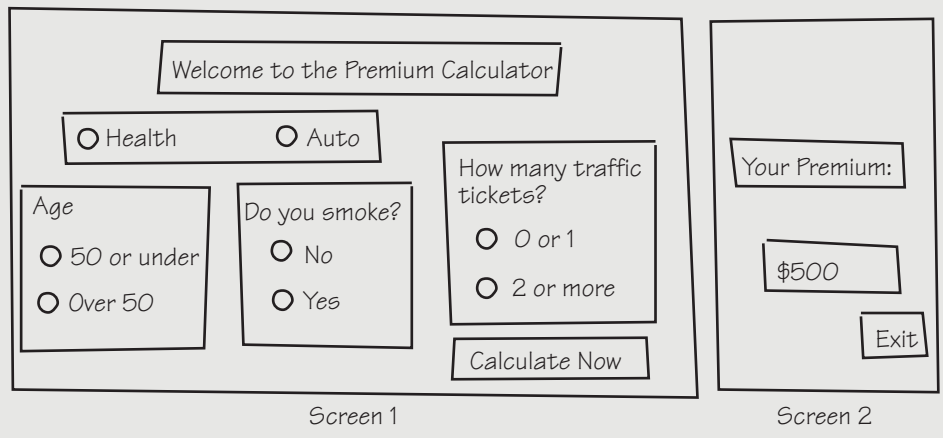
Health policy premiums	Auto policy premiums
Base rate: $500	Base rate: $750
Add $100 if over age 50	Add $400 if more than 2 tickets
Add $250 if smoker	Subtract $200 if over age 50

CREATING STORYBOARDS

A **storyboard** represents a picture or sketch of a screen the user will see when running a program. Filmmakers have long used storyboards to illustrate key moments in the plots they are developing; similarly, GUI storyboards represent "snapshot" views of the screens the user will encounter during the run of a program. If the user could view up to four screens during the insurance premium program, then you would draw four storyboard cells, or frames.

Figure 14-2 shows two storyboard sketches for the insurance program. They represent the introductory screen at which the user selects a premium type and answers questions, and the final screen that displays the semiannual premium.

FIGURE 14-2: STORYBOARD FOR INSURANCE PREMIUM PROGRAM



DEFINING THE OBJECTS IN AN OBJECT DICTIONARY

An event-driven program may contain dozens, or even hundreds, of objects. To keep track of them, programmers often use an object dictionary. An **object dictionary** is a list of the objects used in a program, including which screens they are used on and whether any code or script is associated with them.

Figure 14-3 shows an object dictionary for the insurance premium program. The type and name of each object to be placed on a screen is listed in the left column. The second column shows the screen number on which the object appears. The next column names any variables that are affected by an action on the object. The right column indicates whether any code or script is associated with the object. For example, the label named `welcomeLabel` appears on the first screen. It has no associated actions—it does not call any methods or change any variables; it is just a label. The `calculateButton`, however, does cause execution of a method named `calcRoutine()`. This method calculates the semiannual premium amount and stores it in the `premiumAmount` variable. Depending on the programming language you use, you might need to name `calcRoutine()` something similar to `calculateButton.click()` to identify it as the module that executes when the user clicks the `calculateButton`.

TIP Some organizations also include the disk location where an object is stored as part of the object dictionary.

FIGURE 14-3: OBJECT DICTIONARY FOR INSURANCE PREMIUM PROGRAM

Object type	Object name	Screen number	Variables affected	Script?
Label	welcomeLabel	1	none	none
Choice	healthOrAuto	1	policyType	none
Choice	age	1	ageOfInsured	none
Choice	smoker	1	insuredIsSmoker	none
Choice	tickets	1	numTickets	none
Button	calculateButton	1	premiumAmount	calcRoutine()
Label	yourPremium	2	none	none
Text field	premAmtField	2	none	none
Button	exitButton	2	none	exitRoutine()

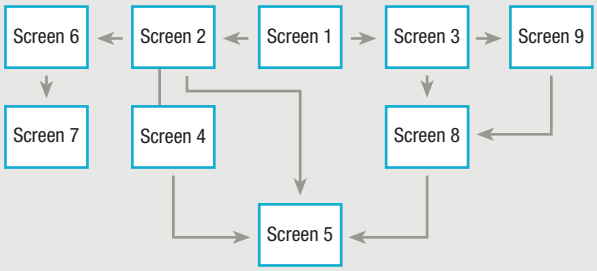
DEFINING THE CONNECTIONS BETWEEN THE USER SCREENS

The insurance premium program is a small one, but with larger programs you may need to draw the connections between the screens to show how they interact. Figure 14-4 shows an interactivity diagram for the screens used in the insurance premium program. An **interactivity diagram** shows the relationship between screens in an interactive GUI program. Figure 14-4 shows that the first screen calls the second screen, and the program ends.

