Develop in Swift Explorations

When you learn to code, you're not only learning the language of technology. You're learning new ways to think and bring ideas to life. And coding with Swift, Apple's powerful, intuitive, and easy-to-learn programming language, provides fun and engaging ways to prepare for the future. Everyone should have the opportunity to create something that can change the world.

Welcome

Great apps help us to create, connect, play, learn, and solve problems. This course is about designing and building excellent apps using Swift, Apple's powerful and intuitive programming language.

Develop in Swift Explorations will help you build a solid foundation in programming skills using Swift as the language. Throughout this course, you'll get practical experience with the tools, techniques, and concepts needed to create a basic iOS app from scratch. You'll also learn user interface design principles, which are fundamental to programming and making great apps. And, you'll explore how technology affects the world, and your role in making technology a force for good.

Using This Book

Develop in Swift Explorations will introduce you to core programming concepts through hands-on activities, Xcode playgrounds, and app projects. You'll build your app development skills as you go, learning about app design as well as programming in Xcode. You'll also have opportunities to think about the impact of computing innovations, and the decisions you make about your actions online and as an app developer.

The book is divided into four units that focus on building your programming skills and understanding of key concepts. Between units, you'll explore a story about a group of students in a TV club. As you follow these episodes, you'll have a chance to think about different aspects of online engagement, how information is shared online, and what kinds of decisions need to be made in a connected world.

By the end of the book, you will have built several simple apps and prototyped an app of your own design. And along the way, you'll have gotten a taste for the world of app development—gaining an appreciation for how technology works, how you can use it to express yourself, and how it can be used to solve problems.

Each unit is divided into four sections:

Get Started

You'll begin by learning the key concepts covered in the unit, exploring how they relate to your everyday experiences, and completing activities that deepen your understanding. By using coding concepts to think about everyday problems, you'll also be learning to think critically, to see the world as a programmer, and to apply computational reasoning.

Play

In this section, you'll apply the key concepts in Xcode playgrounds, where you can experiment with code and see the results immediately. As you complete each activity, you can check your understanding by answering review questions in the book.

You'll also apply your understanding of the unit concepts through fun, creative playground challenges that will help you start thinking about your own app projects.

Later in the book, you'll build simple apps to explore development topics.

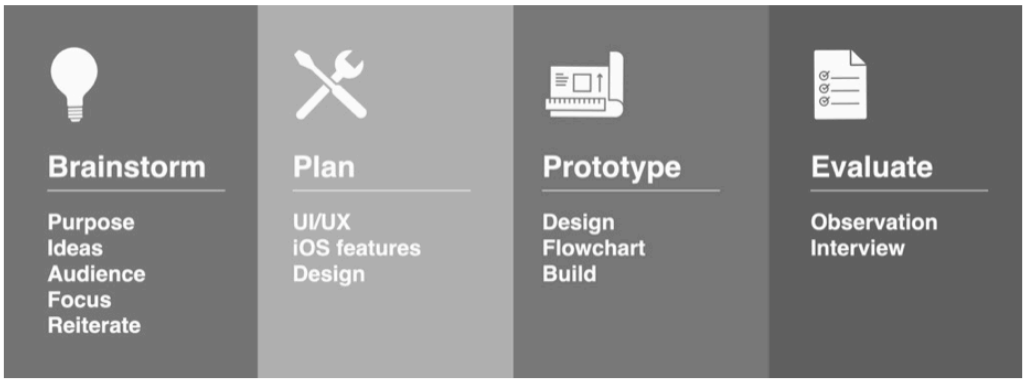
Build

You'll be guided through the steps of building an app in Xcode. For this section, you'll want to keep this book open while you're working in Xcode.

Design

You'll explore the impact of computing innovations and experience the app design process. You'll also consider the choices that app designers and developers make, knowing that their app could impact thousands—even millions—of people.

Using the App Design Journal, you'll get creative. This is your chance to apply your design thinking, develop a great idea, create an interface, and prototype and pitch your concept. You'll learn how to consider different perspectives, and how to use feedback from mentors, friends, and diverse users to improve your app.



Between the units, the episodes tell the story of a group of TV club students as they use technology in their daily lives. Each episode introduces key concepts about technology—and challenges you to analyze the students' choices and to reflect on your own practices.

Tools

In this course, you'll be learning the Swift programming language and working in Xcode. As a result, you'll be joining an important Apple community—more than 20 million developers who write software for iOS, iPadOS, macOS, tvOS, and watchOS in the fast-growing app economy.

Swift

Swift

Swift is a powerful and intuitive programming language created by Apple. It makes programming easier, more flexible, and more fun. But Swift isn't only great for getting you started with coding. It has the power to take you into your future—so you can graduate from writing the simplest program (like "Hello, world!") to creating the world's most advanced software.

Announced in 2014, Swift has become one of the fastest-growing languages in history. It's open-source, which means that anyone—including the app developer community—can contribute directly to its continued development, helping it to evolve and become more capable. And another thing about Swift: You can apply your Swift skills to an ever-broader range of platforms, from mobile devices to desktop computers to the cloud.

To find out more about Swift, visit swift.org. You might also want to download the Swift Programming Language Book.

Xcode

Xcode

Xcode is Apple's Integrated Development Environment (IDE) that's used to create apps for all Apple platforms: iOS, iPadOS, macOS, tvOS, and watchOS. It includes a **source code editor** for writing and managing code, a debugger for diagnosing problems, and a user interface editor—called Interface Builder—for laying out the visual elements of your app and connecting them to your code.

Much of the applied learning in this course takes place in an **Xcode playground**. Playgrounds allow you to write Swift code and immediately see the results in a live preview. Playing with code—and seeing what it does—is a great way to get started coding and to experiment with new ideas.

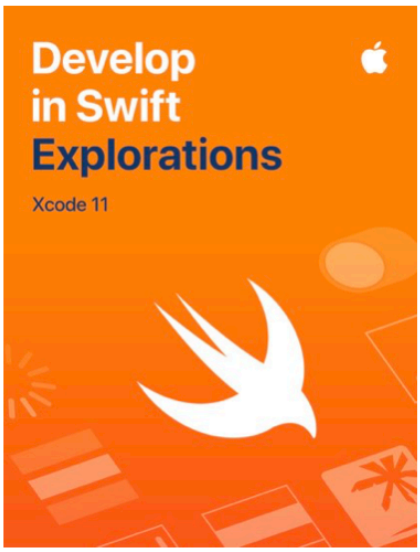
- Explore the Xcode Support page.
- Get inspired watching videos from the Apple Worldwide Developers Conference (WWDC).

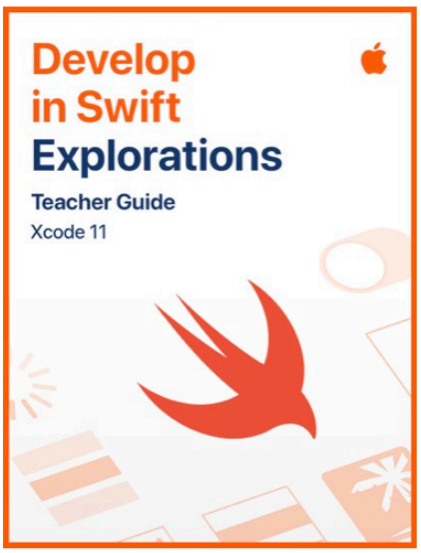
Curriculum Pathway

Develop in Swift curriculum encourages students to solve real world challenges creatively through app development. Students build foundational knowledge with Explorations or Fundamentals courses then progress to more advanced concepts in Data Collections. All courses include free teacher guides to support educators—regardless of experience teaching Swift or other programming languages.

Explorations (180 hours)

Students learn key computing concepts, building a solid foundation in programming with Swift. They'll learn about the impact of computing and apps on society, economy, and our culture while exploring iOS app development. Lessons take students through the app design process: brainstorming, planning, prototyping, and evaluating an app of their own.

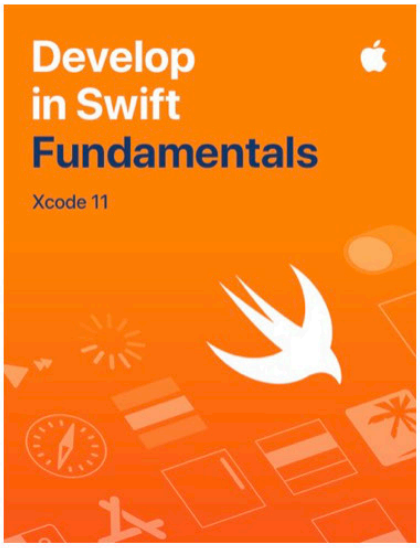


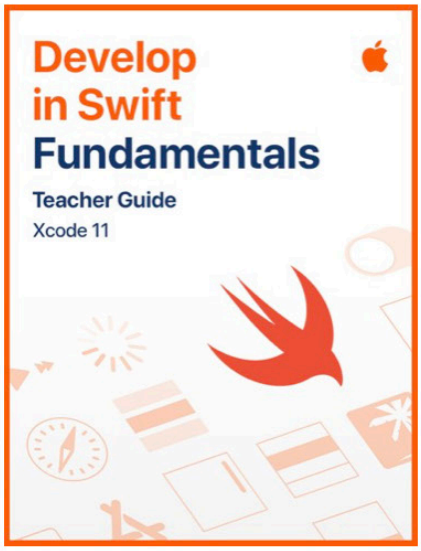


Unit 1: Values
Episode 1: The TV Club
Unit 2: Algorithms
Episode 2: The Viewing Party
Unit 3: Organizing Data
Episode 3: Sharing Photos
Unit 4: Building Apps

Fundamentals (180 hours)

Students build fundamental iOS app development skills with Swift. They'll master the core concepts and practices that Swift programmers use daily and build a basic fluency in Xcode's source and UI editors. Students will be able to create iOS apps that adhere to standard practices, including use of stock UI elements and layouts.

