Chapter 12

Event-Driven GUI Programming, Multithreading, and Animation

- Chapter Introduction
- 12-1 Understanding Event-Driven Programming
- 12-2 User-Initiated Actions and GUI Components
- 12-3 Designing Graphical User Interfaces
  - 12-3a The Interface Should Be Natural and Predictable
  - 12-3b The Interface Should Be Attractive, Easy to Read, and Nondistracting
  - 12-3c To Some Extent, It's Helpful If the User Can Customize Your Applications
  - 12-3d The Program Should Be Forgiving
  - 12-3e The GUI Is Only a Means to an End
- 12-4 Developing an Event-Driven Application
  - 12-4a Creating Storyboards
  - 12-4b Defining the Storyboard Objects in an Object Dictionary
  - 12-4c Defining Connections Between the User Screens
  - 12-4d Planning the Logic
- 12-5 Understanding Threads and Multithreading
- 12-6 Creating Animation
- 12-7 Chapter Review
  - 12-7a Chapter Summary
  - 12-7b Key Terms
  - 12-7c Review Questions
  - 12-7d Exercises

Chapter Introduction

In this chapter, you will learn about:

- The principles of event-driven programming
- User-initiated actions and GUI components
- Designing graphical user interfaces
- Developing an event-driven application
- Threads and multithreading
- Creating animation

12-1 Understanding Event-Driven Programming

From the 1950s, when businesses began to use computers, through the 1980s, almost all interactive dialogues between people and computers took place at the command prompt. (Programmers also call the command prompt the *command line*, and users of the Disk Operating System often call the command line the **DOS prompt** (The command line in the DOS operating system.) .) In Chapter 1, you learned that the command line is used to type entries to communicate with the computer's **operating system** (The software that runs a computer and manages its resources.) —the software that you use to run a computer and manage its resources. In the early days of computing, interacting with an operating system was difficult because users had to know the exact syntax to use when typing commands, and they had to spell and type those commands accurately. (Syntax is the correct sequence of words and symbols that form the operating system's command set.) Figure 12-1 shows a command in the Windows operating system.

Figure 12-1

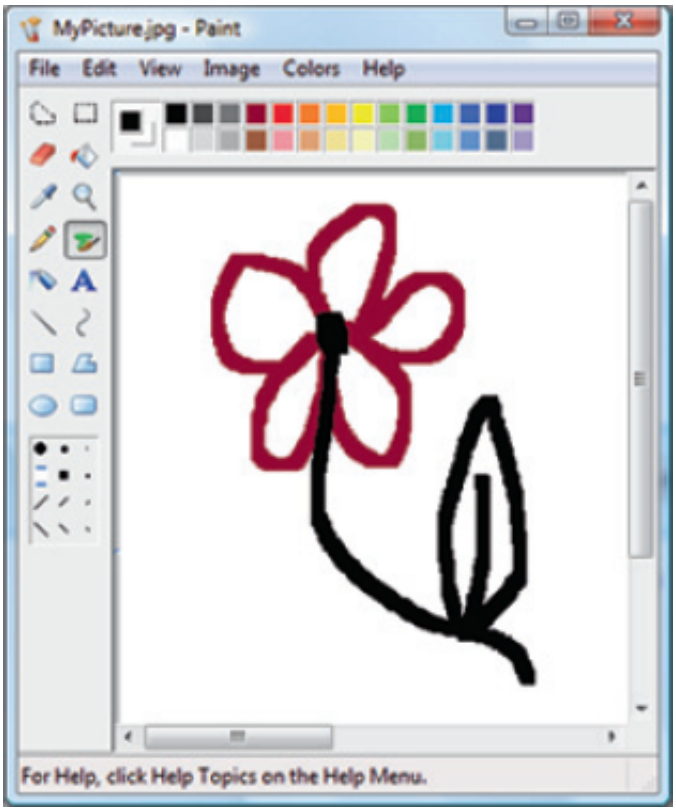
Command Prompt Screen

If you use the Windows operating system on a PC, you can locate the command prompt by clicking Start and pointing to the command prompt window shortcut on the Start menu. Alternatively, you can point to All Programs (or Programs in some earlier operating systems), then Accessories, and then click Command Prompt.

Fortunately for today's computer users, operating system software allows them to use a mouse or other pointing device to select pictures, or **icons** (Small pictures on a screen that a user can select with a mouse.) , on the screen. As you learned in Chapter 1, this type of environment is a graphical user interface, or GUI. Computer users can expect to see a standard interface in GUI programs. 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 a recycling bin icon to delete the files in the folder. Figure 12-2 shows a Windows program named Paint in which icons representing pencils, paint cans, and other objects appear on clickable buttons.

Figure 12-2

A GUI Application that Contains Buttons and Icons



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.) —an occurrence that generates a message sent to an object. GUI programs frequently are called **event-driven or event-based** (Describes programs and actions that occur in response to user-initiated events such as clicking a mouse button.) 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 objects that users can manipulate, such as text boxes, buttons, and menus, and on events that users can initiate with those objects, such as typing, pointing, clicking, or double-clicking. The programmer writes instructions within modules that execute in response to each type of event.