FIGURE 14-4: DIAGRAM OF INTERACTION FOR INSURANCE PREMIUM PROGRAM

Figure 14-5 shows how a diagram might look for a more complicated program in which the user has several options available at screens 1, 2, and 3. Notice how each screen may lead to different screens, depending on the options the user selects at any one screen.

FIGURE 14-5: DIAGRAM OF INTERACTION FOR A HYPOTHETICAL COMPLICATED PROGRAM



PLANNING THE LOGIC

In an event-driven program, you design the screens, define the objects, and define how the screens will connect. Then, you can plan the logic for each of the modules (or methods or scripts) that the program will use. For example, given the program requirements shown in Table 14-3, you can write the pseudocode for the `calcRoutine()` method of the insurance premium program, as shown in Figure 14-6. The `calcRoutine()` method does not execute until the user clicks the `calculateButton`. At that point, the user's choices are used to calculate the premium amount.

In Figure 14-6, assume that the user's data variables, such as `policyType` and `ageOfInsured`, have been correctly defined and passed to the method. Their definitions are not included here to keep the example simple.

The pseudocode in Figure 14-6 should look very familiar to you—it declares constants and uses decision-making logic you have used since the early chapters of this book. Everything you have learned about variables and constants, your comfort with the basic structures of sequence, selection, and looping, and all you know about methods and arrays will continue to serve you well, whether you are programming in a procedural or event-driven environment.

TIP

With some object-oriented programming languages, you must **register**, or sign up, components that will react to events initiated by other components. The details of how this is accomplished vary among languages.

FIGURE 14-6: PSEUDOCODE FOR `calcRoutine()`

```
calcRoutine()
   const char HEALTH = "H"
   const char YES = "Y"
   const num BASE_PREMIUM_HEALTH = 500
   const num AGE_CUTOFF = 50
   const num AGE_HEALTH_EXTRA = 100
   const num SMOKER_EXTRA = 250
   const num BASE_PREMIUM_AUTO = 750
   const num TICKET_CUTOFF = 2
   const num TICKET_EXTRA = 400
   const num AGE_AUTO_DISCOUNT = -200
   if policyType = HEALTH then
      premiumAmount = BASE_PREMIUM_HEALTH
      if ageOfInsured > AGE_CUTOFF then
          premiumAmount = premiumAmount + AGE_HEALTH_EXTRA
      endif
      if insuredIsSmoker = YES then
          premiumAmount = premiumAmount + SMOKER_EXTRA
      endif
   else
      premiumAmount = BASE_PREMIUM_AUTO
      if numTickets > TICKET_CUTOFF then
          premiumAmount = premiumAmount + TICKET_EXTRA
      endif
      if ageOfInsured > AGE_CUTOFF then
          premiumAmount = premiumAmount + AGE_AUTO_DISCOUNT
      endif
   endif
return
```

UNDERSTANDING THE DISADVANTAGES OF TRADITIONAL ERROR-HANDLING TECHNIQUES

A great deal of the effort that goes into writing programs involves checking data items to make sure they are valid and reasonable. A great advantage of using GUI data-entry objects is that you often can control much of what a user enters by limiting the user's options. When you provide a user with a finite set of buttons to click, or a limited number of menu items from which to choose, the user does not have the opportunity to make unexpected, illegal, or bizarre choices. For example, if you provide a user with only two buttons, so that the only insurance policy types the user can select are *Health* or *Auto*, then you can eliminate checking for a valid policy type within your interactive program.

TIP In Chapter 10, you learned the phrase programmers use to describe worthless or invalid input: "GIGO."

Not all user entries are limited to a finite number of possibilities, however; there are many occasions on which you must allow the user to enter data that you cannot validate—for example, names and addresses. Professional data-entry operators who create the files used in business computer applications (for example, typing data from phone or mail orders) spend their entire working day entering facts and figures that your applications use; operators can and do make

typing errors. When programs depend on data entered by average users who are not trained typists, the chance of error is even more likely. In Chapter 10, you learned some useful techniques to check for valid and reasonable input data.

