Learn to code quickly and painlessly using Apple's newest Swift programming language

Swift OS X Programming for Absolute Beginners

Wallace Wang

Apress®

Swift OS X Programming for Absolute Beginners

Wallace Wang

Apress®

Swift OS X Programming for Absolute Beginners

ISBN-13 (pbk): 978-1-4842-1234-9

ISBN-13 (electronic): 978-1-4842-1233-2

Managing Director: Welmoed Spahr
Lead Editor: Louise Corrigan
Technical Reviewer: Bruce Wade
Editorial Board: Steve Anglin, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Michelle Lowman, James Markham, Susan McDermott, Matthew Moodie, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Gwenan Spearing, Steve Weiss
Coordinating Editor: Mark Powers
Copy Editor: Karen Jameson
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover photograph by Martijn Vroom

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com/9781484212349. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

This book is dedicated to everyone who always wanted to learn how to program on the Macintosh. You can learn anything you want just as long as you're willing to take the time and believe in yourself.

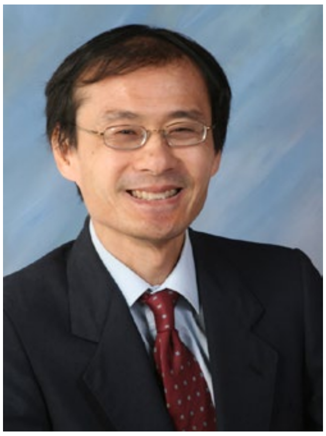
*"Promise me you'll always remember: You're braver than you believe,
and stronger than you seem, and smarter than you think."*

—Christopher Robin to Winnie the Pooh

Contents at a Glance

Contents

About the Author



Wallace Wang is a former Windows enthusiast who took one look at Vista and realized that the future of computing belonged to the Macintosh. He has written more than forty computer books, including *Microsoft Office for Dummies, Beginning Programming for Dummies, Steal This Computer Book, My New Mac,* and *My New iPad*. In addition to programming the Macintosh and iPhone/iPad, he also performs stand-up comedy, having appeared on A&E s “Evening at the Improv,” and having performed in Las Vegas at the Riviera Comedy Club at the Riviera Hotel & Casino. When he is not writing computer books or performing stand-up comedy, he also appears on a local San Diego radio show called “Notes From the Underground” that airs twice a week on a radio station KNSJ (`www.knsj.org`).

In addition, he enjoys blogging about screenwriting at his site, The 15 Minute Movie Method (`www.15minutemoviemethod.com`), where he shares screenwriting tips with other aspiring screenwriters who all share the goal of breaking into Hollywood. Some of his other sites include Cat Daily News (`www.catdailynews.com`) that highlights interesting news about cats, The Electronic Author (`www.electronicauthor.com`) that focuses on self-publishing, and Top Bananas (`www.topbananas.com`) that covers the possibilities and application of technology related to Apple and other mobile and wearable computer manufacturers.

About the Technical Reviewer

Bruce Wade is the founder of Warply Designed Inc. (`www.warplydesigned.com`), a company specializing in using game technology for real-world applications. He has more than sixteen years of software development experience with a strong focus on 2D/3D animation and interactive applications, primarily using Apple technology.

Acknowledgments

Thanks go to all the wonderful people at APress for giving me a chance to write about Swift, my latest favorite programming language for the Macintosh, iPhone, iPad, and Apple Watch.

Additional thanks go to Dane Henderson and Elizabeth Lee (www.echoludo.com) who share the airwaves with me on our radio show “Notes From the Underground” (http://notesfromtheu.com) on KNSJ.org. More thanks go to Chris Clobber, Diane Jean, Ikaika Patria and Robert Weems for letting me share their friendship and lunacy on another KNSJ radio show called “Laugh in Your Face,” (http://www.laughinyourfaceradio.com) which combines comedy with political activism and commentary.

A special mention goes towards Michael Montijo and his indomitable spirit that has him driving from Phoenix to Los Angeles once a month for the past fifteen years to meet with Hollywood executives. His cartoon show ideas, “Life of Mikey” and “Pachuko Boy,” will one day make it to television because he never gave up despite all the obstacles in his way.

Thanks also go to my wife, Cassandra, and son, Jordan, for putting up with a house filled with more gadgets than actual living people. Final thanks go to my cats, Oscar and Mayer, for walking over the keyboard, knocking over laptops, and chewing on power cords at the most inconvenient times of the day.