Starting in Chapter 1, and throughout the rest of this book so far, the program logic you have developed has been procedural, and not event-driven; each step occurs in the order the programmer determines. In a procedural application, if you write statements that display a prompt and accept a user's response, the processing goes no further until the input is completed. When you write a procedural program and `call moduleA()` before calling `moduleB()`, you have complete control over the order in which all the statements will execute.

In contrast, with most event-driven programs, the user might initiate any number of events in any order. For example, when 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 italics, choose a menu item such as *Save* or *Print*, and so on. With each word-processing document you create, the program must be ready to respond to any event you initiate. The programmers who created the word processor are not guaranteed that you will select *Bold* before you select *Italics*, or that you will select Save before you select *Quit*, so they must write programs that are more flexible than their procedural counterparts.

Within an event-driven program, a component from which an event is generated is the **source of the event** (The component from which an event is generated.) . A button that users can click to cause an action is an example of a source; a text box in which users enter typed characters is another source. An object that is "interested in" an event to which you want it to respond is a **listener** (An object that is prepared to respond to events from specified sources.) . 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 many areas of the screen has no effect. If you want an object such as a button to be a listener for an event such as a mouse click, you must write two types of appropriate program statements. You write the statements that define the object as a listener and the statements that constitute the event.

Although event-driven programming is newer than procedural programming, the instructions that programmers write to respond to events are still simply sequences, selections, and loops. Event-driven programs still have methods that declare variables, use arrays, and contain all the attributes of their procedural-program ancestors. The user's screen in an event-driven program might contain buttons or check boxes with labels like *Sort Records*, *Merge Files*, or *Total Transactions*, but each of these processes represents a method that uses the same logic you have learned throughout this book for programs that did not have a graphical interface. In object-oriented languages, the procedural modules that depend on user-initiated events are often called **scripts** (A procedural module that

depends on user-initiated events in object-oriented programs.) . Writing event-driven programs involves thinking of possible events, writing scripts to execute actions, and writing the statements that link user-initiated events to the scripts.

Two Truths & A Lie

Understanding Event-Driven Programming

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

   T F

2. With event-driven programs, the user might initiate any number of events in any order.

   T F

3. Within an event-driven program, a component from which an event is generated, such as a button, is a listener. An object that is "interested in" an event is the source of the event.

   T F

12-2 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. A partial list is shown in Table 12-1. Most languages allow you to distinguish between many additional events. For example, you might be able to initiate different events when a mouse key is pressed, during the time it is held down, and when it is released.

Table 12-1

Common User-Initiated Events

Event	Description of User's Action

Key press	Pressing a key on the keyboard
Mouse point or mouse over	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 down the left mouse button while moving the mouse over the desk surface

You also need to be able to picture common GUI components. Table 12-2 describes some common GUI components, and Figure 12-3 shows how they look on a screen.

Table 12-2

Common GUI Components

Component	Description
Label	A rectangular area that displays text
Text box	A rectangular area into which the user can type text
Check box	A label placed beside a small square; you can click the square to display or remove a check mark, which selects or deselects an option
Option buttons	A group of options that are similar to check boxes. When the options are square, users typically can select any number of them; such options are called a *check box group*. When the options are round, they are often mutually exclusive and are called radio buttons.

List box	When the user clicks a list box, a menu of items appears. Depending on the options the programmer sets, you might be able to make only one selection, or you might be able to make multiple selections.
Button	A rectangular object you can click; when you do, its appearance usually changes to look pressed

Figure 12-3

Common GUI Components

When you program in a language that uses GUI components, you do not create them from scratch. Instead, you call prewritten methods that draw the GUI components on the screen for you. The components themselves are constructed using existing classes complete with names, attributes, and methods. In some programming languages, you can work in a text environment and write statements that instantiate GUI objects. In other languages, you can work in a graphical environment, drag GUI objects onto your screen from a toolbox, and arrange them appropriately for your application. Some languages offer both options. Either way, you do not think about the details of constructing the components. Instead, you concentrate on the actions that should occur 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.

When you use existing GUI components, you instantiate objects, each of which belongs to a

prewritten class. For example, you might use a `Button` object when you want the user to click a button to make a selection. Depending on the programming language, the `Button` class might contain attributes or properties such as the text on the `Button` and its position on the screen. The class might also contain methods such as `setText()` and `setPosition()`. For example, Figure 12-4 shows how a built-in `Button` class might be written.