Programmers had to deal with error conditions long before object-oriented methods were conceived. Probably the most often used error-handling method was to terminate the program, or at least the module in which the offending statement occurred. For example, Figure 14-7 shows a segment of pseudocode that causes the insurance premium `calcRoutine()` module to end if `policyType` is invalid; in the shaded `if` statement, the module ends abruptly when `policyType` is not "A" or "H". Not only is this method of handling an error unforgiving, it isn't even structured. Recall that a structured module should have one entry and one exit point. The module in Figure 14-7 contains two exit points at the two `return` statements.

FIGURE 14-7: UNFORGIVING, UNSTRUCTURED METHOD OF ERROR HANDLING

```
calcRoutine()
   const char HEALTH = "H"
   const char AUTO = "A"
   const char YES = "Y"
   const num BASE_PREMIUM_HEALTH = 500
   const num AGE_CUTOFF = 50
   const num AGE_HEALTH_EXTRA = 100
   const num SMOKER_EXTRA = 250
   const num BASE_PREMIUM_AUTO = 750
   const num TICKET_CUTOFF = 2
   const num TICKET_EXTRA = 400
   const num AGE_AUTO_DISCOUNT = -200
   if policyType not = HEALTH AND policyType not = AUTO then
      return
   else
      if policyType = HEALTH then
         premiumAmount = BASE_PREMIUM_HEALTH
         if ageOfInsured > AGE_CUTOFF then
             premiumAmount = premiumAmount + AGE_HEALTH_EXTRA
         endif
         if insuredIsSmoker = YES then
             premiumAmount = premiumAmount + SMOKER_EXTRA
         endif
      else
         premiumAmount = BASE_PREMIUM_AUTO
         if numTickets > TICKET_CUTOFF then
             premiumAmount = premiumAmount + TICKET_EXTRA
         endif
         if ageOfInsured > AGE_CUTOFF then
             premiumAmount = premiumAmount + AGE_AUTO_DISCOUNT
         endif
      endif
   endif
return
```

In the example in Figure 14-7, if `policyType` is an invalid value, the module in which the code appears is terminated. If the program that contains this module is part of a business program or a game, the user may be annoyed that the program has stopped working and that an early exit has been made. However, an early exit in a program that monitors a hospital patient's vital signs or navigates an airplane might cause results that are far more serious.

Rather than ending a method prematurely just because it encounters a piece of invalid data, a more elegant solution involves creating an error flag variable and looping until the data item becomes valid, as shown in the highlighted portion of Figure 14-8. The flag variable `errorFlag` is set to 1 at the beginning of the module, so that the `while` loop statements (beginning with the first highlighted line in the figure) will execute at least once. You enter the loop, and if `policyType` is valid, you set `errorFlag` to 0 (the second highlighted line) so the loop will not repeat. If `policyType` is "A" or "H", the appropriate auto or health calculations and assignments are made. Otherwise, you prompt the user to reenter the policy type and set `errorFlag` to 1 (the last highlighted statement in the figure). This forces the loop to repeat. The new user entry is checked and the loop repeats until an "A" or "H" is entered.

FIGURE 14-8: USING A LOOP TO HANDLE INTERACTIVE ERRORS

```
calcRoutine()
   const char HEALTH = "H"
   const char AUTO = "A"
   const char YES = "Y"
   const num BASE_PREMIUM_HEALTH = 500
   const num AGE_CUTOFF = 50
   const num AGE_HEALTH_EXTRA = 100
   const num SMOKER_EXTRA = 250
   const num BASE_PREMIUM_AUTO = 750
   const num TICKET_CUTOFF = 2
   const num TICKET_EXTRA = 400
   const num AGE_AUTO_DISCOUNT = -200
   num errorFlag = 1
   while errorFlag = 1
      errorFlag = 0
      if policyType = HEALTH then
         premiumAmount = BASE_PREMIUM_HEALTH
         if ageOfInsured > AGE_CUTOFF then
             premiumAmount = premiumAmount + AGE_HEALTH_EXTRA
         endif
         if insuredIsSmoker = YES then
             premiumAmount = premiumAmount + SMOKER_EXTRA
         endif
      else
         if policyType = AUTO then
            premiumAmount = BASE_PREMIUM_AUTO
            if numTickets > TICKET_CUTOFF then
               premiumAmount = premiumAmount + TICKET_EXTRA
            endif
            if ageOfInsured > AGE_CUTOFF then
               premiumAmount = premiumAmount + AGE_AUTO_DISCOUNT
            endif
         else
            print "Invalid policy type. Please reenter"
            read policyType
            errorFlag = 1
         endif
      endif
   endwhile
return
```

There are at least two shortcomings to the error-handling logic shown in Figure 14-8. First, the module is not as reusable as it could be, and second, it is not as flexible as it might be.