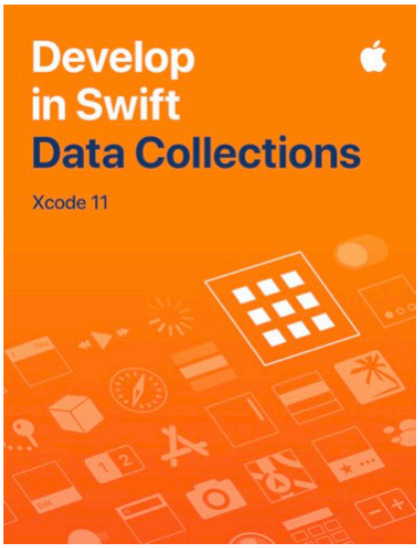


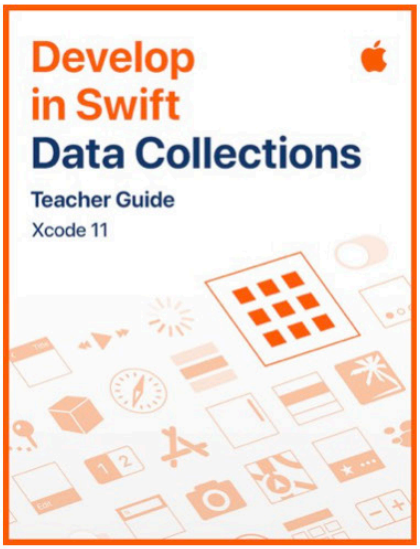


Unit 1: Getting Started with App Development
Unit 2: Introduction to UIKit
Unit 3: Navigation and Workflows
Unit 4: Build Your App

Data Collections (180 hours)

Students extend their knowledge and skill developed in Fundamentals by extending their work in iOS app development creating more complex and capable apps. They'll work with data from a server and explore new iOS APIs that allow for much richer app experiences—including displaying large collections of data in multiple formats. Students experience learning new features of the iOS SDK to continue their app developer journey.

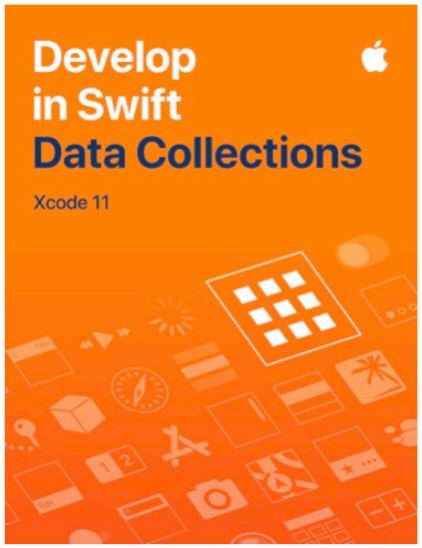


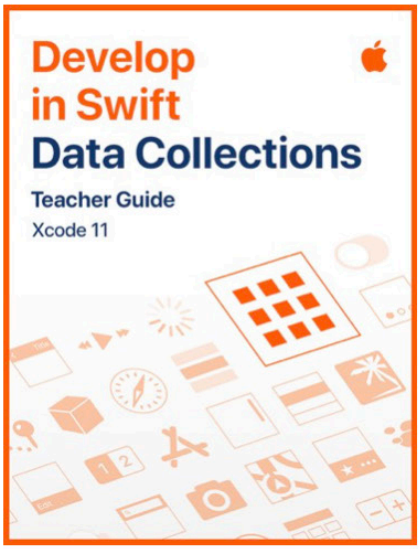


Unit 1: Tables and Persistence
Unit 2: Working with the Web
Unit 3: Advanced Data Display
Unit 4: Build Your App

Data Collections (180 hours)

Students extend their knowledge and skill developed in Fundamentals by extending their work in iOS app development creating more complex and capable apps. They'll work with data from a server and explore new iOS APIs that allow for much richer app experiences—including displaying large collections of data in multiple formats. Students experience learning new features of the iOS SDK to continue their app developer journey.





Unit 1: Tables and Persistence
Unit 2: Working with the Web
Unit 3: Advanced Data Display
Unit 4: Build Your App

Get Set

To complete the lessons in this guide, you'll need the following:

- A Mac running macOS Catalina.
- Xcode 11, available on the Mac App Store.
- Project files for the course. Download the resources for Xcode using the link below.

Download
student materials

Note

To access these materials, you might need to enter the administrator name and password for the Mac you're using.

Get ready to have some fun!

Unit 1

Values

In this unit, you'll learn that all programming is about data, and you'll begin to use numbers and letters to express and manipulate values to solve problems. You'll consider how to name and describe different types of data, and consider how you can select and use values to simulate the real world.

You'll explore these concepts in **Xcode playgrounds**, build a word game in a playground, and get started with Interface Builder to **build** and **run** your own app that displays a photo.

You'll consider how computing innovations have changed people's lives and how technology impacts people differently. At the same time, you'll consider how to reduce bias in technology design, and how to protect your own and others' copyrights. You'll be introduced to the app design process and begin to shape your own app idea.

1.1

Get Started With Values

What Is Programming?

Computers are made by people. They only know what people have told them. And they only do what people tell them to do. When you program a computer, you're telling it what you want it to know and what to do with that information.

Programming is all about **inputs** and **outputs**. Inputs are what people provide to the computer—by tapping with their fingers, typing on a keyboard, talking, taking a photo, and other activities. The program tells the computer how to use that information, for example, sending it somewhere, transforming it, or analyzing it. After working with the inputs, the program produces outputs, such as a drawing, a slide presentation, a chart or graph, or maybe just a big list of numbers. Both inputs and outputs can take many forms: tactile, audio, visual, or text.

Explore This

Consider your favorite app on your phone.

Brainstorm the ways you might provide input to the app. Remember to consider all the sensors in the phone, not just the screen-based inputs.

What forms of output does the app generate?

Values

The inputs and outputs of a program are the information it works with. Another word for information is **data**. Before you started this course, you might have thought about data as lots of numbers. But everything in our world can be captured and described by some form of data—and then analyzed, stored, and shared by computers and the programs that run on them.

Explore This

Take a photo of an everyday object.

Now use Markup to identify as many different physical attributes of that object as you can. Think about components, size, colors, materials, and textures.

You've just captured physical data about the object. What other data might you use to describe the object?

Data is a collection of individual **values**. For example, you can describe the object in your photo by specifying a value, or set of values, for each attribute. All these values, taken together, work as a data set that describes the object.

Like any programming **language**, **Swift** has ways to express and manipulate data. In fact, the term for representing a value—whether it's a simple number or the **result** of a calculation—is **expression**. In this unit, you'll learn how to use values and expressions to explore an idea or to solve a problem.

Naming and Identifiers

Naming is critical to understanding and communicating all the data in the world around us. In everyday life, people use names so that they can refer to the named thing as a whole, without having to think about all its details.

Imagine you're an intergalactic explorer and you arrive on a new planet teeming with alien lifeforms. How would you tell your colleagues which alien has a taste for human or which one is good for a chat? You'd give them names. At first, you might say, "The yellow wobbly one with eyes on stalks and sticky tentacles that lives in the methane swamp." But to save time, you'd probably agree on a name—so you can say, for example, "Look out! Bananacle Monster!" when a tentacle is reaching for your buddy's ankle. Right?

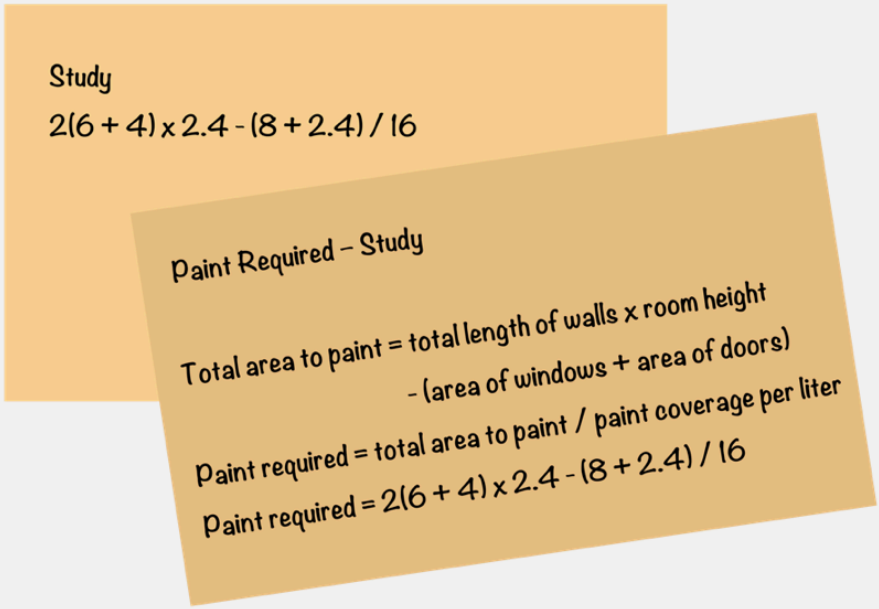
Naming is a form of **abstraction**. When people use a name to refer to something, they can think about the relevant attributes of the named thing (like the fact that it might want to eat you) without having to think about absolutely everything they know about it. Abstraction lets us hide the details of something so they can focus on what's important.

Naming is an important skill in programming too. In programming, you need to create clear, descriptive names—called **identifiers**—that reveal the intent of your **code**. If you choose good identifiers, your code will be easy to read and edit. As a result, other programmers will be able to work with your code and collaborate with you more effectively, and your code will be easier to **debug**. And when you return to code you wrote months earlier, you'll be able to remember your original thought process.

Explore This

Compare these two notes.

Which one will be more useful months later, when you come back to your decorating project and want to buy enough paint to do the whole apartment the same color?



Now imagine you're designing a new game. Draw a quick sketch of your game design, and name as many important components of the game as you can. Include visual elements as well as the elements that determine how the game works.

You can start naming your identifiers with **camel case**, the convention coders use in Swift. In camel case, you begin each identifier with a lowercase letter and then use uppercase letters to start each additional word, for example, speedOfCar, roadFriction, and tireDamage. Use clear, descriptive names that will help you remember your intent when you come back to your design.

Constants and Variables

When you create an identifier—when you call something by a name—in a program, you also need to **declare** whether it's a **constant** or a **variable**. As these terms suggest, the value of a constant doesn't change, while the value of a variable may change.

Constants are useful because their values will never change, which lets programmers make assumptions about the code that uses them. That certainty can reduce the chance of making mistakes in code and can make it easier to find the mistakes that do happen.

Variables change as the program runs, often in response to inputs or actions. For example, the score in a game is a variable, the amount of money in your bank account is a variable, and the time it takes to travel somewhere is a variable. In each case, the value can change even though the type of data remains the same.

Explore This

Imagine your life is a program.

What about you are constants, and what are variables?

For example:

- My eye color is a constant, but my hair length is a variable.
- My date of birth is a constant, but my street address is a variable.

Simulation

Programmers often use **simulations** to replicate real-world systems and to predict their behavior. The creators of a simulation choose which aspects of the system they're interested in measuring, and those decisions impact its design. Think about a crash test dummy. The dummy simulates a human body, but of course it's not an exact replica. In designing a crash dummy, the creators choose which features of the human body are relevant for their test—such as the body's mechanical response to impact, with particular focus on the skull and the neck.

A computer simulation works the same way. By **modeling** a system in a computer simulation, people can see how the system responds to change, and they can test hypotheses. In fact, computer simulations have enabled researchers to predict internal injuries in crashes more accurately than road tests using physical dummies.