Figure 12-4

Button Class

```
class Button
   Declarations
      private string text
      private num x_position
      private num y_position

   public void setText(string messageOnButton)
      text = messageOnButton
   return

   public void setPosition(num x, num y)
      x_position = x
      y_position = y
   return
endClass
```

The `x_position` and `y_position` of the `Button` object in Figure 12-4 refer to horizontal and vertical coordinates where the `Button` appears on an object, such as a window that appears on the screen during program execution. A **pixel** (A picture element; one of the tiny dots of light that form a grid on a monitor.) is one of the tiny dots of light that form a grid on your screen. The term *pixel* derives from combining the first syllables of *picture* and *element*. You will use x- and y-positions again when you learn about animation later in this chapter.

Watch the video *GUI Components*.

The `Button` class shown in Figure 12-4 is an abbreviated version so you can easily see its similarity to a class such as `Employee`, which you read about in Chapter 11. The figure shows three fields and two set methods. A working `Button` class in most programming languages would contain many more fields and methods. For example, a full-blown class

might also contain get methods for the text and position, and other fields and methods to manipulate a `Button's` font, color, size, and so on.

To create a `Button` object in a client program, you would write a statement similar to the following:

In this statement, `Button` is the type and `myProgramButton` is the name of the object you create. To use a `Button`'s methods, you would write statements such as the following:

Different GUI classes support different attributes and methods. For example, a `CheckBox` class might contain a method named `getCheckedStatus()` that returns `true` or `false`, indicating whether the `CheckBox` object has been checked. A `Button`, however, would have no need for such a method.

An important advantage of using GUI data-entry objects is that you often can control what users enter by limiting their options. When you provide a finite set of buttons to click or a limited number of menu items, the user cannot make unexpected, illegal, or bizarre choices. For example, if you provide only two buttons so the user must click Yes or No, you can eliminate writing code to handle invalid entries.

Two Truths & A Lie

User-Initiated Actions and GUI Components

1. In a GUI program, a key press is a common user-initiated event and a check box is a typical GUI component.

   T F

2. When you program in a language that supports event-driven logic, you call prewritten methods that draw GUI components on the screen for you.

   T F

3. An advantage of using GUI objects is that each class you use to create the objects supports identical methods and attributes.

T F

12-3 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 interface should be attractive, easy to read, and nondistracting.
- To some extent, it's helpful if the user can customize your applications.
- The program should be forgiving.
- The GUI is only a means to an end.

12-3a The Interface Should Be Natural and Predictable

The GUI program interface should represent objects like their real-world counterparts. In other words, it makes sense to use an icon that looks like a recycling bin to let a user drag files or other components to the bin and delete them. Using a recycling bin icon is “natural” in that people use one in real life when they want to discard actual items; dragging files to the bin is also “natural” because that's what people do with real items they discard. Using a recycling bin for discarded items is also predictable, because users are already familiar with the icon in other programs. 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 seems natural, but because other programs do not use such imagery, it is not as predictable.

GUIs should also be predictable in their layout. For example, when a user must enter personal information in text boxes, the street address is expected to come before the city and state. Also, a menu bar appears at the top of the screen in most GUI programs, and the first menu item is almost always *File*. If you design a program interface in which the menu 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 users. Either they will make mistakes when using your

program or they may give up using it entirely. It doesn't matter if you canprove that your layout plan is more efficient than the standard one—if you do not use a predictable layout, your program will be rejected in the marketplace.

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 is named for 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 because it is not predictable to users who know the QWERTY keyboard.

Stovetops often have an unnatural interface, making unfamiliar stoves more difficult for you to use. Most 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 likely to select the wrong knob when adjusting the burner's flame or heating element.

12-3b The Interface Should Be Attractive, Easy to Read, and Nondistracting

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. When it comes to GUI design, fancy fonts and weird color combinations are the signs of amateur designers. In addition, you should make sure that unavailable screen options are either dimmed (also called *grayed*) or removed, so the user does not waste time clicking components that aren't functional. An excellent way to learn about good GUI design is to pay attention to the design features used in popular applications and in Web sites you visit. Notice that the designs you like to use feel more “natural.”

Screen designs should not be distracting. When a screen has too many components, users can't find what they're looking for. When a component is no longer needed, it should be

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

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 filename dances across the screen or the speakers play a tune. However, after one or two experiences with your creative additions, users find that intruding design elements hamper the actual work of the program. Also, creative embellishments might consume extensive memory and CPU time, slowing an application's performance.

12-3c To Some Extent, It's Helpful If the User Can Customize Your Applications

All users work in their own way. If you are designing an application that will use numerous menus and toolbars, it's helpful if users can position components in the order that's easiest for them. Users appreciate being able to change features like 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. Making programs easier to use for people with physical limitations is known as enhancing **accessibility** (Describes screen design issues that make programs easier to use for people with physical limitations.) .

Don't forget that many programs are used internationally. If you can allow the user to work with a choice of languages, you might be able to market your program more successfully in other countries. If you can allow the user to convert prices to multiple currencies, you might be able to make sales in more markets.