One of the principles of modular and object-oriented programming is reusability. The module in Figure 14-8 is only reusable under limited conditions. The `calcRoutine()` module 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 `calcRoutine()` would simply calculate the premium amount without deciding what to do about data errors. The `calcRoutine()` method will be most flexible if it can detect an error and then notify the calling program or module that an error has occurred. Each program or module that uses the `calcRoutine()` module then can determine how it should best handle the mistake.

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 in an interactive program; if stored data is invalid, it remains invalid. Object-oriented exception-handling techniques overcome the limitations of simply repeating a request.

UNDERSTANDING THE ADVANTAGES OF OBJECT-ORIENTED EXCEPTION HANDLING

Object-oriented, event-driven programs employ a more specific group of methods for handling errors called **exception-handling methods**. The methods check for and manage errors. The generic name used for errors in object-oriented languages is **exceptions** because, presumably, errors are not usual occurrences; they are the "exceptions" to the rule.

In object-oriented terminology, an exception is an object that represents an error. You **try** a module that might throw an exception. The module might **throw**, or pass, an exception out, and the original calling module can then **catch**, or receive, the exception and handle the problem. The exception object that is thrown can be any data type—a numeric or character data item or a programmer-created object such as a record complete with its own data fields and methods. For example, Figure 14-9 shows a `calcRoutine()` module that throws an `errorFlag` only if `policyType` is neither "H" nor "A". If `policyType` is "H" or "A", the premium is calculated and the module ends naturally, but if neither of the valid codes is stored in `policyType`, the highlighted `throw` statement executes. The module in Figure 14-9 might be used within the program segment shown in Figure 14-10.

TIP In Figure 14-9, `errorFlag` could also be declared a constant, because it never changes.

FIGURE 14-9: THROWING AN EXCEPTION

```
calcRoutine()
   const char HEALTH = "H"
   const char AUTO = "A"
   const char YES = "Y"
   const num BASE_PREMIUM_HEALTH = 500
   const num AGE_CUTOFF = 50
   const num AGE_HEALTH_EXTRA = 100
   const num SMOKER_EXTRA = 250
   const num BASE_PREMIUM_AUTO = 750
   const num TICKET_CUTOFF = 2
   const num TICKET_EXTRA = 400
   const num AGE_AUTO_DISCOUNT = -200
   num errorFlag = 1
   if policyType = HEALTH then
       premiumAmount = BASE_PREMIUM_HEALTH
       if ageOfInsured > AGE_CUTOFF then
           premiumAmount = premiumAmount + AGE_HEALTH_EXTRA
       endif
       if insuredIsSmoker = YES then
           premiumAmount = premiumAmount + SMOKER_EXTRA
       endif
   else
      if policyType = AUTO then
         premiumAmount = BASE_PREMIUM_AUTO
         if numTickets > TICKET_CUTOFF then
             premiumAmount = premiumAmount + TICKET_EXTRA
         endif
         if ageOfInsured > AGE_CUTOFF then
             premiumAmount = premiumAmount + AGE_AUTO_DISCOUNT
         endif
      else
         throw errorFlag
      endif
   endif
return
```

FIGURE 14-10: PROGRAM SEGMENT USING `calcRoutine()`

```
try
   perform calcRoutine()
endTry
catch(num thrownCode)
    policyType = "H"
    try
       perform calcRoutine()
    endTry
endCatch
// Program continues
```

In the program segment in Figure 14-10, the call to `calcRoutine()` is placed in a **`try` block**, a segment of code in which an attempt is made to execute the module. If `calcRoutine()` encounters an error and throws an exception,

the highlighted `catch` block in Figure 14-10 will catch it. A **`catch` block** contains code that executes when an exception is thrown from within a `try` block. The `catch` block contains a variable in parentheses that is the same data type as the object thrown from the `calcRoutine()` module; in this case the thrown object is numeric. In the program segment in Figure 14-10, the `catch` block has been written to force the premium type to a health policy, and `calcRoutine()` is attempted again. This time, execution of the `calcRoutine()` module will be successful because `policyType` is guaranteed to be valid.