To build a simulation, programmers need to consider which aspects of the system we'll model and how those aspects will change. Sound familiar? Simulations are built on variables. In fact, computer simulations are powerful because of the incredible number of values they can compute for each variable and the interaction among many hundreds, thousands, or even millions of variables.

Explore This

Imagine you work as a car safety engineer.

If you were building a crash simulation to determine the effectiveness of airbags, what variables might you consider?

List as many as you can, then combine your list with your classmates to create a comprehensive list of variables.

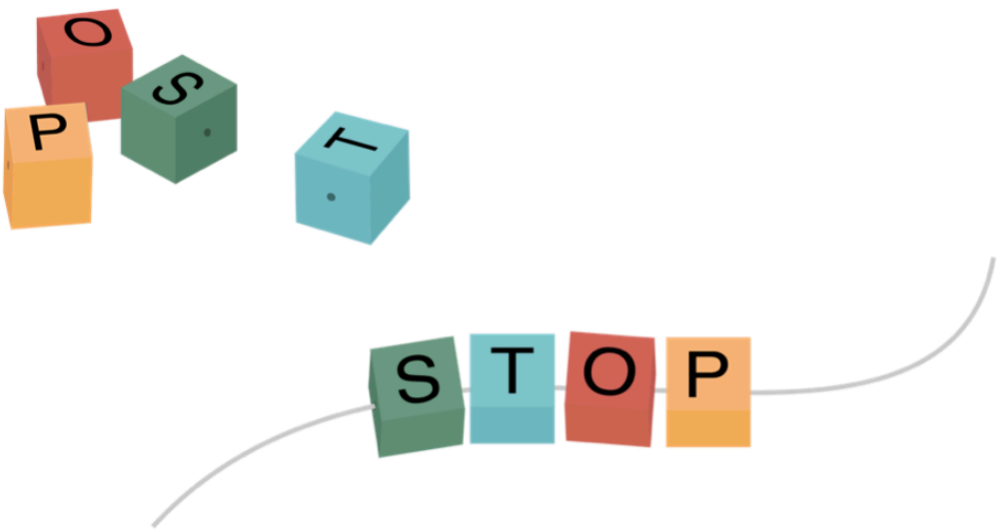
Consider the combined list. Why would running a computer simulation be better than conducting tests with real cars and dummies?

Computer simulations are a great example of abstraction in action. When you select the features of a system to simulate (omitting those that aren't relevant to your task), you're using abstraction. And without all that unnecessary detail, you can work more efficiently and more effectively.

Strings

Look at just about any app—messaging apps, news readers, social media, maps, you name it—and you'll see text. Even camera apps might use text to label buttons and controls.

In most programming languages, including Swift, text values are called **strings** and are composed of **characters**. The name "string" reflects the idea that the characters are in a specific sequence, like beads on a string. This is important because the order of the characters gives a string its meaning. For example, consider these four characters: O, P, S, and T. They could be strung together as POST, TOPS, SPOT, STOP, OPTS, or POTS. Each string has a different meaning, even though it uses the same characters.



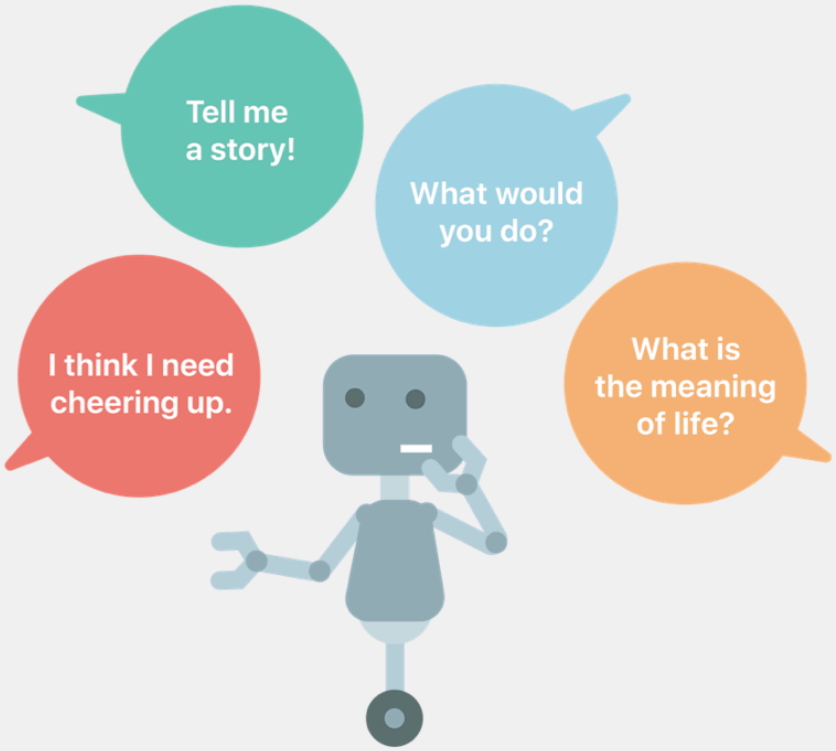
Characters in a string can be letters, numbers, punctuation marks, symbols, and even emoji. Invisible things, like spaces and tabs, are characters too.

Explore This

Imagine you're tasked with creating chatbot responses to the user inputs below.

Chatbots are programs that interact with the user by interpreting strings and responding with strings. You can respond with a maximum of 20 characters—and don't forget to count spaces and punctuation marks!

Your challenge is to design creative, funny responses using all the different types of characters available to you.



1.2

Play With Values

Experiment With Values

A. Playground Basics

Find your way around Xcode playgrounds and learn how to enter and change code in basic programs.

B. Naming and Identifiers

Learn about the importance of naming in programming and build some simple programs to solve problems.

C. Simulation

Learn about the features and applications of simulations, experiment with one.

D. Constants and Variables

Learn how to declare variables and constants, and build a program to keep track of a score.

E. Strings

Learn about strings and how to use them in your code to create a simple game.

Creatively Apply Your Thinking

F. Word Games

Use your knowledge of values, constants, and strings to create and play word games in playgrounds.

Playground Basics

What You'll Build

- A programming setup that you can type in and get answers to basic calculations.

What You'll Learn

- How to type and change code in a playground.
- Where to look for the results of your work.
- How to add notes that will help you remember what your code means.
- What it looks like if something goes wrong.

Key Vocabulary

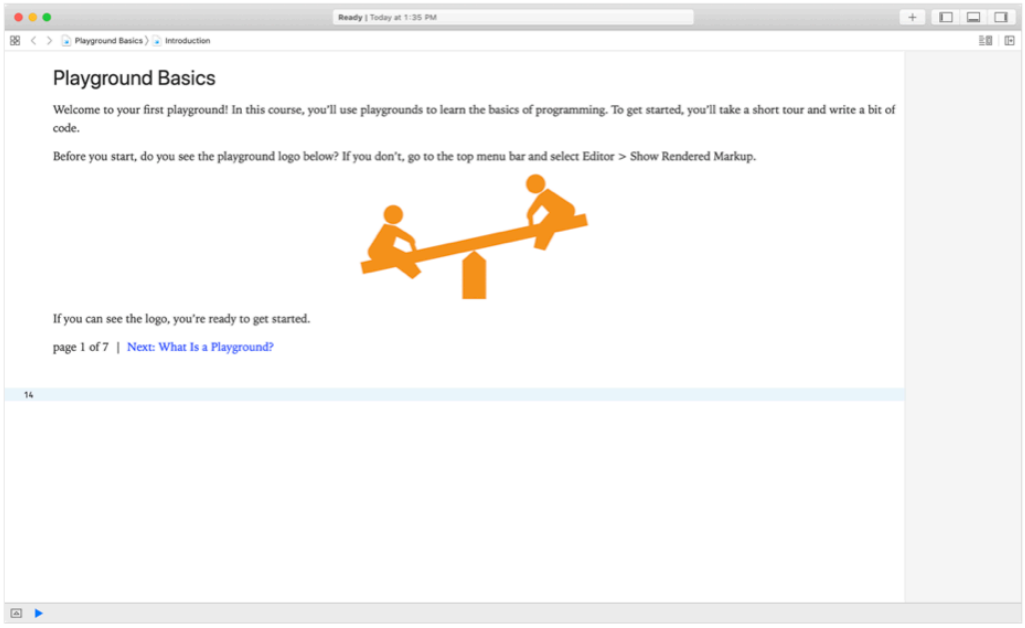
- **Comments**
- **Comment out**
- **Error**
- **Playground**
- **Results sidebar**

Introduction

Playgrounds allow you to experiment with programming ideas without **building** an entire app. You'll write some code, watch it run, think about it a bit, change some lines, and watch it run again. Down the road, you can turn your successful experiments into a fully featured app. But for now, just play—and learn.

Go Build

Open the **`Playground Basics.playground`** file in the folder you downloaded and follow the instructions.



Reflection Questions

What surprised you most about using the playground in this lesson?

What other kinds of calculations could you store and run in a playground?

Summary

You worked on simple problems in a real programming environment and saw the results of the computer doing some work for you. You even saw an **error** from some mistyped instructions and checked where it was coming from so that you could fix the issue. Now you're ready to start one of the most fundamental concepts in programming—using names to label your work. Labeling your work allows you to reuse and remix the results so the computer can solve increasingly difficult problems.

Naming and Identifiers

What You'll Build

- A program to count animals.
- A program to figure out whether your friend's concert will make or lose money.
- A program to decide how you can use your time to have a better morning.

What You'll Learn

- Why writing code to solve a problem is preferable to writing things down and figuring out solutions on your own.