12-3d 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 similar problems. You should always provide an escape route to accommodate users who make bad choices or change their minds. By providing a Back button or a working Escape key, you provide more functionality to your users. It also can be helpful to include an option for the user to revert to the default settings after making changes. Some users might be afraid to alter an application's features if they are not sure they can easily return to the original settings.

Users also appreciate being able to perform tasks in a variety of ways. For example, you might allow a user to select a word on a screen by highlighting it using a mouse or by holding down the Ctrl and Shift keys while pressing the right arrow key. A particular technique might be easier for people with disabilities, and it might be the only one available after the mouse batteries fail or the user accidentally disables the keyboard by spilling coffee on it.

12-3e The GUI Is Only a Means to an End

The most important principle of GUI design is to remember that a GUI is only an interface. Using a mouse to click items and drag them around is not the point of any business programs except those that train 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 a GUI program is done after the user clicks a button or makes a list box selection. Actual program tasks then take place.

Two Truths & A Lie

Designing Graphical User Interfaces

1. To keep the user's attention, a well-designed GUI interface should contain unique and creative controls.

   T F

2. To be most useful, a GUI interface should be attractive, easy to read, and nondistracting.

   T F

3. To avoid frustrating users, a well-designed program should be forgiving.

   T F

12-4 Developing an Event-Driven Application

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

1. Understanding the problem
2. Planning the logic
3. Coding the program
4. Translating the program into machine language
5. Testing the program
6. Putting the program into production
7. Maintaining the program

When you develop an event-driven application, you expand on Step 2 (planning the logic) and include three new substeps as follows:

2a. Creating storyboards

2b. Defining the objects

2c. Defining the connections between the screens the user will see

For example, suppose that you want to create a simple, interactive program that determines premiums for prospective insurance customers. A graphical interface will allow users to select a policy type—health or auto. Next, users answer pertinent questions about their age, driving record, and whether they smoke. Although most insurance premiums would be based on more characteristics than these, assume that policy rates are determined using the factors shown in Table 12-3. The final output of the program is a second screen that shows the semiannual premium amount for the chosen policy.

Table 12-3

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

12-4a Creating Storyboards

A **storyboard** (A picture or sketch of screens the user will see when running a program.) 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 12-5 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, which displays the semiannual premium.

Figure 12-5

Storyboard for Insurance Program

12-4b Defining the Storyboard 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** (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.) 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 12-6 shows an object dictionary for the insurance premium program. The type and name of each object to be placed on a screen are listed in the two left columns. The third 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 `calcButton`, 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 might need to name `calcRoutine()` something similar to `calcButton.click()`. In languages that use this format, a standard method named `click()` holds the statements that execute when the user clicks the `calcButton`.

Figure 12-6

Object Dictionary for Insurance Premium Program

12-4c Defining Connections Between the User Screens

The insurance premium program is small, but with larger programs you may need to draw the connections between the screens to show how they interact. Figure 12-7 shows an interactivity diagram for the screens used in the insurance premium program.

Figure 12-7

Interactivity Diagram for Insurance Premium Program

An **interactivity diagram** (A diagram that shows the relationship between screens in an interactive GUI program.) shows the relationship between screens in an interactive GUI program. Figure 12-7 shows that the first screen calls the second screen, and the program ends.

Figure 12-8 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 of these three screens may lead to different screens, depending on the options the user selects at a previous screen.

Figure 12-8

Interactivity Diagram for a Complicated Program

12-4d 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 start to plan the program class. For example, following the storyboard plan for the insurance program (see Figure 12-5), you need to create the first

screen, which contains four labels, four sets of radio buttons, and a button. Figure 12-9 shows the pseudocode that creates these components.

Figure 12-9

Component Definitions for First Screen of Insurance Program

When you use an integrated development environment to create applications, you can drag components like those in Figure 12-9 onto a screen without explicitly writing all the statements shown in the pseudocode. In that case, the coding statements will be generated

for you. It's beneficial to understand these statements so that you can more easily modify and debug your programs.

In Figure 12-9, the statement `calcButton.registerListener(calcRoutine())` specifies that `calcRoutine()` executes when a user clicks the `calcButton`. The syntax of this statement varies among programming languages. With most object-oriented programming (OOP) languages, you must **register components** (The act of signing up components so they can react to events initiated by other components.) , or sign them up so that they can react to events initiated by other components. The details vary among languages, but the basic process is to write a statement that links the appropriate method (such as the `calcRoutine()` or `exitRoutine()` method) withanevent such as auser's button click. In many development environments, the statement that registers a component to react to a user-initiated event is written for you automatically when you click components while designing your screen.

In reality, you might generate more code than that shown in Figure 12-9 when you create the insurance program components. For example, each component might require a color and font. You also might want to initialize some components with default values to indicate they are selected. For example, you might want one radio button in a group to be selected already, which allows the user to click a different option only if he does not want the default value.