During any execution of the `calcRoutine()` module, if `policyType` is valid, no exception is thrown, the module continues to calculate the policy premium, and when the module returns, the calling program skips the `catch` block and continues with any code following it. In other words, the `catch` block in the calling module executes only when an exception is thrown, and only when the thrown exception is the data type that the `catch` block has been programmed to accept.

Because the `calcRoutine()` module throws an exception, a different program can use it and handle the exception more appropriately for that application. For example, in an application in which an invalid policy type should not be forced to a health policy but should be reentered by the user, the exception can be handled as shown in Figure 14-11. In this figure, a variable named `thrownCode` is set to 1. This ensures that the `while` loop that follows will execute at least once. Then, `calcRoutine()` executes, as shown in Figure 14-9.

FIGURE 14-11: ALTERNATE PROGRAM SEGMENT USING `calcRoutine()`

```
num thrownCode = 1
while thrownCode = 1
   try
      perform calcRoutine()
      thrownCode = 0
   endTry
   catch(num thrownCode)
      print "Reenter the policy type"
      read policyType
   endCatch
endwhile
// Program continues
```

When the `calcRoutine()` module is used with the program segment in Figure 14-11, there are two possible outcomes:

- An error occurs and an exception is thrown. When the `calcRoutine()` module throws the `errorFlag` that has a value of 1, the `calcRoutine()` module ends immediately, and the program segment in Figure 14-11 bypasses all statements following the method call `perform calcRoutine()` and proceeds directly to the `catch` block. This means that the statement `thrownCode = 0` is skipped, and the value of the `thrownCode` variable remains 1. In the `catch` statement, the program catches the value thrown from the `calcRoutine()` module (and stores it in the `thrownCode` variable that is declared in the `catch` statement). The two statements `print "Reenter the policy type"` and `read policyType` execute. Then, when the `catch` block ends (indicated by the `endCatch` statement in the pseudocode), `thrownCode` is still 1, because the statement setting it to 0 was bypassed. The `while` loop executes again, and `calcRoutine()` is attempted again.

- No error occurs and no exception is thrown. When `calcRoutine()` is successful (that is, if `policyType` is valid), nothing is thrown from the `calcRoutine()` module, so in the program segment in Figure 14-11, the logic proceeds to the statement following `perform calcRoutine()`, the statement that sets `thrownCode` to 0. The `catch` block does not execute, and the 0 code stops the loop from executing again. In other words, when nothing is thrown from the `calcRoutine()` module, the code becomes 0, the `catch` block is bypassed, and the program proceeds to the `endwhile` statement; `thrownCode` is no longer 1, and the loop ends.

TIP

Programmers sometimes refer to the situation where nothing goes wrong as the **sunny day case**.

TIP

You declare `thrownCode` as a numeric variable in the `catch` block in much the same way as you declare a passed variable in a method header—by providing the variable with a type and a name. When `calcRoutine()` throws its `errorFlag` variable, its value becomes known as `thrownCode` within the `catch` block. You could provide the thrown variable with any legal identifier within the `catch` block. You also could use `thrownCode` like any other variable, perhaps displaying it or making a decision based on its value.

The program segment in Figure 14-11 is difficult to follow if you are new to exception-handling techniques. You can see the flexibility of using thrown exceptions when you consider the program segments in Figures 14-12 and 14-13. Like the program segment in Figure 14-11, the one in Figure 14-12 also uses `calcRoutine()`, but this module does not allow the user to reenter the `policyType` value. If `calcRoutine()` throws a value, the `catch` block executes, printing a message that includes the error code. This logic would be even more useful if several error code values were possible. The user could analyze the message and take appropriate action.

FIGURE 14-12: PROGRAM SEGMENT THAT DISPLAYS THROWN ERROR CODE

```
try
  perform calcRoutine()
endTry
catch(num thrownCode)
      print "Error #", thrownCode," has occurred"
endCatch
// Program continues
```

FIGURE 14-13: PROGRAM SEGMENT THAT SETS `premiumAmount` TO 0 WHEN EXCEPTION IS THROWN

```
try
  perform calcRoutine()
endTry
catch(num thrownCode)
      premiumAmount = 0
endCatch
// Program continues
```

The program segment in Figure 14-13 also uses the same `calcRoutine()` method. This program segment simply assumes that the premium amount is 0 for invalid policy types. When the `catch` block in Figure 14-13 executes, the program doesn't use the value that is thrown. The fact that a value *is* thrown is all that is required to cause the `catch` block to execute; although an identifier must be provided for the thrown value, the value that is caught does not have to be used in any way.