Key Vocabulary

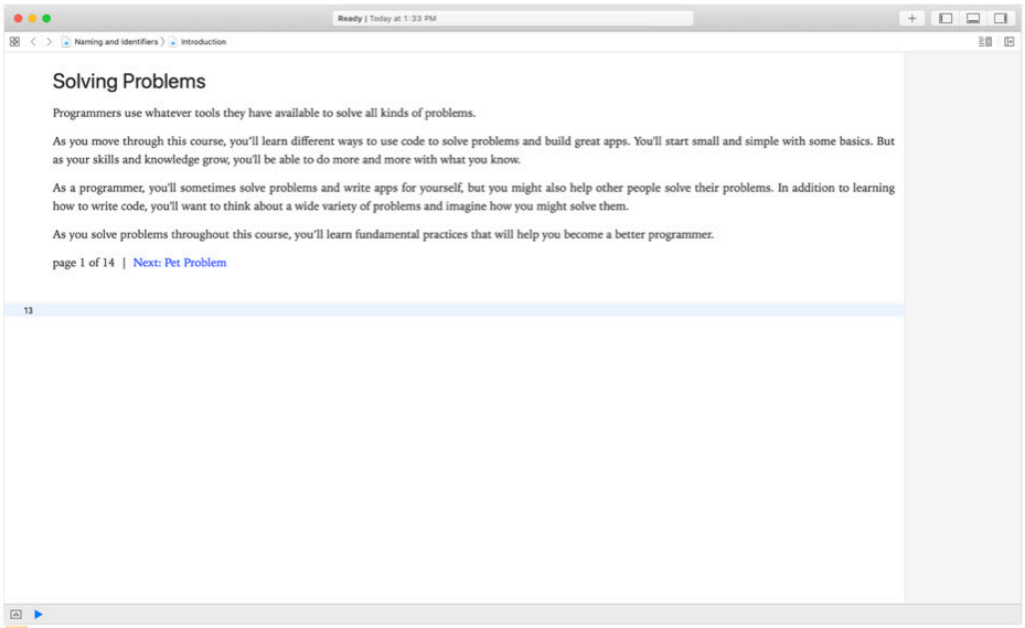
- **Assignment**
- **Constant**
- **Declaration**
- **Identifier**
- **Keyword**

Introduction

People who are new to programming often think that coding is mostly about numbers. In fact, programmers actually spend much more time thinking about names. A programmer who sees an expression like 4 + 5 is usually more interested in what the 4 and 5 mean (are they minutes? marbles? grapefruits?) than that they add up to 9. Becoming skilled at naming will make you a much better programmer.

Go Build

Open the **`Naming and Identifiers.playground`** file in the folder you downloaded and follow the instructions.



Reflection Questions

What might this code be calculating?

```
// Option 1

let total = 4 + (7 * 3)
```

Without good names for these values, a programmer—especially one who's unfamiliar with this code—has no idea what they represent. This code could be calculating anything from the distance of a sidewalk to the area of a rug plus a remnant.

What might this code be calculating?

```
// Option 2

let totalVideoDuration = introDuration +
(clipDuration * numberOfClips)
```

Because the code above uses clear names for its values, anybody who reads it will know it calculates the duration of a video.

Why is it important to use clear names when writing code?

Summary

You might think programming is difficult because you have to keep track of different parts of a program in your head. Seasoned programmers struggle with this, too, but they make life easier by combining calculations into groups, giving names to those groups, and then combining group names into other groups. You've just experienced this in this lesson's playground exercises. You gave names to some ideas, such as calculated costs for concert tickets, and then used those names to do other things, like finding out whether your musical event was a financial success.

You also took a peek at some programming terminology, like *expression* and *declaration*, which you'll be seeing again as you continue to build your knowledge. These terms might seem a bit foreign at first, but don't worry—by the end of the course, you'll be using them to describe your code without even thinking about it.

Many programmers will tell you that naming is one of the hardest problems they face. Creating clear, descriptive names reveals the intent of your code. As a result, other programmers will be able to work with your code and collaborate with you more effectively. It'll also be easier to find the errors in your code. And when you return to code you wrote months earlier, you'll be able to recall your thought process. (Remember to include your future self when you think about those "other programmers.") It's always worth the time to think carefully about names when you code your projects.

Simulation

What You'll Build

- A simulation of an ant colony.

What You'll Learn

- How values are used as parameters to simulations to affect the way they run.
- How to interpret the visual information in a simulation.

Key Vocabulary

- **Simulation**
- **Parameter**
- **Bias**

Introduction

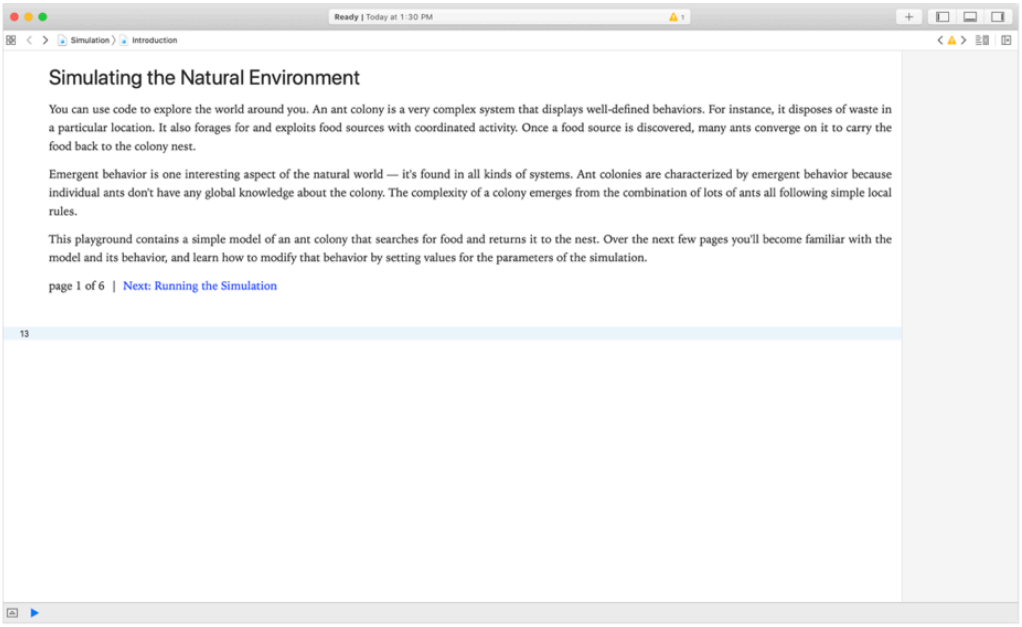
Problems like weather prediction, drug research and design, and rocket dynamics are hard because they involve extreme scales of space or time, vast amounts of data, and potentially dangerous conditions. Computers allow us to simulate complex or dangerous environments to test hypotheses about their behavior.

In this lesson, you'll be experimenting with a simulation of an ant colony to learn how the parameters of a simulation affect behavior in the simulated environment. A simulation tracks large numbers of values as they change over time. In this case, each ant has a location, a speed, and a direction. There's also information in the simulation about the environment around the ants, such as location and amount of food. As the ant simulation runs, you'll see the virtual ants scurry around their environment looking for food to bring to the nest.

Parameters built into a simulation allow experimenters to vary its conditions—for example, how many ants are in the colony. In the playground, you'll be observing the behavior of the simulation and how it's affected by modifying its parameters. Parameters are expressed as variables in the playground—you'll modify them by assigning them different values. Each page will explain the names of the parameters and how they affect the simulation.

Go Build

Open the **`Simulation.playground`** file in the folder you downloaded and follow the instructions.



Reflection Questions

The ant simulation replicates the foraging behavior of a colony. Aspects of an ant colony that aren't directly related to foraging can be discarded as irrelevant. For example, the underground existence of the colony's nest is abstracted away by representing the entrance as a location from which ants emerge and into which they drop off bits of food. Other aspects are present, like the grid of pheromone concentrations and food, but modeled much more simply than in the real world.

What other simplifications can you identify in the ant simulation?

Why do model designers make simplifications?

How do they decide which parts of the real-world system to discard in their design?

How might you extrapolate from hypotheses about a system to the simplifications you'd design into its model?

Simplifications come with tradeoffs. The bias of a simulation is directly related to its simplifications. Simulation bias affects how accurately you can make predictions about the system it represents.

How does the size of environment cells bias the ant simulation?

What other forms of bias exist in the ant simulation?

Summary

You've seen how single values can have a very large impact on a program. Just by changing the value of one of the parameters of the ant colony simulation, you could cause the behavior of the colony to change.

This simulation is a program, its parameters such as `numberOfAnts` are inputs, and its visualization is the output. Inside is some complex code that uses the inputs to model how ants search for food.

The simulation is dynamic—it changes over time. So far, if you wanted your code to use different values, you've had to redo their declarations. In the next lesson, you'll learn how to write code to manage values that can change many times as the program runs.

Constants and Variables

What You'll Build

- A program to help you track scores of a game of darts.

What You'll Learn

- How to make values that are allowed to change over time.
- When you might want values to be changeable and when you might want them to remain constant.

Key Vocabulary

- **Constant**
- **Immutable**
- **Mutable**
- **Variable**

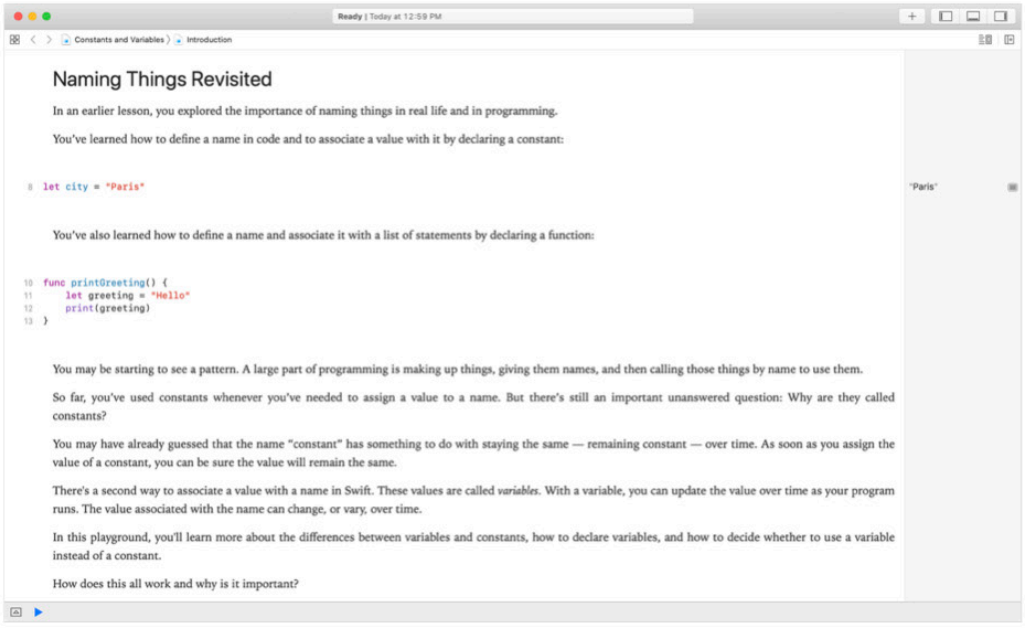
Introduction

When you use an app, you're almost always changing something—for example, the score in a game, the text in a message, or the number of photos in your photo album.

In this lesson, you'll learn how to create values that can change as your program runs. Changeable values allow you to build programs that can adjust to the shifting nature of real-life situations. You'll also learn why it might be important to create values that will never change.

Go Build

Open the **`Constants and Variables.playground`** file in the folder you downloaded and follow the instructions.



Reflection Questions

Look around the room you're in.

Can you divide the things in the room into constants and variables? If you were in the same room tomorrow, or next week, what things would be the same and what will have changed?

Think of the rules of a sport or game. During the course of the game, what remains constant and what varies? Would the game be better if anything about it could change at any time, or do the constants make it a better game?