You also need to create the component that holds all the GUI elements in Figure 12-9. Depending on the programming language, you might use a class with a name such as `Screen`, `Form`, or `Window`. Each of these generically is a **container** (One of a class of objects whose main purpose is to hold other elements—for example, a window.) , or a class of objects whose main purpose is to hold other elements. The container class contains methods that allow you to set physical properties such as height and width, as well as methods that allow you to add the appropriate components to a container. Figure 12-10 shows how you would define a Screen class, set its size, and add the necessary components.

Figure 12-10

Statements that Create `screen1`

Similarly, Figure 12-11 shows how you can create and define the components for the second screen in the insurance program and how to add the components to the container. Notice the label that holds the user's insurance premium is not filled with text, because the amount is not known until the user makes all the selections on the first screen.

Figure 12-11

Statements that Define and Create `screen2` and Its Components

After the GUI components are designed and arranged, you can plan the logic for each of the methods or scripts that the program will use. For example, given the program requirements shown earlier in Table 12-3, you can write the pseudocode for the `calcRoutine()` method of the insurance premium program, as shown in Figure 12-12. The `calcRoutine()` method does not execute until the user clicks the `calcButton`. At that point,

the user's choices are sent to the method and used to calculate the premium amount.

Figure 12-12

Pseudocode for `calcRoutine()` Method of Insurance Premium Program

The pseudocode in Figure 12-12 should look very familiar to you—it declares numeric constants and a variable and uses decision-making logic you have used since the early chapters of this book. After the premium is calculated based on the user's choices, it is placed in the label that appears on the second screen. The basic structures of sequence, selection, and looping will continue to serve you well, whether you are programming in a procedural or event-driven environment.

The last two statements in the `calcRoutine()` method indicate that after the insurance premium is calculated and placed in its label, the first screen is removed and the second screen is displayed. Screen removal and display are accomplished differently in different languages; this example assumes that the appropriate methods are named `remove()` and `display()`.

Two more program segments are needed to complete the insurance premium program. These segments include the main program that executes when the program starts and the last method that executes when the program ends. In many GUI languages, the process is

slightly more complicated, but the general logic appears in Figure 12-13. The final method in the program is associated with the `exitButton` object on `screen2`. In Figure 12-13, this method is called `exitRoutine()`. In this example, the main program sets up the first screen and the last method removes the last screen.

Figure 12-13

The Main Program and `exitRoutine()` Method for the Insurance Program

Two Truths & A Lie

Developing an Event-Driven Application

1. A storyboard represents a diagram of the logic used in an interactive program.

   T F

2. 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.

   T F

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

   T F

12-5 Understanding Threads and Multithreading

A **thread** (The flow of execution of one set of program statements.) is the flow of execution of one set of program statements. When you execute a program statement by statement, from beginning to end, you are following a thread. Many applications follow a single thread; this means that the application executes only a single program statement at a time. If a computer has more than one central processing unit (CPU), then each can execute a thread at the same time. However, if a computer has a single CPU and the system supports only single threading, then tasks must occur one at a time. For example, Figure 12-14 shows how three tasks might execute in a single thread in a computer with a single CPU. Each task must end before the next task starts.

Figure 12-14

Executing Multiple Tasks as Single Threads in a Single-Processor System

Even if the computer has only one CPU, all major OOP languages allow you to launch, or start, multiple threads of execution by using a technique known as **multithreading** (Using multiple threads of execution.) . With multithreading, threads share the CPU's time, as shown in Figure 12-15. The CPU devotes a small amount of time to one task, and then devotes a small amount of time to another. The CPU never actually performs two tasks at the same instant. Instead, it performs a piece of one task and then part of another. The CPU performs so quickly that each task seems to execute without interruption.

Figure 12-15

Executing Multiple Threads in a Single-Processor System

Perhaps you have seen an expert chess player participate in games with several opponents at once. The expert makes a move on the first chess board, and then moves to the second board against a second opponent while the first opponent analyzes his next move. The master can move to the third board, make a move, and return to the first board before the first opponent is even ready to respond. To the first opponent, it might seem as though the expert is devoting all of his time to the first game. Because the expert is so fast, he can play other opponents while the first opponent contemplates his next move. Executing multiple threads on a single CPU is a similar process. The CPU transfers its attention from thread to thread so quickly that the tasks don't even “miss” the CPU's attention.

You use multithreading to improve the performance of your programs. Multithreaded programs often run faster, but more importantly, they are more user-friendly. With a multithreaded program, a user can continue to make choices by clicking buttons while the program is reading a data file. An animated figure can appear on one part of the screen while the user makes menu selections elsewhere on the screen. When you use the Internet, the benefits of multithreading increase. For example, you can begin to read a long text file, watch a video, or listen to an audio file while the file is still downloading. Web users are likely to abandon a site if they cannot use it before a lengthy downloading process completes. When you use multithreading to perform concurrent tasks, you are more likely to retain visitors to your Web site—this is particularly important if your site sells a product or service.

Programmers sometimes describe thread execution as a *lightweight process* because it is not a full-blown program. Rather, a thread must run within the context of a full, heavyweight program.