The general principle of exception handling in object-oriented programming is that a module 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 module appropriately.

TIP

In most object-oriented programming languages, a module can throw any number of exceptions, with one restriction—there must be a `catch` block available for each type of exception. In other words, a module might throw a numeric variable under one error condition, a character variable under another, and a complex class object under a third. When an exception is thrown, only the matching `catch` block executes. Many languages provide a generic type you can use in a `catch` block so that it can catch anything that is thrown.

CHAPTER SUMMARY

- Interacting with a computer operating system from the command line is difficult; it is easier to use an event-driven graphical user interface (GUI), in which users manipulate objects such as buttons and menus. Within an event-driven program, a component from which an event is generated is the source of the event. A listener is an object that is "interested in" an event to which you want it to respond.
- The possible events a user can initiate include a key press, mouse point, click, right-click, double-click, and drag. Common GUI components include labels, text fields, buttons, check boxes, option buttons, list boxes, and toolbars. GUI components are excellent examples of the best principles of object-oriented programming—they represent objects with attributes and methods that operate like black boxes.
- When you create a program that uses a GUI, the interface should be natural, predictable, attractive, easy to read, and nondistracting. It's helpful if the user can customize your applications. The program should be forgiving, and you should not forget that the GUI is only a means to an end.
- You can modify the attributes of GUI components. For example, you can set the size, color, screen location, font, visibility, and enabled status of the component.
- Developing an event-driven application is more complicated than developing a standard procedural program. You must understand the problem, create storyboards, define the objects, define the connections between the screens the user will see, plan the logic, code the program, translate the program into machine language, test the program, and put the program into production.
- Traditional error-handling methods have limitations.
- Object-oriented error handling involves throwing exceptions. An exception is an object that you throw from the module where a problem occurs to another module that will catch it and handle the problem. The general principle of exception handling in object-oriented programming is that a module 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 module appropriately.

KEY TERMS

The **command line** is the location on your computer screen at which you type entries to communicate with the computer's operating system.

An **operating system** is the software that you use to run a computer and manage its resources.

The command line also is called the **command prompt**.

The **DOS prompt** is the command line in the DOS operating system.

Icons are small pictures on the screen that the user can select with a mouse.

A **graphical user interface**, or **GUI**, allows users to interact with an operating system by clicking icons to select options.

An **event** is an occurrence that generates a message sent to an object.

GUI programs are called **event-based** or **event-driven** because actions occur in response to user-initiated events such as clicking a mouse button.

A component from which an event is generated is the **source** of the event.

A **listener** is an object that is "interested in" an event to which you want it to respond.

A disabled component is identified by **dimming** or **graying** it—that is, by making its appearance muted or softer.

The screen design issues that make programs easier to use for people with physical limitations are known as **accessibility** issues.

A **storyboard** represents a picture or sketch of a screen the user will see when running a program.

An **object dictionary** is a list of the objects used in a program, including which screens they are used on and whether any code, or script, is associated with them.

An **interactivity diagram** shows the relationship between screens in an interactive GUI program.

In some object-oriented programming languages, you **register**, or sign up, components that will react to events initiated by other components.

Object-oriented, event-driven programs employ a group of error-handling methods called **exception-handling methods**.

The generic name used for errors in object-oriented languages is **exceptions** because, presumably, errors are not usual occurrences; they are the "exceptions" to the rule.

You **try** a module that might throw an exception.

You **throw**, or pass, an exception from the module where a problem occurs to another module.

A module that receives an exception and handles a problem **catches** the exception.

A **`try` block** is a segment of code in which an attempt is made to execute a module that might throw an exception.

A **`catch` block** is a group of statements that execute when a value is caught.

The **sunny day case** is the case in which no errors occur.