Think of an app you use and all of the information it holds to do its job. Can you divide the information into variables and constants?

Summary

Introducing variables makes your code more exciting and powerful, but it can also make unexpected things happen if you aren't careful.

Constants and variables are both useful tools, and it's important to know when and where to use each one. You've started to think about how a program models a real-life situation, which aspects of that situation can change, and which should stay the same.

Strings

What You'll Build

- A program to build a fill-in-the-blanks game.
- A lyrics-builder for a rallying song.

What You'll Learn

- How to use text in programming.
- How to use a program to change and combine words.
- What counts as text to a Swift program.

Key Vocabulary

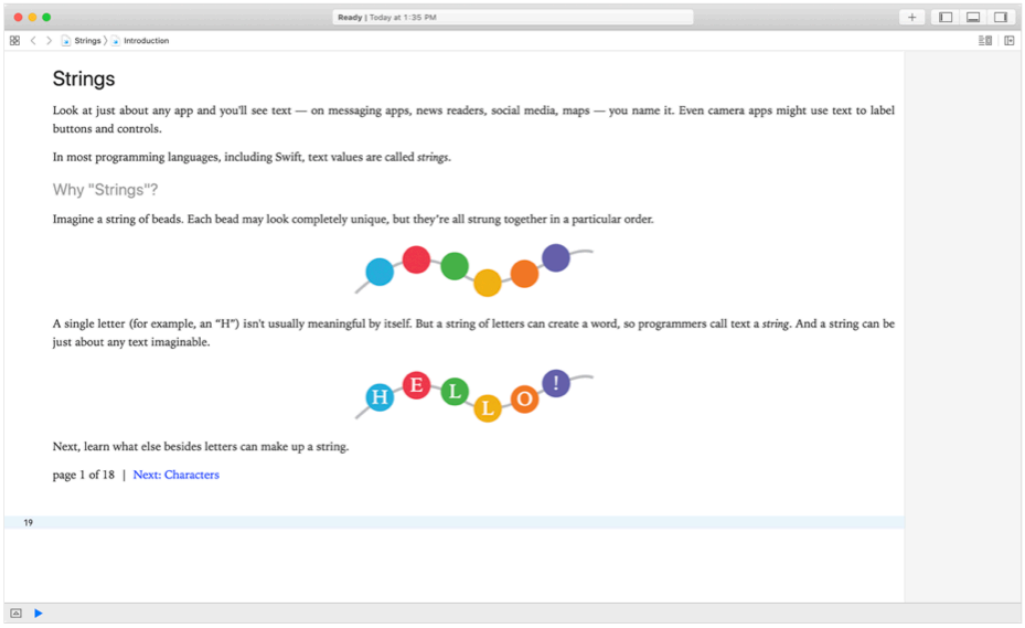
- **Character**
- **Escape character**
- **Escape sequence**
- **Quick Look button**
- **String**
- **String interpolation**
- **String concatenation**
- **Unicode**

Introduction

Numbers are wonderful, especially 0 and 1. But for the most part, people spend a lot more time interacting with text—whether it's sending a message, jotting down a reminder, or reading a post online. In this lesson, you'll learn how computers handle text, and you'll think up some creative string combinations along the way.

Go Build

Open the **`Strings.playground`** file in the folder you downloaded and follow the instructions.



Reflection Questions

What apps do you use that show text?

What apps do you enter text into by typing or dictating?

Do apps you use behave differently depending on what text you enter?

Summary

You've been using playgrounds to experiment with strings and numbers. And you'll be using them to learn other programming concepts as you move through the course. But first, you'll take some time to get creative with the skills you've already learned.

Creatively Apply Your Thinking

Word Games

Introduction

You've learned about strings and how they work together with variables and constants to let you do all sorts of things with text. Coding is impossible without declaring names for values, and strings are an essential kind of data found in almost all code, so these are foundational skills you'll use constantly. Now it's time to use some of your knowledge in the service of creativity, self expression, and humor.

Challenge Yourself

Word games are a fun way to express yourself and amuse others.

In this lesson's playground, you'll challenge your peers to play a word-substitution game that results in funny stories. You'll need to use your knowledge of declaring constants and variables, as well as string escape characters, concatenation and interpolation.

To get started, open `Word Games.playground` from your resource files and follow the instructions.

Have fun!

Go Further

Now that you've built some fun games, think about how you might make them more fun or more interesting.

For example, how could you use variables instead of constants to build a story?

If you declared a number or a string as a variable, how could you modify it as you built the text of the story?

1.3

Build a Photoframe App

Build a PhotoFrame App

What You'll Build

- An app that displays a photo.

What You'll Learn

- How to use Xcode to build and run an app.

Key Vocabulary

- **Asset catalog**
- **Attributes inspector**
- **Editor area**
- **Image view**
- **Project navigator**
- **Simulator**
- **Storyboard**

Introduction

Now that you're getting more comfortable with playgrounds, you might be wondering how to build an app you can use on your iOS **device**, or even your Apple Watch. A lot of moving parts need to work together to make an app run, and Xcode is the best tool for putting them all together.

In this four-part lesson, you'll build PhotoFrame—a simple iOS app that displays a single photo. In the first part, you'll create an app project from scratch. Then you'll use Xcode to explore your project and learn to navigate your coding environment.

In part three, you'll add an image to your project. Finally, you'll create the **user interface** using Interface Builder—a powerful component of Xcode that has a storyboard to lay out your app screens, a library of user interface objects, and many controls and settings for customizing them.

At the end of this lesson, your app will look like this, but it will display a photo of your own choosing.

Part 1

New Project

Going Further With Xcode

So far you've been working in playgrounds, which allow you to focus on code. Apps aren't built just from code, though. There are designs for the look of the screens. There are images, sounds, and fonts. And there are instructions for putting all these parts together. Xcode manages it all in a workspace.

In this part, you'll create an app project and look at it in the Xcode workspace view. If it's a bit overwhelming at first, don't worry. You don't need to know everything about Xcode before you can use it to make apps. You'll be guided through the process step by step, learning as you go along.

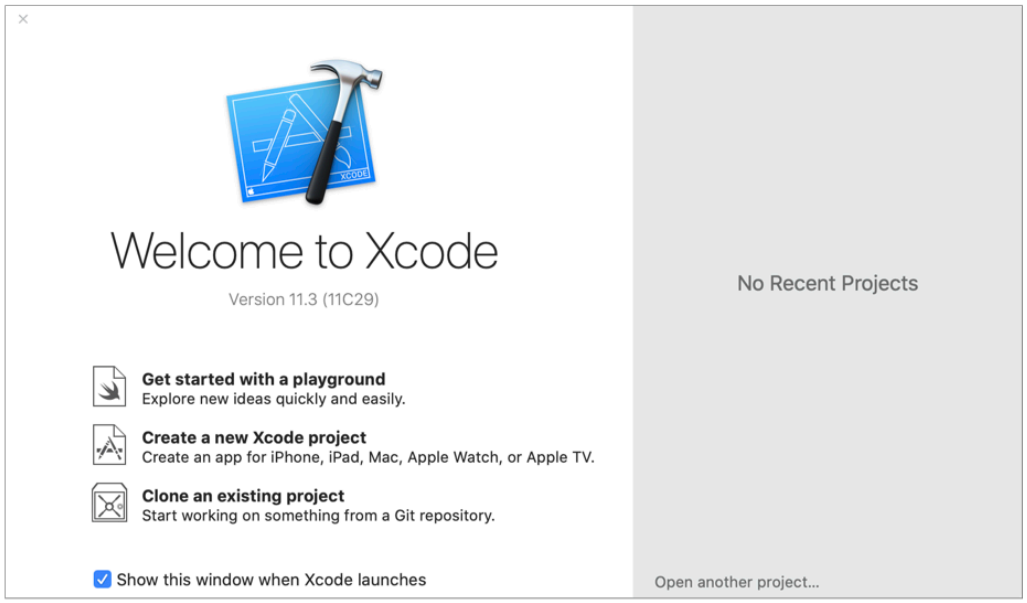
Create a New Project

Xcode includes several built-in app templates for developing common types of iOS apps, such as games, apps with tab-based navigation, and **table view**–based apps. Most of these templates have a preconfigured interface and source code files. For this lesson, you'll start with the most basic template, called Single View Application.

To create a new project:

1. Open Xcode from the /Applications directory. The "Welcome to Xcode" window appears.

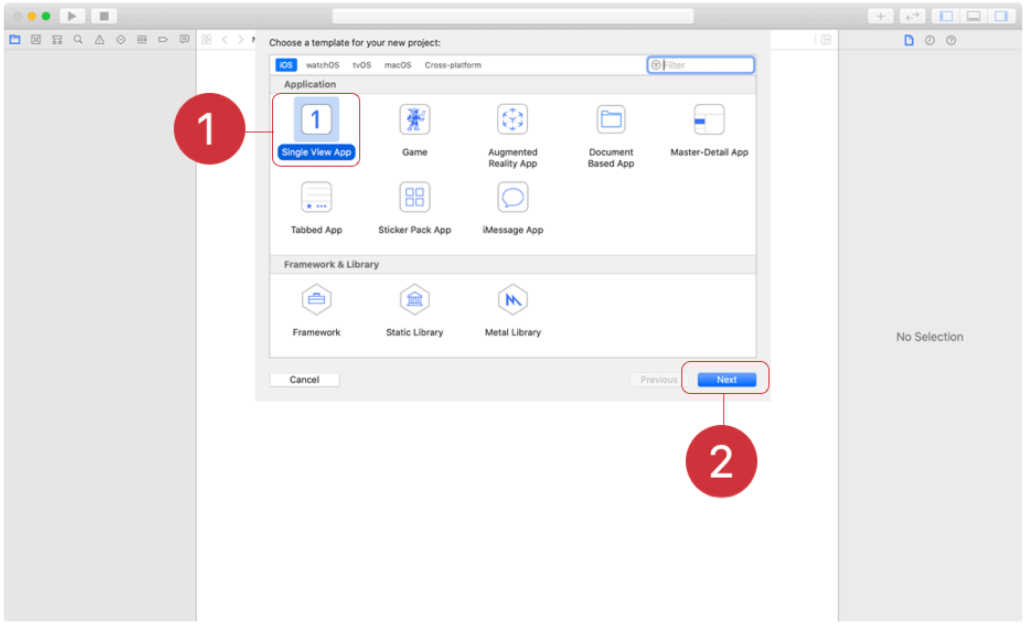
2. If a project window appears instead, it's OK—you probably created or opened a project in Xcode previously. Just use the menu item in the next step to create the project.