Writing good code to execute multithreading requires skill. Without careful coding, problems can arise such as **deadlock** (A flaw in multithreaded programs in which two or more threads wait for each other to execute.) , in which two or more threads wait for each other to execute, and **starvation** (A flaw in multithreaded programs in which a thread is abandoned because other threads occupy all the computer's resources.) , in which a thread is abandoned because other threads occupy all the computer's resources.

When threads share an object, special care is needed to avoid unwanted results.

For example, consider a customer order program in which two clerks are allowed to fill orders concurrently. Imagine the following scenario:

- The first clerk accesses an inventory file and tells a customer that one item is left.
- A second clerk accesses the file and tells a different customer that one item is left.
- The first customer places an order, and inventory is reduced to 0.

- The second customer places an order, and inventory is reduced to –1.

Two items have been ordered, but only one exists, and the inventory file is now incorrect. There will be confusion in the warehouse, problems in the Accounting department, and one unsatisfied customer. Similar problems can occur in programs that reserve airline seats or concert tickets. OOP languages provide sophisticated techniques, known as **thread synchronization** (A set of techniques that coordinates threads of execution to help avoid potential multithreading problems.) , that help avoid these potential problems.

Object-oriented languages often contain a built-in `Thread` class that contains methods to help handle and synchronize multiple threads. For example, a `sleep()` method is frequently used to pause program execution for a specified amount of time. Computer processing speed is so rapid that sometimes you have to slow down processing for human consumption. The next section describes one application that frequently requires a `sleep()` method—computer animation.


Watch the video *Threads and Multithreading*.

Two Truths & A Lie

Understanding Threads and Multithreading

1. In the last few years, few programs that follow a single thread have been written.

   T F

2. Single-thread programs contain statements that execute in very rapid sequence, but only one statement executes at a time.

   T F

3. When you use a computer with multiple CPUs, the computer can execute multiple instructions simultaneously.

   T F

12-6 Creating Animation

Animation (The rapid sequence of still images, each slightly different from the previous one, that produces the illusion of movement.) is the rapid sequence of still images, each slightly different from the previous one, that produces the illusion of movement. Cartoonists create animated films by drawing a sequence of frames or cells. These individual drawings are shown to the audience in rapid succession to create the sense of natural movement. You create computer animation using the same techniques. If you display computer images as fast as your CPU can process them, you might not be able to see anything. Most computer animation employs a `Thread` class `sleep()` method to pause for short intervals between animation cells so the human brain has time to absorb each image's content.

Many object-oriented languages offer built-in classes that contain methods you can use to draw geometric figures. The methods typically have names like `drawLine()`, `drawCircle()`, `drawRectangle()`, and so on. You place figures on the screen based on a graphing coordinate system. Each component has a horizontal, or **x-axis** (An imaginary line that represents horizontal positions in a screen window.) , position as well as a vertical, or **y-axis** (An imaginary line that represents vertical positions in a screen window.) , position on the screen. The upper-left corner of a display is position 0, 0. The first, or **x-coordinate** (A position value that increases from left to right across a window.) , value increases as you travel from left to right across the window. The second, or **y-coordinate** (A position value that increases from top to bottom across a window.) , value increases as you travel from top to bottom. Figure 12-16 shows four screen coordinate positions.

Figure 12-16

Selected Screen Coordinate Positions

Artists often spend a great deal of time creating the exact images they want to use in an animation sequence. As a simple example, Figure 12-17 shows pseudocode for a `MovingCircle` class. As its name implies, the class moves a circle across the screen. The class contains data fields to hold x- and y-coordinates that identify the location at which a circle appears. The constants `SIZE` and `INCREASE` define the size of the first circle drawn and the relative increase in size and position of each subsequent circle. The `MovingCircle` class assumes that you are working with a language that provides a `drawCircle()` method, which creates a circle when given parameters for horizontal and vertical positions and

radius. Assuming you are working with a language that provides a `sleep()` method to accept a pause time in milliseconds, the `SLEEP_TIME` constant provides a 100-millisecond gap before the production of each new circle.

Figure 12-17

The `MovingCircle` Class

In most object-oriented languages, a method named `main()` executes automatically when a class object is created. The `main()` method in the `MovingCircle` class executes a continuous loop. A similar technique is used in many languages that support GUI interfaces. Program execution will cease only when the user quits the application—by clicking a window's Close button, for example. In the `repaintScreen()` method of the `MovingCircle` class, a circle is drawn at the `x`, `y` position, then `x`, `y`, and the circle size are increased. The application sleeps for one-tenth of a second (the `SLEEP_TIME` value), and then the `repaintScreen()` method draws a new circle more to the right, further down, and a little larger. The effect is a moving circle that leaves a trail of smaller circles behind as it moves diagonally across the screen. Figure 12-18 shows the output as a Java version of the application executes.