REVIEW QUESTIONS

1. **As opposed to using a command line, an advantage to using an operating system that employs a graphical user interface is _____.**
   a. you can interact directly with the operating system
   b. you do not have to deal with confusing icons
   c. you do not have to memorize complicated commands
   d. all of the above

2. **When users can initiate actions by clicking a mouse on an icon, the program is said to be _____-driven.**
   a. event
   b. prompt
   c. command
   d. incident

3. **A component from which an event is generated is the _____ of the event.**
   a. base
   b. icon
   c. listener
   d. source

4. **An object that responds to an event is a _____.**
   a. source
   b. listener
   c. transponder
   d. snooper

5. **All of the following are user-initiated events except a _____.**
   a. key press
   b. key drag
   c. right mouse click
   d. mouse drag

6. **All of the following are typical GUI components except a _____.**
   a. label
   b. text field
   c. list box
   d. button box

7. **GUI components operate like _____.**
   a. black boxes
   b. procedural functions
   c. looping structures
   d. command lines

8. Which is not a principle of good GUI design?

a. The interface should be predictable.
b. The fancier the screen design, the better.
c. The program should be forgiving.
d. The user should be able to customize your applications.

9. Which of the following aspects of a GUI layout is most predictable and natural for the user?

a. A menu bar appears running down the right side of the screen.
b. Help is the first option on a menu.
c. Saving a file is represented by a dollar sign icon.
d. Pressing Esc allows the user to cancel a selection.

10. In most GUI programming environments, which of the following component attributes cannot be changed?

a. color
b. screen location
c. size
d. You can change all of these attributes.

11. Depending on the programming language you use, you might _____ to change a screen component's attributes.

a. use an assignment statement
b. call a module
c. enter a value into a list of properties
d. all of the above

12. When you create an event-driven application, which of the following must be done before defining the objects you will use?

a. Plan the logic.
b. Create storyboards.
c. Test the program.
d. Code the program.

13. A _____ is a sketch of a screen the user will see when running a program.

a. flowchart
b. hierarchy chart
c. storyboard
d. UML diagram

14. A list of objects used in a program is an object _____.

a. thesaurus
b. glossary
c. index
d. dictionary

15. A(n) _____ diagram shows the connections between the various screens a user might see during a program's execution.

a. interactivity
b. help
c. cooperation
d. communication

16. In object-oriented programs, errors are known as _____.

a. faults
b. gaffes
c. exceptions
d. omissions

17. When an object-oriented program detects an error, it _____ it.

a. absorbs
b. throws
c. displays
d. records

18. In object-oriented programs, if a module calls another that generates an exception, then the first module _____ it.

a. catches
b. destroys
c. records
d. throws

19. An exception can be a _____.

a. number
b. character
c. user-defined type
d. any of the above

20. The general principle of exception handling in object-oriented programming is that a module 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

FIND THE BUGS

Each of the following pseudocode segments contains one or more bugs that you must find and correct.

1. **In the following code, the main program tries the `dataEntryRoutine()` module, which prompts the user for an item the user is ordering. If the item is valid, the price is returned from the function. Otherwise, an exception is thrown and the main program displays an error message and sets the price to 0. Finally, the main program calculates the tax (5 percent of the price) and displays the customer's total.**

```
num dataEntryRoutine()
   const num SZ = 5;
   const num itemsForSale[SZ] = {111, 22, 333, 444, 555}
   const num itemsPrice[2] = {2.34, 5.67, 12.75, 15.00, 21.00}
   num x
   num orderedItem
   num price
   char found
   print "Enter item to order"
   read orderedItem
   found = "N"
   while found < "N" AND x < SZ
     if orderedItem = itemsForSale[SZ] then
          price = itemsPrice[SZ]
          found = "N"
     endif
     x = x * 1
   endwhile
   if found = "N" then
      throw orderedItem
   endif
return price

start
   num price
   num total
   try
     price = dataEntryRoutine()
   endTry
   catch(num orderedItem)
      print "You tried to order item number ", item,
        "which is not a valid item number"
      price = 0
   endCatch
   total = price + price * TAX
   print "Your total is $", sum
stop
```

2. The following main program represents a different application that can use the same `dataEntryRoutine()` module as in the previous exercise. Instead of forcing the user's price to 0 when an exception is thrown, this application requires the user to reenter the order until a valid item is ordered.

```
start
   num price
   num total
   num item
   num okFlag
   while okFlag = 0
     try
        price = dataEntryRoutine()
        okFlag = 1
     catch(num orderedItem)
        print "You tried to order item number ", orderedItem,
          "which is not a valid item number"
        print "Please reenter the item number"
     endTry
     endCatch
   total = pr + pr * TAX
   print "Your total is $", total
stop
```