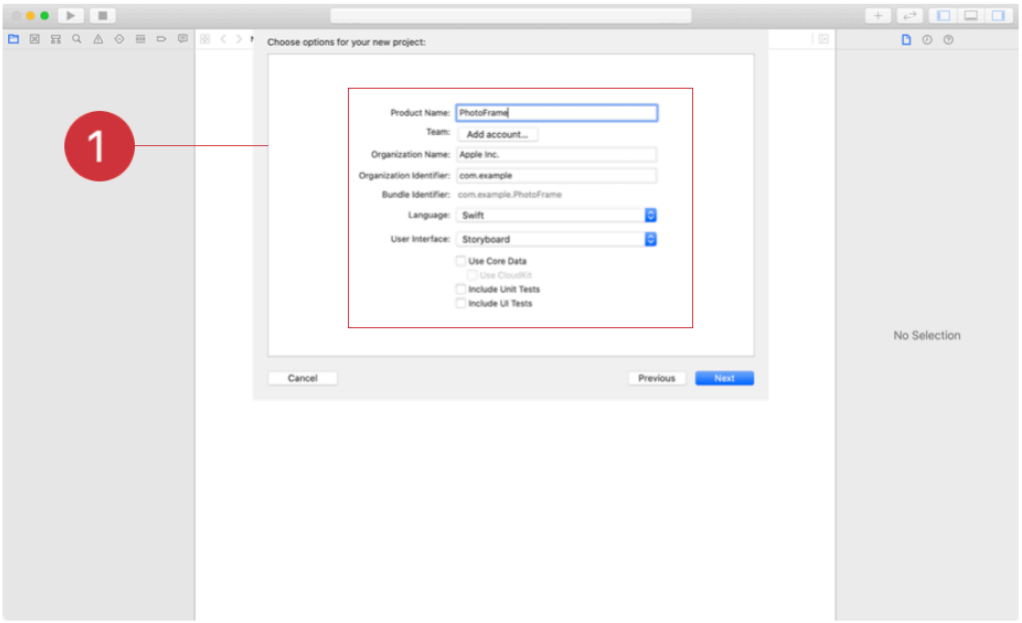
3. In the welcome window, click "Create a new **Xcode project**" (or choose **File > New > Project**). Xcode opens a new window where you can choose a template.

4. Make sure the iOS section is selected at the top of the dialogue, and that you can see the Application heading.

5. In the main area of the dialog, click Single View App[1] and then click Next[2].



6. In the dialog that appears, use the following values to name your app and choose additional options for your project:[1]



Product Name: `PhotoFrame` Xcode uses the product name you entered to name your project and the app.

Team: If you have an Apple developer account, logging in here will allow you to build your app on a device. If you don't have one yet, you can skip this part and test your app in the simulator on your Mac.

Organization Name: The name of your organization or your own name. You can leave this blank.

Organization Identifier: Your organization identifier, if you have one. If you don't, use com.example.

Bundle Identifier: This value is automatically generated based on your product name and organization identifier.

Language: Swift

User Interface: Storyboard

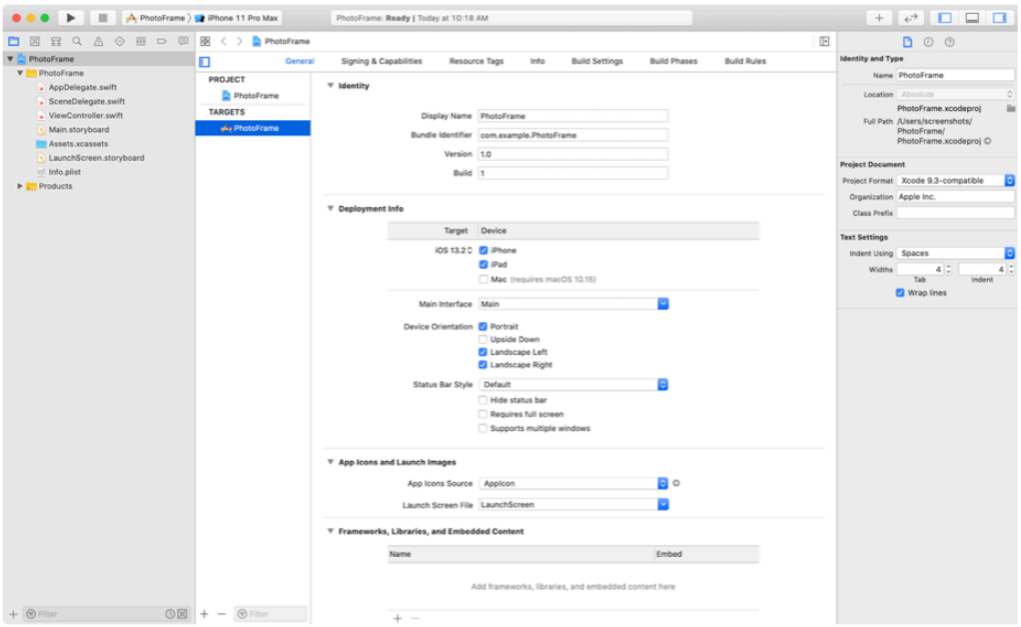
Use Core Data: Unselected

Include Unit Tests: Unselected

Include UI Tests: Unselected

7. Click Next.

8. In the dialog that appears, select a location to save your project and click Create. (If you're not familiar with source control yet, you can uncheck the box that asks if you'd like to create a git repository.) Xcode opens a workspace window showing your new project. You'll see a lot of information on the screen, which you can ignore for now. You'll learn more about these details later.

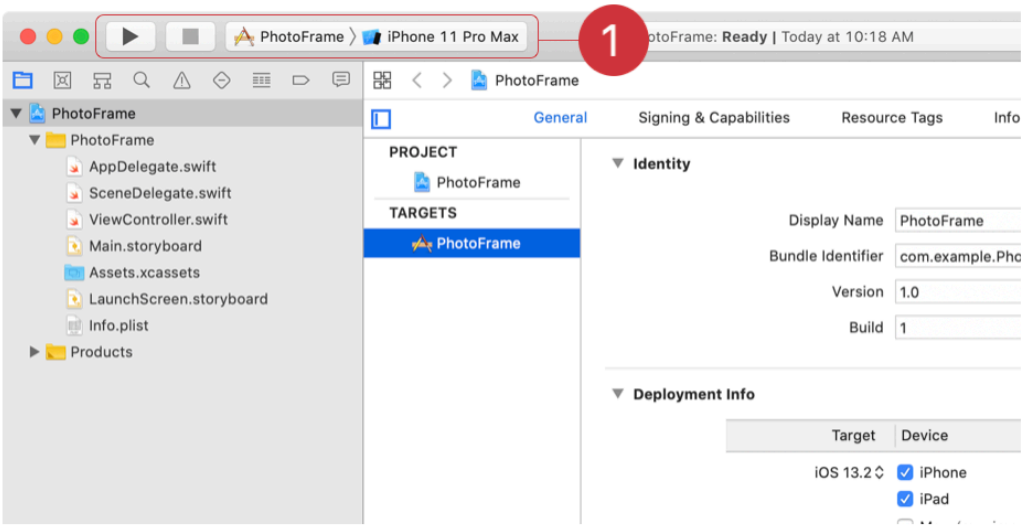


Build and Run Your App

Because you based your project on an Xcode template, the basic app environment is automatically set up for you. So even though you haven't written any code, you can actually build and run the Single View Application template without any additional configuration.

To build and run your app, use the **simulator** app that's included in Xcode. Simulator gives you an idea of how your app would look and behave if it were running on a device. Simulator can model a number of different types of hardware with different screen sizes—including iPhone and iPad—so you can simulate your app on every device you're developing for.

These buttons[1] appear in the **toolbar** at the top of the Xcode window:

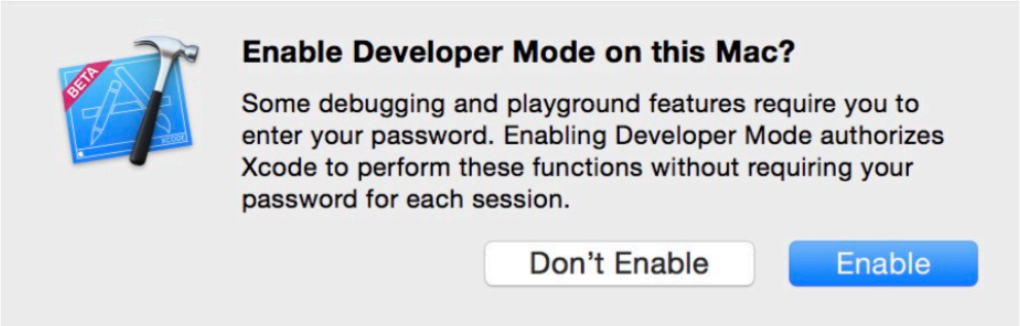


Clicking the Run button ▶ will tell Xcode to build and **run** PhotoFrame on the iPhone 11 Pro Max Simulator.

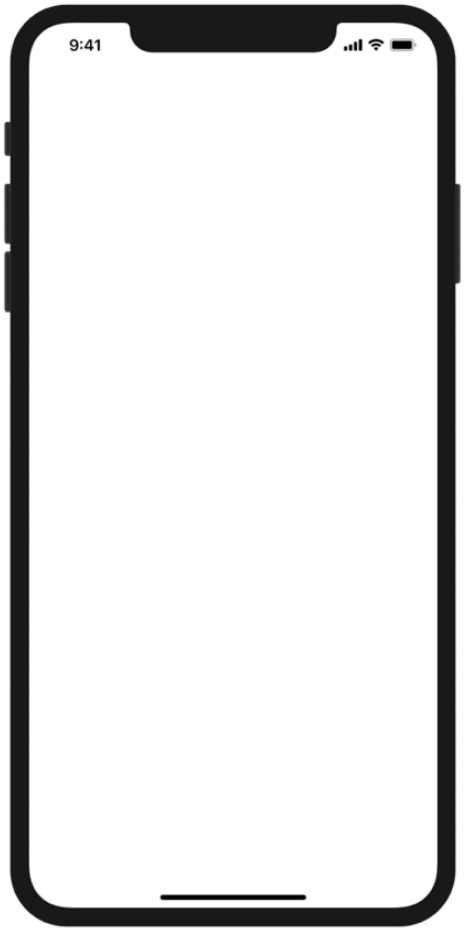
Build means "Assemble all the parts into an app." Run means "Start the app," as if you'd tapped its icon on the Home screen. Follow the steps below to build and run your app.

1. Click the Run button ▶.
2. If you're running an app for the first time, Xcode asks whether you'd like to enable Developer Mode on your Mac. Developer Mode gives Xcode access to certain **debugging** features without requiring you to enter your password each time. Decide whether you'd like to enable Developer Mode, then follow the prompts.