Figure 12-18

Output of the `MovingCircle` Application

Although an object-oriented language might make it easy to draw geometric shapes, you also can substitute a variety of more sophisticated, predrawn animated images to achieve the graphic effects you want within your programs. An image is loaded in a separate thread of execution, which allows program execution to continue while the image loads. This is a significant advantage because loading a large image can be time-consuming.

Many animated images are available on the Web for you to use freely. Use your search engine and keywords such as *gif files*, *jpeg files*, and *animation* to find sources for shareware and freeware files.

Two Truths & A Lie

Creating Animation

1. Each component you place on a screen has a horizontal, or x-axis, position as well as a vertical, or y-axis, position.

   T F

2. The x-coordinate value increases as you travel from left to right across a window.

   T F

3. You almost always want to display animation cells as fast as your processor can handle them.

T F

12-7 Chapter Review

12-7a 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.

- A user can initiate many events, such as a key press, mouse point, click, right-click, double-click, and drag. Common GUI components include labels, text boxes, buttons, check boxes, check box groups, option buttons, and list boxes. 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.

- Developing event-driven applications requires more steps than developing other programs. The steps include creating storyboards, defining objects, and defining the connections between the screens the user will see.

- A thread is the flow of execution of one set of program statements. Many applications follow a single thread; others use multithreading so that diverse tasks can execute concurrently.

- Animation is the rapid sequence of still images that produces the illusion of movement. Many object-oriented languages contain built-in classes that contain methods you can use to draw geometric figures on the screen. Each component has a horizontal, or x-axis, position as well as a vertical, or y-axis, position on the screen.

Chapter Review

12-7b Key Terms

- **DOS prompt** (The command line in the DOS operating system.)
- **operating system** (The software that runs a computer and manages its resources.)
- **Icons** (Small pictures on a screen that a user can select with a mouse.)
- **event** (An occurrence that generates a message sent to an object.)
- **event-driven or event-based** (Describes programs and actions that occur in response to user-initiated events such as clicking a mouse button.)
- **source of an event** (The component from which an event is generated.)
- **listener** (An object that is prepared to respond to events from specified sources.)
- **script** (A procedural module that depends on user-initiated events in object-oriented programs.)
- **pixel** (A picture element; one of the tiny dots of light that form a grid on a monitor.)
- **Accessibility** (Describes screen design issues that make programs easier to use for people with physical limitations.)
- **storyboard** (A picture or sketch of screens the user will see when running a program.)
- **object dictionary** (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.)
- **interactivity diagram** (A diagram that shows the relationship between screens in an interactive GUI program.)
- **Registering components** (The act of signing up components so they can react to events initiated by other components.)
- **container** (One of a class of objects whose main purpose is to hold other elements—for example, a window.)
- **thread** (The flow of execution of one set of program statements.)
- **Multithreading** (Using multiple threads of execution.)
- **Deadlock** (A flaw in multithreaded programs in which two or more threads wait for each other to execute.)

- **Starvation** (A flaw in multithreaded programs in which a thread is abandoned because other threads occupy all the computer's resources.)
- **Thread synchronization** (A set of techniques that coordinates threads of execution to help avoid potential multithreading problems.)
- **Animation** (The rapid sequence of still images, each slightly different from the previous one, that produces the illusion of movement.)
- **x-axis** (An imaginary line that represents horizontal positions in a screen window.)
- **y-axis** (An imaginary line that represents vertical positions in a screen window.)
- **x-coordinate** (A position value that increases from left to right across a window.)
- **y-coordinate** (A position value that increases from top to bottom across a window.)

Chapter 12: Event-Driven GUI Programming, Multithreading, and Animation: 12-7c Review Questions
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)

Chapter Review

12-7c Review Questions

1. Compared to using a command line, an advantage to using an operating system that employs a GUI 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 the mouse on an icon, the program is ____ -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 box

    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 of the following 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 applications.

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

    a. A menu bar runs down the right side of the screen.

    b. *Help* is the first option on a menu.

    c. A dollar sign icon represents saving a file.

    d. Pressing *Esc* allows the user to cancel a selection.

10. In most GUI programming environments, the programmer can change all of the following attributes of most components except their ____ .

    a. color

    b. screen location

    c. size

    d. You can change all of these attributes.

11. Depending on the programming language, 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 objects?

    a. Translate the program.

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. tale timber

14. An object ____ is a list of objects used in a program.

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. The flow of execution of one set of program statements is a ____ .

a. thread

b. string

c. path

d. route

17. When a computer contains a single CPU, it can execute ____ computer instruction(s) at a time.

a. one

b. several

c. an unlimited number of

d. from several to thousands of

18. Multithreaded programs usually ____ than their procedural counterparts.

a. run faster

b. are harder to use

c. are older

d. all of the above

19. An object's horizontal position on the computer screen is its ____ .

a. a-coordinate

b. h-coordinate

c. x-coordinate

d. y-coordinate

20. You create computer animation by ____ .

a. drawing an image and setting its animation property to true

b. drawing a single image and executing it on a multiprocessor system

c. drawing a sequence of frames that are shown in rapid succession

d. Animation is not used in computer applications.