Introduction

Whether you're a complete novice looking to get started in programming, someone familiar with programming but curious about learning more, or a seasoned programmer comfortable with other programming languages but unfamiliar with Macintosh programming, this book is for you. Whatever your skill level, this book will help everyone understand how to use Apple's latest programming language, Swift, to create OS X programs for the Macintosh.

Now you may be wondering why learn Swift and why program for the Macintosh? The answer is simple.

First, Swift is Apple's newest programming language designed to make creating OS X and iOS programs faster, easier, and more reliable than before. Previously, you had to use Objective-C to create OS X and iOS apps. While powerful, Objective-C is much harder to learn; more complicated to read and write; and because of its complexity, more prone to introducing errors or bugs in a program.

On the other hand, Swift is just as powerful as Objective-C (actually more powerful as you'll soon see), far easier to learn, and much simpler to read and write while also minimizing common programming errors at the same time. Swift gives you all the benefits of Objective-C with none of the drawbacks. Plus Swift gives you features that Objective-C doesn't offer, which makes Swift a far better programming language to learn and use today and tomorrow. Since Swift is Apple's official programming language, you can be certain learning Swift will lead to greater opportunities now and long into the future.

Second, you may wonder why learn to create Macintosh programs? After all, the hot trend is learning to create iOS apps for the iPhone, iPad, and Apple Watch. If you plan on developing software, you definitely want to use Swift to create iOS apps.

However, learning Swift means understanding the following:

- The principles of programming and object-oriented programming in particular
- The syntax of the Swift programming language
- Xcode's features
- Apple's software development framework (called Cocoa) that forms the foundation of every OS X and iOS program
- The principles of user interface design

Does this sound like a lot to learn? Don't worry. We'll go through each process step by step so you won't feel lost. The point is that to create OS X programs and iOS apps, you need to learn multiple topics, but creating iOS apps poses an additional challenge.

For example, an iOS app needs to respond to touch gestures with one finger, two fingers, swipes, shakes, and motion in addition to adapting to changes when the user flips an iPhone or iPad left, right, upside down, or right side up.

In comparison, a Macintosh program only needs to respond to keyboard and mouse input. That means OS X programs are much simpler to create and understand, which also means that learning Swift to create OS X programs is far easier than learning Swift to create iOS apps.

Best of all, the principles are exactly the same. What you learn creating OS X programs are the exact same skills you need to create iOS apps. The difference is that creating OS X programs is far easier, less confusing, and much less intimidating than creating iOS apps.

Trying to create iOS apps right from the start can be like trying to swim across the English Channel before you even know how to hold your breath underwater.

You don't want to frustrate yourself unnecessarily. That's why it's much easier to learn the principles of iOS app programming by first learning OS X programming. Once you're familiar with OS X programming, you'll find it's trivial to transfer your programming skills to creating iOS apps. By learning to create OS X programs in Swift, you'll learn everything you need to know to eventually create iOS apps in Swift, plus you'll know how to create OS X programs so you can tap into the growing Macintosh market as well.

Following Lucrative Programming Trends

The introduction of a new computer platform has always ushered in a lucrative period for programmers. In the early 80s, the hottest platform was the Apple II computer. If you wanted to make money writing programs, you wrote programs to sell to Apple II computer owners, such as Dan Bricklin did, an MBA graduate student at the time, when he wrote the first spreadsheet program, VisiCalc.

Then the next big computing platform shift occurred in the mid-80s with the IBM PC and MS-DOS. People made fortunes off the IBM PC including Bill Gates and Microsoft, which went from a small, startup company to the most dominant computer company in the world. The IBM PC made millionaires out of hundreds of people including Scott Cook, a former marketing director at Proctor & Gamble, who developed the popular money manager program, Quicken.

Microsoft helped usher in the next computer platform when they shifted from MS-DOS to Windows and put a friendly graphical user interface on IBM PCs. Once again, programming Windows became the number one way that programmers and non-programmers alike made fortunes by writing and selling their own Windows programs. Microsoft took advantage of the shift to Windows by releasing several Windows-only programs that have become fixtures of the business world such as Outlook, Access, and Excel.

Now the world is shifting toward the new computer platform of Apple products running OS X and iOS. Thousands of people, just like you, are eager to start writing programs to take advantage of the Macintosh's rising market share along with the dominant position of the iPhone and the iPad in the smart phone and tablet categories, and the Apple Watch in the wearable computer market.