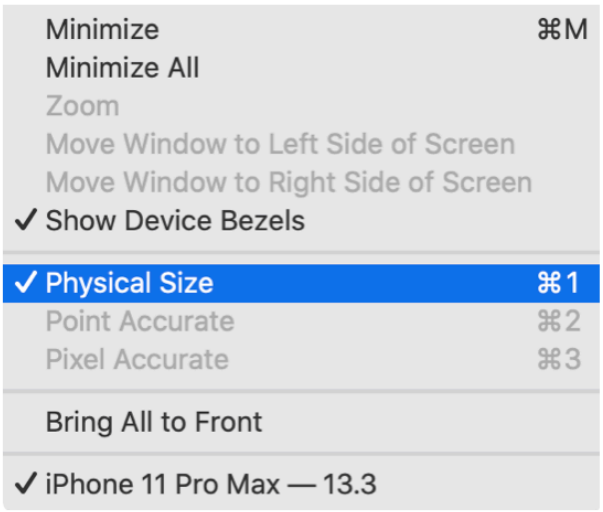
If you choose not to enable Developer Mode, you might be asked for your password later on. The lessons assume Developer Mode is enabled.



You'll see the simulator start up, then show the Home screen, then start your app. Congratulations—you made an app!



The iPhone 11 Pro Max Simulator may appear large on your screen. You can make it smaller by choosing **Window > Physical Size**, or by dragging the corner of the window with your mouse (as you would with any other window). You can also drag it around the screen. You may find it easier to drag and resize the window if you uncheck the **Window > Show Device Bezels** menu item. Otherwise, dragging from the top or bottom of the screen may be interpreted as swipe gestures, such as for invoking Notification Center.



To quit Simulator, you can choose **Simulator > Quit Simulator** press **Command-Q** on your keyboard. But you should keep the iPhone Simulator running while you work on your project, as long as your Mac doesn't slow down too much—you'll save time not having to boot the simulated iPhone each time you build and run your project.

You'll use "Build and Run" a lot while developing your app, not just when you're done. You'll add the image later, but first take a tour of Xcode.

Part 2

Explore Your Project

The first time you open a project in Xcode, you'll see dozens of buttons, panels, and tabs. It can be hard to know which buttons will show you different parts of your app and which will actually change the way your app works.

Take it slow. You'll become familiar with many of the controls in Xcode over time. You don't have to understand exactly what all the buttons and menus do right away. And keep in mind that even the most experienced developers are always learning new tricks and techniques.

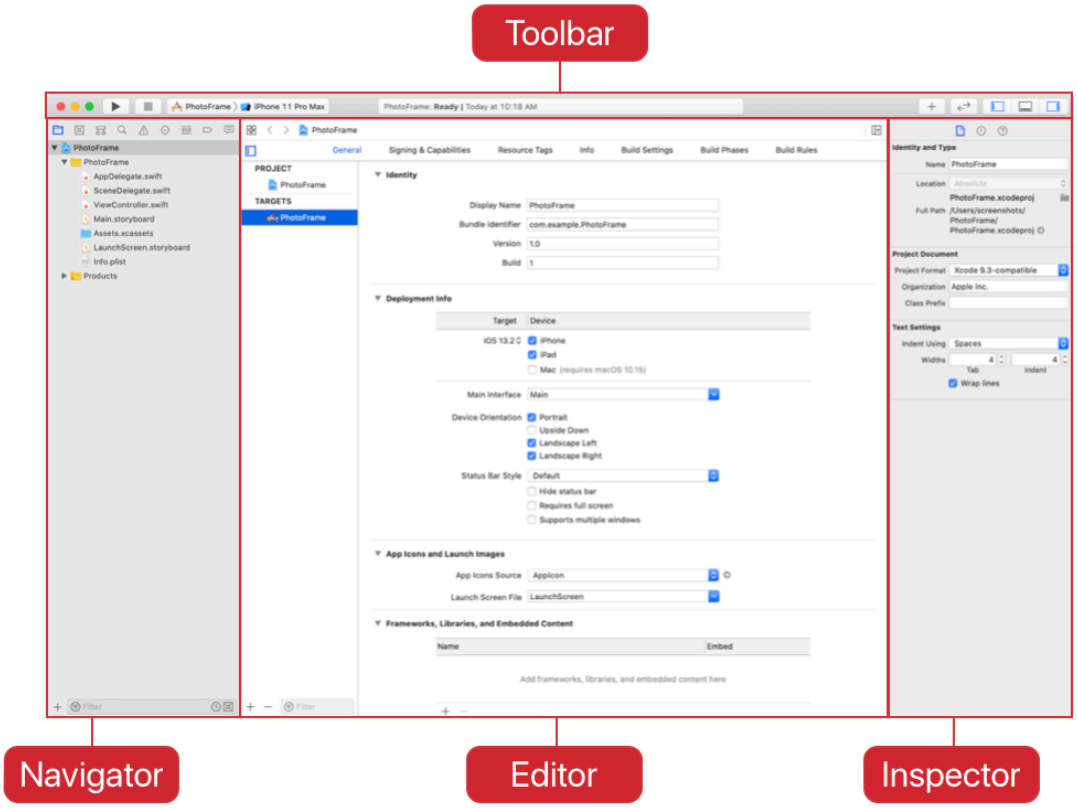
To get started building apps in Xcode, it's enough to:

- Find your code and interface files
- Edit them
- Add more files
- Build and run

You just built and ran in Part 1 and you'll add files in Part 3. But for right now, you'll be exploring the Xcode project you just created. You'll also get a feel for navigating around an Xcode workspace and then learn tips for finding your way back to your code.

Finding Your Way Around Xcode

The Xcode workspace is divided into a few main areas, each with different information about your app and its files. Tap or click the labels to learn more.



Now follow along by opening your PhotoFrame project in Xcode. Does your Xcode screen show the same panels as the image on the previous page?

Just like app developers can configure editor colors just the way they like, they can also change Xcode's layout to display varied facets of their project. Here are some ways you can change Xcode to see different information:

Show and Hide Navigator and Inspector Areas

You may hide both the **navigator area** (on the left of the window) and **inspector area** (on the right of the window) if you'd like more space to edit your code or interface files. You can either do this from the menu options **View > Navigators > Hide Navigator**, and **View > Inspectors > Hide Inspectors**, or from the buttons in the top right corner of the screen[1].

> Exercise
> Practice opening and closing the panels from both the buttons and the View menu.

Show different information in the navigator and inspector areas

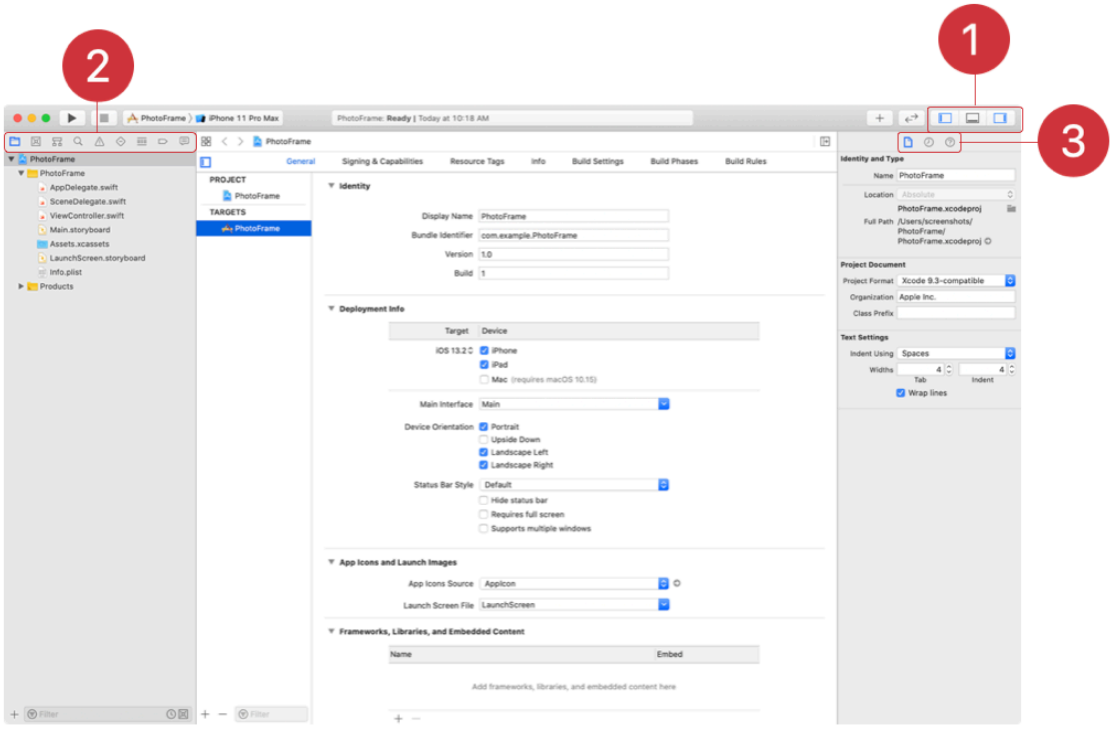
Both the navigator and inspector areas have a row of buttons at the top.

Each icon in the navigator panel[2] represents one of the eight different ways you can explore your project. The highlighted icon tells you which navigator you're currently looking at. You'll usually use the **project navigator** that shows all your files.

The buttons at the top of the inspector area change depending on what kind of file is being shown in the Editor area. Sometimes they include these three icons.[3]

There can be up to seven different inspector icons.

Just like with the navigator area, clicking these different icons changes the display inside the inspector area.



This video shows how to hide and show the navigator area and the inspector area.

Checking Out Your Files

Xcode is a powerful tool not just because it shows so much information about your app, but because it lets you change and update your interface and your code.

The names of your project's files carefully describe what each file is for. The first part of the file's name comes before the period—like "LaunchScreen," "Main," or "ViewController"—and describes the particular file's identity. The second part of the file's name comes after the period, like "swift" or "storyboard." This is called a **file extension** and describes the kind of file, whether Swift code in `.swift` files or user interface in `.storyboard` files. Each file in the project navigator has an icon next to its name.

The main files you'll edit in Xcode are `.swift` code files and `.storyboard` interface files.

`.swift` files These tell your app what to do, and when and how to do it. You edit these files by typing Swift code (which should feel familiar after all your practice in Swift playgrounds).

`.storyboard` files Storyboard files tell your app where to display information on the screen. Xcode opens them in a special environment called Interface Builder. You edit these by clicking, dragging, and choosing options in the inspector area.