Chapter Review

12-7d Exercises

1. Take a critical look at three GUI applications you have used—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 you have used. 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 scripts for an interactive program for customers of Sanderson's Ice Cream Sundaes.

   Allow customers the option of choosing a three-scoop, two-scoop, or one-scoop creation at a base price of $4.00, $3.00, or $2.20, respectively. Let the customer choose chocolate, strawberry, or vanilla as the primary flavor. If the customer adds nuts, whipped cream, or cherries to the order, add $0.50 for each 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 scripts for an interactive program for customers of Carrie's Custom T-Shirts.

   Allow customers the option of five T-shirt sizes and styles—for example, *XL long sleeve* or *M short sleeve*. Assume that each product has a unique price that is displayed when the user clicks a Buy Now button.

6. Design the storyboards, interactivity diagram, object dictionary, and any necessary scripts for an interactive program for customers of the Dharma Day Spa.

   Allow customers the option of choosing a manicure ($10), pedicure ($25), or both ($32). After the customer clicks a Select button, display the price of the service.

Find the Bugs

7. Your downloadable files for Chapter 12 include DEBUG12-01.txt, DEBUG12-02.

txt, and DEBUG12-03.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

8. Design the storyboards, interactivity diagram, object dictionary, and any necessary scripts for an interactive program that allows a user to play a card game named Lucky Seven. In real life, the game can be played with seven cards, each containing a number from 1 through 7, that are shuffled and dealt number-side down. To start the game, a player turns over any card. The exposed number on the card determines the position (reading from left to right) of the next card that must be turned over. For example, if the player turns over the first card and its number is 7, the next card turned must be the seventh card (counting from left to right). If the player turns over a card whose number denotes a position that was already turned, the player loses the game. If the player succeeds in turning over all seven cards, the player wins.

   Instead of cards, you will use seven buttons labeled 1 through 7 from left to right. Randomly associate one of the seven values 1 through 7 with each button. (In other words, the associated value might or might not be equivalent to the button's labeled value.) When the player clicks a button, reveal the associated hidden value. If the value represents the position of a button already clicked, the player loses. If the revealed number represents an available button, force the user to click it—that is, do not take any action until the user clicks the correct button. After a player clicks a button, remove the button from play.

   For example, a player might click Button 7, revealing a 4. Then the player clicks Button 4, revealing a 2. Then the player clicks Button 2, revealing a 7. The player loses because Button 7 is already “used.”

9. In the Game Zone sections of Chapters 6 and 9, you designed and fine-tuned the logic for the game Hangman, in which the user guesses letters in a series of hidden words. Design the storyboards, interactivity diagram, object dictionary, and any necessary scripts for a version of the game in which the user clicks lettered buttons to fill in the secret words. Draw a “hanged” person piece by piece with each missed letter. For example, when the user

chooses a correct letter, place it in the appropriate position or positions in the word, but the first time the user chooses a letter that is not in the target word, draw a head for the “hanged” man. The second time the user makes an incorrect guess, add a torso. Continue with arms and legs. If the complete body is drawn before the user has guessed all the letters in the word, display a message indicating that theplayerhas lost thegame. If theusercompletes thewordbeforeall thebody parts are drawn, display a message that the player has won. Assume that you can use built-in methods named `drawCircle()` and `drawLine()`.The `drawCircle()` method requires three parameters—the x-and y-coordinates of the center, and aradiussize. The `drawLine()` method requires four parameters—the x- and y-coordinates of the start of the line, and the x- and y-coordinates of the end of the line.

Up for Discussion

10. Making exciting, entertaining, professional-looking GUI applications becomes easier once you learn to include graphics images. You can copy these images from many locations on the Web. Should there be any restrictions on their use? Does it make a difference if you are writing programs for your own enjoyment as opposed to putting them on the Web where others can see them? Is using photographs different from using drawings? Does it matter if the photographs contain recognizable people? Would you impose any restrictions on images posted to your organization's Web site?

11. Playing computer games has been shown to increase the level of dopamine in the human brain. High levels of this substance are associated with addiction to drugs. Suppose that you work for a computer game company that decides to research how its products can produce more dopamine in the brains of players. Would you support the company's decision?

12. When people use interactive programs on the Web, do you feel it is appropriate to track which buttons they click or to record the data they enter? When is it appropriate, and when is it not? Does it matter how long the data is stored? Does it matter if a profit is made from using the data?

13. Should there be limits on Web content? Consider sites that might display pornography, child abuse, suicide, or the assassination of a political leader. Does it make a difference if the offensive images are shown as animation?

Chapter 12: Event-Driven GUI Programming, Multithreading, and Animation: 12-7d Exercises
Book Title: Programming Logic and Design
Printed By: Ronald Suchy (rsuchy@mayfieldschools.org)
© 2013 ,