Besides experienced developers, amateurs, hobbyists, and professionals in other fields are also interested in writing their own games, utilities, and business software specific to their particular niche.

Many programmers have gone from knowing nothing about programming to earning thousands of dollars a day by creating iPhone/iPad apps or Macintosh programs. As the Macintosh, iPhone, iPad, and now the Apple Watch continue gaining market share all over the world, more people will use one or more of these products, increasing the potential market for you.

All this means is that it's a perfect time for you to start learning how to program your Macintosh right now because the sooner you understand the basics of Macintosh programming, the sooner you can start creating your own Macintosh programs along with iPhone/iPad/Apple Watch apps.

What to Expect From This Book

Whether you're a complete novice or a seasoned programmer coming from another programming environment, this book will minimize technical jargon and focus on helping you understand what to do and why.

If you just want to get started and learn the basics of programming in Swift, this book is for you. If you're already an experienced Windows programmer and want to get started programming the Macintosh, this book can be especially helpful in teaching you the basics in a hurry.

If you've never programmed before in your life, or if you're already familiar with programming but not with Macintosh programming, then this book is for you. Even if you're experienced with Macintosh programming, you may still find this book handy as a reference to help you achieve certain results without having to wade through several books to find an answer.

You won't learn everything you need to create your own super-sophisticated programs, but you'll learn just enough to get started, feel comfortable using Xcode, and be able to tackle other programming books with more confidence and understanding. Fair enough? If so, then turn the page and let's get started.

Note All code in this book was tested using Swift 2 in Xcode 7. If you're using Xcode 6, some features in this book won't work.

Chapter 1

Understanding Programming

Programming is nothing more than writing step-by-step instructions for a computer to follow. If you've ever written down the steps for a recipe or scribbled directions for taking care of your pets while you're on vacation, you've already gone through the basic steps of writing a program. The key is simply knowing what you want to accomplish and then making sure you write the correct instructions that will tell someone how to achieve that goal.

Although programming is theoretically simple, it's the details that can trip you up. First, you need to know exactly what you want. If you wanted a recipe for cooking chicken chow mein, following a recipe for cooking baked salmon won't do you any good.

Second, you need to write down every instruction necessary to get you from your starting point to your desired result. If you skip a step or write steps out of order, you won't get the same result. Try driving to a restaurant where your list of driving instructions omits telling you when to turn on a specific road. It doesn't matter if 99 percent of the instructions are right; if just one instruction is wrong, you won't get to your desired goal.

The simpler your goal, the easier it will be to achieve it. Writing a program that displays a calculator on the screen is far simpler than writing a program to monitor the safety systems of a nuclear power plant. The more complex your program, the more instructions you'll need to write, and the more instructions you need to write, the greater the chance you'll forget an instruction, write an instruction incorrectly, or write instructions in the wrong order.

Programming is nothing more than a way to control a computer to solve a problem, whether that computer is a laptop, smart phone, tablet, or wearable watch. Before you can start writing your own programs, you need to understand the basic principles of programming in the first place.

Note Don't get confused between learning programming and learning a particular programming language. You can actually learn the principles of programming without touching a computer at all. Once you understand the principles of programming, you can easily learn any particular programming language such as Swift.

Programming Principles

To write a program, you have to write instructions that the computer can follow. No matter what a program does or how big it may be, every program in the world consists of nothing more than step-by-step instructions for the computer to follow, one at a time. The simplest program can consist of a single line such as:

```
print ("Hello, world"!)
```

Obviously, a program that consists of a single line won't be able to do much, so most programs consist of multiples lines of instructions (or code) such as:

```
print ("Hello, world!")
print ("Now the program is done.")
```

This two-line program starts with the first line, follows the instructions on the second line, and then stops. Of course, you can keep adding more instructions to a program until you have a million instructions that the computer can follow sequentially, one at a time.

Listing instructions sequentially is the basis for programming. Unfortunately, it's also limiting. For example, if you wanted to print the same message five times, you could use the following:

```
print ("Hello, world!")
print ("Hello, world!")
print ("Hello, world!")
print ("Hello, world!")
print ("Hello, world!")
```

Writing the same five instructions is tedious and redundant, but it works. What happens if you want to print this same message a thousand times? Then you'd have to write the same instruction a thousand times.

Writing the same instruction multiple times is clumsy. To make programming easier, the goal is to write the least number of instructions to get the most work done. One way to avoid writing the same instruction multiple times is to organize your instructions using a second basic principle of programming, which is called a loop.