EXERCISES

1. **Take a critical look at three GUI applications with which you are familiar—for example, a spreadsheet, a word-processing program, and a game. Describe how well each conforms to the GUI design guidelines listed in this chapter.**

2. **Select one element of poor GUI design in a program with which you are familiar. Describe how you would improve the design.**

3. **Select a GUI program that you have never used before. Describe how well it conforms to the GUI design guidelines listed in this chapter.**

4. **Design the storyboards, interactivity diagram, object dictionary, and any necessary methods for an interactive program for customers of Sunflower Floral Designs. Allow customers the option of choosing a floral arrangement ($25 base price), cut flowers ($15 base price), or a corsage ($10 base price). Let the customer choose roses, daisies, chrysanthemums, or irises as the dominant flower. If the customer chooses roses, add $5 to the base price. After the customer clicks an "Order Now" button, display the price of the order.**

5. **Design the storyboards, interactivity diagram, object dictionary, and any necessary methods for an interactive program for customers of Toby's Travels. Allow customers to choose from at least five trip destinations and four means of transportation, each with a unique price. After the customer clicks the "Plan Trip Now" button, display the price of the trip.**

6. **Complete the following tasks:**
   a. Design a method that calculates the cost of a painting job for College Student Painters. Variables include whether the job is location "I" for interior, which carries a base price of $100, or "E" for exterior, which carries a base price of $200. College Student Painters charges an additional $5 per square foot over the base price. The method should throw an exception if the location code is invalid.
   b. Write a module that calls the module designed in Exercise 6a. If the module throws an exception, force the price of the job to 0.
   c. Write a module that calls the module designed in Exercise 6a. If the module throws an exception, require the user to reenter the location code.
   d. Write a module that calls the module designed in Exercise 6a. If the module throws an exception, force the location code to "E" and the base price to $200.

7. **Design the storyboards, interactivity diagram, object dictionary, and any necessary methods for an interactive program for customers of The Mane Event Hair Salon. Allow customers the option of choosing a haircut ($15), coloring ($25), or perm ($45). After the customer clicks a "Select" button, display the price of the service.**

8. Complete the following tasks:

a. Design a method that calculates the cost of a semester's tuition for a college student at Mid-State University. Variables include whether the student is an in-state resident ("I" for in-state or "O" for out-of-state) and the number of credit hours for which the student is enrolling. The method should throw an exception if the residency code is invalid. Tuition is $75 per credit hour for in-state students and $125 per credit hour for out-of-state students. If a student enrolls in six hours or fewer, there is an additional $100 surcharge. Any student enrolled in 19 hours or more pays only the rate for 18 credit hours.

b. Write a module that calls the module designed in Exercise 8a. If the module throws an exception, force the tuition to 0.

9. Complete the following tasks:

a. Design a method that validates a date. The date is constructed from three numeric variables representing month, day, and year. The method should throw an exception if the date is invalid—for example, if the month does not fall between 1 and 12 inclusive, or if the day does not fall within the legal days for a given month—for example, 4/31. Assume that any year is acceptable, and assume that 2/29 is always a valid date.

b. Write a module that prompts the user to enter a date, and then calls the module designed in Exercise 9a. If the module throws an exception, force the month, day, and year to 0.

c. Write a different module that reads an employee record from a data file. The employee record contains fields for name; month, day, and year of birth; and month, day, and year of hire. Call the module designed in Exercise 9a. If the module throws an exception, force the birth date variables to 99 and force the hire date variables to today's date.

d. Write another module that prompts the user to enter a date, and then calls the module designed in Exercise 9a. If the module throws an exception, display an error message and continue to prompt the user until no exception is thrown.

DETECTIVE WORK

1. Find out what you can about the Visual Basic programming language. What are its origins? Describe the interface with which programmers work in this language. For what types of applications is Visual Basic most often used?
2. Is there a gender gap in programming? Is it different for traditional Web programming and GUI Web-based programming?

UP FOR DISCUSSION

1. This chapter discusses "unnatural" design and mentions typewriter keyboards and stovetops as examples. What other day-to-day objects are unnatural to use?
2. When people use interactive programs on the Web, when is it appropriate to track which buttons or other icons they select or to record the data they enter? When is it not appropriate? Does it matter how long the data is stored? Does it matter if a profit is made from using the data?