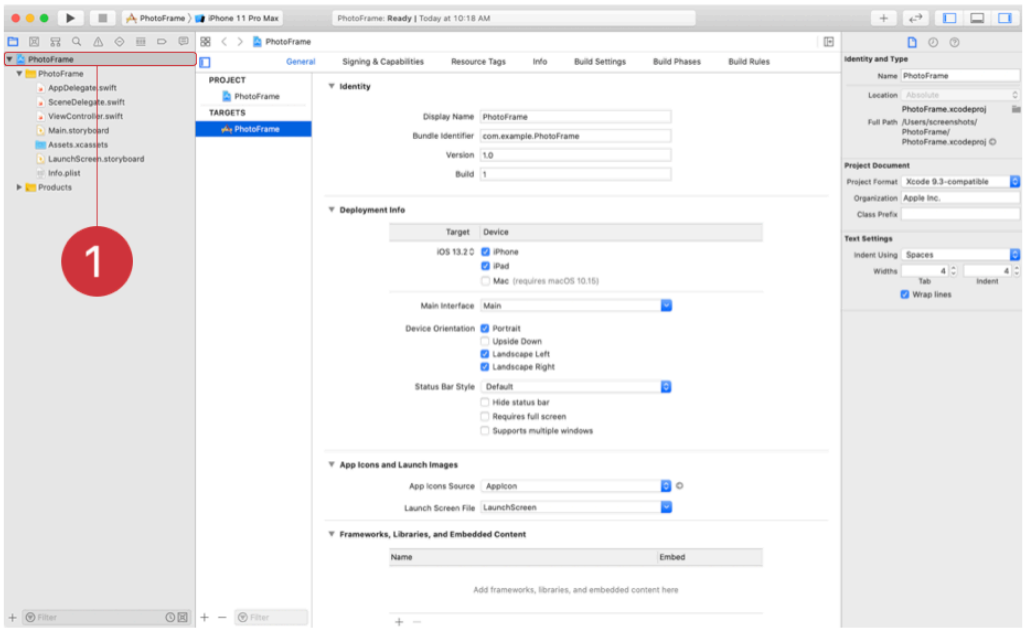
You'll see other file types in the list, too, but you won't interact with these nearly as much:

`.xcassets` This item holds all the images for your app, including the app icon.

`.plist` These files manage your app's setup information. You can open `Info.plist` to take a look around, but you may find it tough to decipher.

Project file

There's a secret other kind of file in your project navigator. At the top of the list of files, the item with your project name has an icon that's different from the other files in your app. It's called the **project file**[1], and you open it by clicking once, just like any other file in your app.



This project file manages information like the display name under your app icon on the Home screen, or whether your app can handle portrait or landscape orientations. During this course, you can safely ignore the project file and the `Info.plist` file.

Getting Back To a Familiar Home

There are lots of ways to configure the Xcode window and navigate. If you ever feel lost or are unsure about how to find your code again, use these tips to get yourself back to familiar ground.

- Make sure the toolbar is visible by choosing **View > Show Toolbar** from the menu options.
- Use the buttons in the top right to make sure the navigator area, inspector area, and **console** aren't covering the editor area.
- Make sure the editor is in standard editor mode from the menu options **Editor > Show Editor Only**. Alternatively, you can click the Adjust Editor Options button (with five horizontal lines) in the top right of the editor area and then choose Show Editor Only from the pop-up menu.
- Show the navigator area with your list of files (**View > Navigators > Show Project Navigator**).
- If you only see the project name and no files underneath it, click the triangle disclosure indicator to reveal the files and directories inside.

Part 3

Add a Photo

Before you edit your user interface in Interface Builder, you need to add a photo or image file to your project. It can be fun to use the Photo Booth app to take a selfie, or you can use something you already have. Make sure you have the image file handy on the desktop.

This video shows you the steps for adding a photo or image file to the Asset Catalog, which are also described in more detail below.

1. **Open the navigator area.** Open the navigator area on the left, with the project navigator showing your list of files. If you're working on a smaller screen, you can close the inspector area on the right.
2. **Select `Assets.xcassets`.** This opens the **Asset Catalog** that Xcode has added to your project. It's the best place to keep all the images used in your app.
3. **Drag in the image file.** This adds the image to your project and gives it a name. The image is now part of your app.

Part 4

Edit the Storyboard

Now you'll edit the content in your **storyboard**, which then will appear in your app.

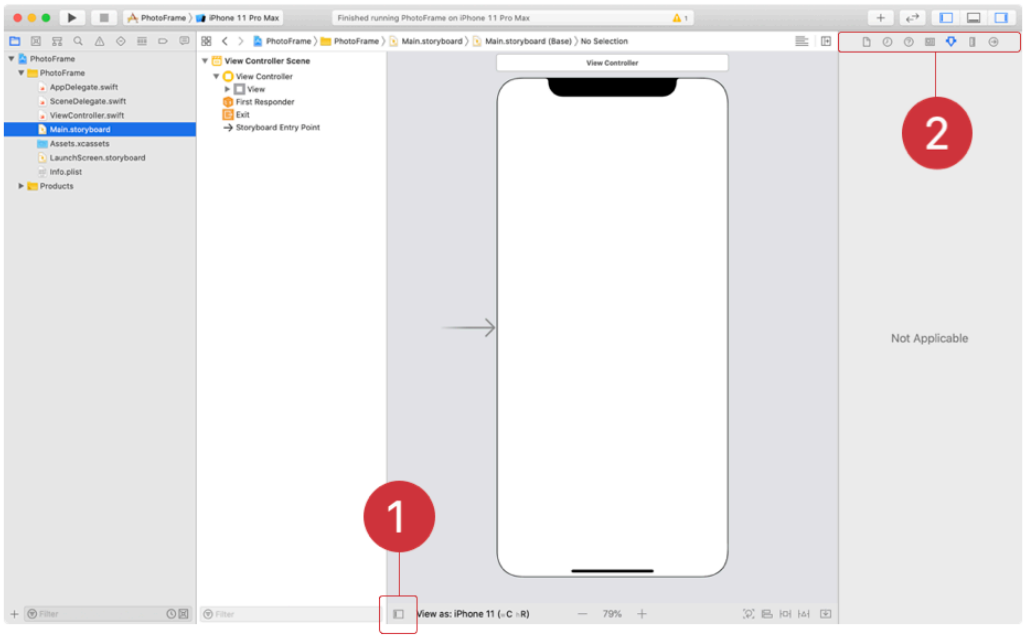
Open the storyboard by selecting `Main.storyboard` from the list of files in the project navigator. The editor area will show an iPhone frame with an empty white screen. This is the main (and only) **scene** in your storyboard.

For the rest of this section, you'll need to keep the inspector area on the right visible. If you need more screen space, you can hide the navigator area on the left.

This video shows you the steps for editing the Storyboard, which are also described in more detail below.

1. **Select the main view.** Click anywhere in the scene to select its main **view**. In the Document Outline to the left, you'll notice that the **View Controller** Scene expands to show the items it contains, and the View item is highlighted. (If you don't see the Document Outline, look for this button at the bottom left of the editor area.)

2. **Show the Attributes inspector.** In the inspector panel on the right, show the **Attributes inspector**[2].



3. **Adjust the background color.** Select the menu next to Background and select Custom. Then choose a new color for the view from the color picker.
4. **Run your app.** When you're happy with your new color, build and run the app again using the Run button in the toolbar. Your app is now a little more interesting.

Adding a Frame

You'll put your photo in a frame by adding another plain view and setting its color, just as you did for the main view in the scene. It's best to add the frame before the photo, because the photo will sit inside it.

This video shows you the steps for adding a View, which are also described in more detail below.

1. **Open the Object library.** To open the object library, choose **View > Show Library** or click the plus button near the top right of the Xcode window. This floating window holds all the objects you can add to a storyboard. Take some time to look at all the different objects—these are the components from which all apps are built. Scroll through until you find the "View" item. It looks like a gray square containing a smaller white square. (If you want to find it quickly, type "UIView" in the search bar at the top to narrow down the selection.)
2. **Drag in a view.** As you drag a view from the Object library, the library window will disappear. Drop it in the view controller.
3. **Position the view.** Move the view to the center of the screen. You'll notice that dotted blue lines will guide you as you drag, and the view will snap to them when it gets close. Then use the resize handles at the corners to make the view large enough to display your photo. As you resize, you'll again notice the guides appear to help you align it.
4. **Set the color.** To change the new view's color, follow the same steps you did for the main view. Since the view is already selected, you can go directly to the Attributes inspector, select the menu next to Background, select Custom, and choose a color from the color picker.

Build and run the app again to see your frame.

Adding and Configuring an Image View

It's time to add your image to the frame. Images are displayed in iOS in **image views**. You'll use a familiar process: dragging a view from the Object library and configuring it in the Attributes inspector.

This video shows you the steps for adding an image view to your app and configuring it to display your image, which are described in detail below.

1. **Open the Object library.** Scroll through the floating window until you find an image view. Alternatively, type in the search bar at the top to narrow down the list.
2. **Drag in an image view.** Drag the image view from the Object library. You'll notice that the main view or the frame view will be highlighted as you drag the image view over them. The highlighted view is the one that will contain the image view when you drop it. Make sure to drop the image view into the frame view.
3. **Resize and position the image view.** Move and resize the image view until the frame behind it is the right width. The guides should help you center it properly. (If the guides are preventing you from getting it just right, you can hold down Command as you drag to prevent snapping and use the arrow keys to fine-tune its position.)

4. **Select the Attributes inspector.** The contents of the Attributes inspector will look a little different now that you've selected an image view. Click the Attributes inspector button, or choose **View > Inspectors > Show Attributes Inspector**.
5. **Choose your image.** Choose the image you added earlier from the pull-down menu at the top of the inspector.
6. **Select Content Mode.** Unless you happened to use an image that's exactly the same shape as the image view, you'll notice that the image doesn't completely fill it. This is because the Content Mode, which you can see under the View heading in the Attributes inspector, is set to Aspect Fit, which fits the image inside the view without stretching or squashing it. Try experimenting with different settings. For example, Aspect Fill will resize the image so there aren't any empty areas, which can cut off portions of the image from the sides or from the top and bottom, but your frame will look the way you intended.

Use the Run button in the toolbar to build and run your app—as you did before—to see your photo display.

Congratulations! Not only have you made an app, you've also made it your own—no other app in the world looks like yours. And you've built it using the same tools that professional developers use every day.

Reflection Questions

What do you want to get out of learning how to code?

- Of those things, which could you do within a playground?
- Which would be more appropriate as a full app?

Summary

Your coding project is made up of many different files, several of which Xcode creates automatically when you make a new project. You edit most of those files right inside Xcode, and when you're ready to view your changes, you tell Xcode to build those files into an app that runs on your simulator or device.

Here's a summary of what you've learned so far about using Xcode to explore your project:

1. The project navigator shows a list of all your code, interface, and configuration files.
2. You can change the editor's contents by choosing different files in the project navigator.
3. When you select different files in the project navigator, you see different options in the inspector area.
4. You can show and hide the navigator and inspector areas as you like.