The idea behind a loop is to repeat one or more instructions multiple times, but only by writing those instructions down once. A typical loop might look like this:

```
for i in 1...5 {
  print ("Hello, world!")
}
```

The first line tells the computer to repeat the loop five times. The second line tells the computer to print the message "Hello, world" on the screen. The third line just defines the end of the loop.

Now if you wanted to make the computer print a message one thousand times, you don't need to write the same instruction a thousand times. Instead, you just need to modify how many times the loop repeats such as:

```
for i in 1...1000 {
  print ("Hello, world!")
}
```

Although loops are slightly more confusing to read and understand than a sequential series of instructions, loops make it easier to repeat instructions without writing the same instructions multiple times.

Most programs don't exclusively list instructions sequentially or in loops, but use a combination of both such as:

```
print ("Hello, world!")
print ("Now the program is starting.")
for i in 1...1000 {
  print ("Hello, world!")
}
```

In this example, the computer follows the first two lines sequentially and then follows the last three lines repetitively in a loop. Generally, listing instructions sequentially is fine when you only need the computer to follow those instructions once. When you need the computer to run instructions multiple times, that's when you need to use a loop.

What makes computers powerful isn't just the ability to follow instructions sequentially or in a loop, but in making decisions. Decisions mean that the computer needs to evaluate some condition and then, based on that condition, decide what to do next.

For example, you might write a program that locks someone out of a computer until that person types in the correct password. If the person types the correct password, then the program needs to give that person access. However, if the person types an incorrect password, then the program needs to block access to the computer. An example of this type of decision making might look like this:

```
if password == "secret" {
    print ("Access granted!")
} else {
    print ("Login denied!")
}
```

In this example, the computer asks for a password and when the user types in a password, the computer checks to see if it matches the word "secret." If so, then the computer grants that person access to the computer. If the user did not type "secret," then the computer denies access.

Making decisions is what makes programming flexible. If you write a sequential series of instructions, the computer will follow those lists of instructions exactly the same, every time. However, if you include decision-making instructions, also known as branching instructions, then the computer can respond according to what the user does.

Consider a video game. No video game could be written entirely with instructions organized sequentially because then the game would play exactly the same way every time. Instead, a video game needs to adapt to the player's actions at all times. If the player moves an object to the left, the video game needs to respond differently than if the player moves an object to the right or gets killed. Using branching instructions gives computers the ability to react differently so the program never runs exactly the same.

To write a computer program, you need to organize instructions in one of the following three ways as graphically show in Figure 1-1:

- Sequentially – the computer follows instructions one after another
- Loop – the computer repetitively follows one or more instructions
- Branching – the computer chooses to follow one or more group of instructions based on outside data

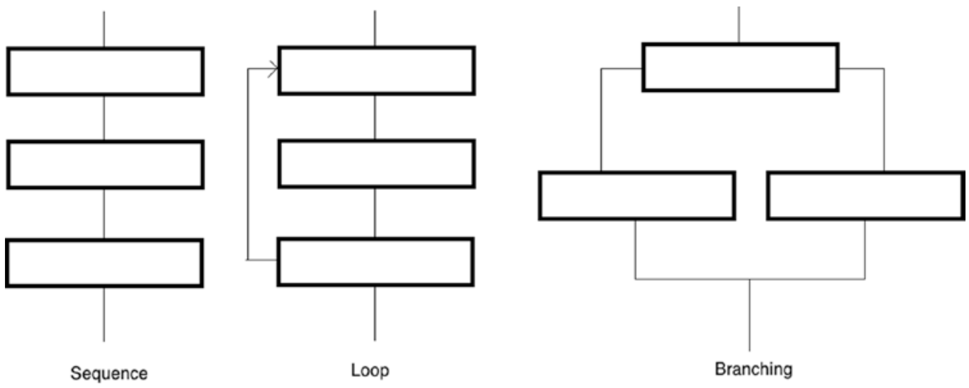


***Figure 1-1.** The three basic building blocks of programming*

While simple programs may only organize instructions sequentially, every large program organizes instructions sequentially, in loops, and in branches. What makes programming more of an art and less of a science is that there is no single best way to write a program. In fact, it's perfectly possible to write two different programs that behave exactly the same.

Because there is no single "right" way to write a program, there are only guidelines to help you write programs easily. Ultimately what only matters is that you write a program that works.

When writing any program, there are two, often mutually exclusive goals. First, programmers strive to write programs that are easy to read, understand, and modify. This often means writing multiple instructions that clearly define the steps needed to solve a particular problem.

Second, programmers try to write programs that perform tasks efficiently, making the program run as fast as possible. This often means condensing multiple instructions as much as possible, using tricks or exploiting little-known features that are difficult to understand and confusing even to most other programmers.

In the beginning, strive toward making your programs as clear, logical, and understandable as possible, even if you have to write more instructions or type longer instructions to do it. Later as you gain more experience in programming, you can work on creating the smallest, fastest, most efficient programs possible, but remember that your ultimate goal is to write programs that just work.

Structured Programming

Small programs have fewer instructions so they are much easier to read, understand, and modify. Unfortunately, small programs can only solve small problems. To solve complicated problems, you need to write bigger programs with more instructions. The more instructions you type, the greater the chance you'll make a mistake (called a "bug"). Even worse is that the larger a program gets, the harder it can be to understand how it works so that you can modify it later.

To avoid writing a single, massive program, programmers simply divide a large program into smaller parts called subprograms or functions. The idea is that each subprogram solves a single task. This makes it easy to write and ensure that it works correctly.

Once all of your separate functions work, then you can connect them all together to create a single large program as shown in Figure 1-2. This is like building a house out of bricks rather than trying to carve an entire house out of one massive rock.

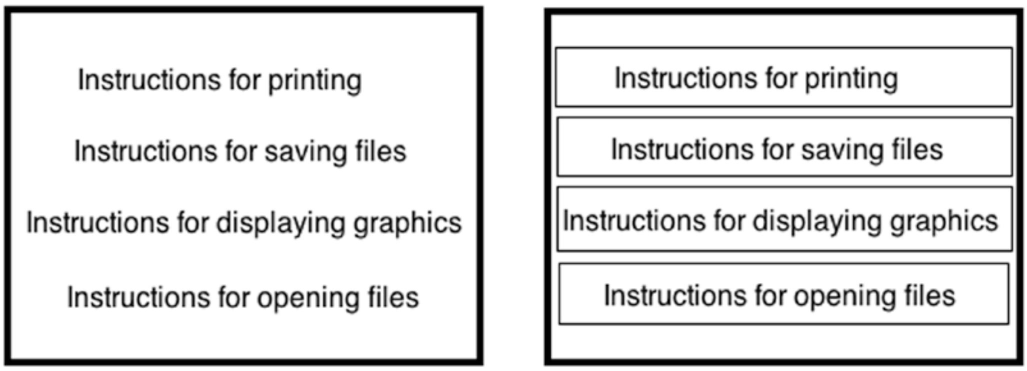

Instructions for printing
Instructions for saving files
Instructions for displaying graphics
Instructions for opening files
Instructions for printing
Instructions for saving files
Instructions for displaying graphics
Instructions for opening files

***Figure 1-2.** Dividing a large program into multiple subprograms or functions helps make programming more reliable*

Dividing a large program into smaller programs provides several benefits. First, writing smaller subprograms is fast and easy, and small subprograms make it easy to read, understand, and modify the instructions.

Second, subprograms act like building blocks that work together, so multiple programmers can work on different subprograms, then combine their separate subprograms together to create a large program.

Third, if you want to modify a large program, you just need to yank out, rewrite, and replace one or more subprograms. Without subprograms, modifying a large program means wading through all the instructions stored in a large program and trying to find which instructions you need to change.

A fourth benefit of subprograms is that if you write a useful subprogram, you can plug that subprogram into other programs. By creating a library of tested, useful subprograms, you can create other programs quickly and easily by reusing existing code, thereby reducing the need to write everything from scratch.

When you divide a large program into multiple subprograms, you have a choice. You can store all your programs in a single file, or you can store each subprogram in a separate file as shown in Figure 1-3. By storing subprograms in separate files, multiple programmers can work on different files without affecting anyone else.

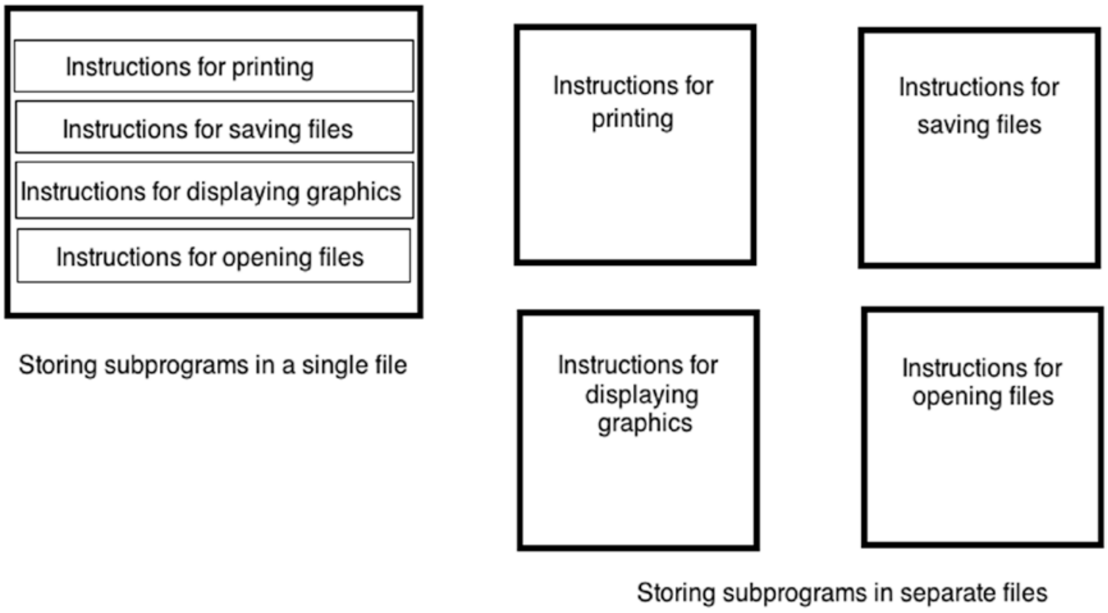

Instructions for printing
Instructions for saving files
Instructions for displaying graphics
Instructions for opening files
Storing subprograms in a single file
Instructions for printing
Instructions for saving files
Instructions for displaying graphics
Instructions for opening files
Storing subprograms in separate files

Figure 1-3. *You can store subprograms in a single file or in multiple files*

Storing all your subprograms in a single file makes it easy to find and modify any part of your program. However, the larger your program, the more instructions you'll need to write, which can make searching through a single large file as clumsy as flipping through the pages of a dictionary.

Storing all of your subprograms in separate files means that you need to keep track of which files contain which subprogram. However, the benefit is that modifying a subprogram is much easier because once you open the correct file, you only see the instructions for a single subprogram, not for a dozen or more other subprograms.

Because today's programs can get so large, it's common to divide its various subprograms in separate files.

Event-Driven Programming

In the early days of computers, most programs worked by starting with the first instruction and then following each instruction line by line until it reached the end. Such programs tightly controlled how the computer behaved at any given time.

All of this changed when computers started displaying graphical user interfaces with windows and pull-down menus so users could choose what to do at any given time. Suddenly every program had to wait for the user to do something such as selecting a menu command or clicking a button. Now programs had to wait for the user to do something before reacting.

Every time the user did something, that was considered an event. If the user clicked the left mouse button, that was a completely different event than if the user clicked the right mouse button. Instead of dictating what the user could do at any given time, programs now had to respond to different events that the user did. Making programs responsive to different events is called event-driven programming.

Event-driven programs divide a large program into multiple subprograms where each subprogram responds to a different event. If the user clicked a menu command, a subprogram would run its instructions. If the user clicked a button, a different subprogram would run another set of instructions.

Event-driven programming always waits to respond to the user's action.

Object-Oriented Programming

Dividing a large program into multiple subprograms made it easy to create and modify a program. However, trying to understand how such a large program worked often proved confusing since there was no simple way to determine which subprograms worked together or what data they might need from other subprograms.

Even worse, subprograms often modified data that other subprograms used. This meant sometimes a subprogram would modify data before another subprogram could use it. Using the wrong data would cause the other subprogram to fail, causing the whole program to fail. Not only does this situation create less reliable software, but it also makes it much harder to determine how and where to fix the problem.

To solve this problem, computer scientists created object-oriented programming. The goal is to divide a large program into smaller subprograms, but organized related subprograms together into groups known as objects. To make object-oriented programs easier to understand, objects also model physical items in the real world.

Suppose you wrote a program to control a robot. Dividing this problem by tasks, you might create one subprogram to move the robot, a second subprogram to tell the robot how to see nearby obstacles, and a third subprogram to calculate the best path to follow. If there was