The **Official** ROBLOX Guide

Coding with Roblox Lua in 24 Hours

Pearson

The Official ROBLOX Guide

Coding with Roblox Lua

in 24 Hours

Pearson

Coding with Roblox Lua in 24 Hours: The Official Roblox Guide

Copyright © 2022 by Roblox Corporation. "Roblox," the Roblox logo, and "Powering Imagination" are among the Roblox registered and unregistered trademarks in the U.S. and other countries. All rights reserved.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-13-682942-3
ISBN-10: 0-13-682942-2

Library of Congress Control Number: 2021948694

ScoutAutomatedPrintCode

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Pearson cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact intlcs@pearson.com.

Editor-in-Chief
Debra Williams Cauley

Acquisitions Editor
Kim Spenceley

Editorial Services
The Wordsmithery LLC

Managing Editor
Sandra Schroeder

Senior Project Editor
Tonya Simpson

Copy Editor
Charlotte Kughen

Indexer
Cheryl Lenser

Proofreader
Sarah Kearns

Editorial Assistant
Cindy Teeters

Cover Designer
Chuti Prasertsith

Compositor
Bronkella Publishing LLC

Graphics Processing
TJ Graham Art

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

Contents at a Glance

Hour 1	Coding Your First Project	1
2	Properties and Variables	17
3	Creating and Using Functions	31
4	Working with Parameters and Arguments	43
5	Conditional Structures	57
6	Debouncing and Debugging	73
7	`while` Loops	91
8	`for` Loops	101
9	Working with Arrays	113
10	Working with Dictionaries	127
11	Client Versus Server	145
12	Remote Events: One-Way Communication	161
13	Using ModuleScripts	173
14	Coding in 3D World Space	187
15	Smoothly Animating Objects	199
16	Solving Problems with Algorithms	209
17	Saving Data	219
18	Creating a Game Loop	229
19	Monetization: One-Time Purchases	243
20	Object-Oriented Programming	259
21	Inheritance	271
22	Raycasting	287
23	Plopping Objects in an Experience: Part 1	297
24	Plopping Objects in an Experience: Part 2	313
Appendix A	Roblox Basics	321
	Index	355

Table of Contents

HOUR 1: Coding Your First Project 1

Installing Roblox Studio 1
Let's Take a Tour 2
Opening the Output Window 5
Writing Your First Script 6
Error Messages 11
Leaving Yourself Comments 12

HOUR 2: Properties and Variables 17

Object Hierarchy 18
Keywords 19
Properties 20
Finding Properties and Data Types 22
Creating Variables 22
Changing the Color Property 25
Instances 26

HOUR 3: Creating and Using Functions 31

Creating and Calling Functions 31
Understanding Scope 33
Using Events to Call Functions 33
Understanding Order and Placement 36

HOUR 4: Working with Parameters and Arguments 43

Giving Functions Information to Use 43
Working with Multiple Parameters and Arguments 45
Returning Values from Functions 49
Returning Multiple Values 50
Returning Nil 51
Dealing with Mismatched Arguments and Parameters 51
Working with Anonymous Functions 52

HOUR 5: Conditional Structures 57

`if/then` Statements 58

`elseif` 62

Logical Operators 62

`else` 63

HOUR 6: Debouncing and Debugging 73

Don't Destroy, Debounce 73

Figuring Out Where Things Go Wrong 82

HOUR 7: `while` Loops 91

Repeat Forever, `while true do` 91

Some Things to Keep in Mind 92

`while` Loops and Scope 98

HOUR 8: `for` Loops 101

How `for` Loops Work 102

Nested Loops 109

Breaking Out of Loops 110

HOUR 9: Working with Arrays 113

What Are Arrays? 113

Adding Items Later 114

Getting Information from a Specific Index 114

Printing an Entire List with `ipairs()` 115

Folders and `ipairs()` 116

Finding a Value on the List and Printing the Index 121

Removing Values from an Array 122

Numeric `for` Loops and Arrays 123

HOUR 10: Working with Dictionaries 127

Intro to Dictionaries 127

Adding and Removing from Dictionaries 130

Removing Key-Value Pairs 130

Working with Dictionaries and Pairs 132

Returning Values from Tables 133

HOUR 11: Client Versus Server 145
Understanding the Client and the Server 145
Working with GUIs 146
Understanding RemoteFunctions 149
Using RemoteFunctions 149

HOUR 12: Remote Events: One-Way Communication 161
Remote Events: A One-Way Street 161
Communicating from the Server to All Clients 162
Communicating from the Client to the Server 165
Communicating from the Server to One Client 170
Communicating from Client to Client 171

HOUR 13: Using ModuleScripts 173
Coding Things Just Once 173
Placing ModuleScripts 174
Understanding How ModuleScripts Work 174
Naming ModuleScripts 174
Adding Functions and Variables 175
Understanding Scope in ModuleScripts 176
Using Modules in Other Scripts 177
Don’t Repeat Yourself 183
Dealing in Abstractions 183

HOUR 14: Coding in 3D World Space 187
Understanding X, Y, and Z Coordinates 187
Refining Placement with CFrame Coordinates 189
Offsetting CFrames 191
Adding Rotations to CFrames 191
Working with Models 192
Understanding World Coordinates and Local Object Coordinates 193

HOUR 15: Smoothly Animating Objects 199
Understanding Tweens 199
Setting `TweenInfo` Parameters 201
Chaining Tweens Together 205

HOUR 16: Solving Problems with Algorithms 209

Defining Algorithms 209
Sorting an Array 210
Sorting in Descending Order 212
Sorting a Dictionary 213
Sorting by Multiple Pieces of Information 216

HOUR 17: Saving Data 219

Enabling Data Stores 219
Creating a Data Store 220
Using Data in the Store 220
Limiting the Number of Calls 225
Protecting Your Data 225
Saving Player Data 226
Using `UpdateAsync` to Update a Data Store 226

HOUR 18: Creating a Game Loop 229

Setting Up Game Loops 229
Working with BindableEvents 230

HOUR 19: Monetization: One-Time Purchases 243

Adding Passes to Your Experience 243
Configuring the Pass 246
Prompting In-Game Purchases 247

HOUR 20: Object-Oriented Programming 259

What Is OOP? 259
Organizing Code and Projects 259
Making a New Class 260
Adding Class Properties 261
Using Class Functions 263

HOUR 21: Inheritance 271

Setting Up Inheritance 272
Inheriting Properties 274
Working with Multiple Child Classes 277

Inheriting Functions 278
Understanding Polymorphism 278
Calling Parent Functions 282

HOUR 22: Raycasting 287
Setting Up the Function to Raycast 287
3D Math Trick: Getting the Direction 289
Setting Raycast Parameters 290
3D Math Trick: Limit Direction 293

HOUR 23: Plopping Objects in an Experience: Part 1 297
Setting Up the Object 298
Creating a Plop Button 302
Tracking Mouse Movements 303
Previewing the Object 307

HOUR 24: Plopping Objects in an Experience: Part 2 313
Detecting Mouse Input 314
Sending a Message to the Server 316
Getting the Message 317

APPENDIX A: Roblox Basics 321
Keywords 322
DataType Index 322
Operators 324
Naming Conventions 325
Animation Easing 325
Possible Solutions to Exercises 326

Index 355

About the Author

Genevieve Johnson is the senior instructional designer for Roblox, the world's largest user-generated social platform for play. In her role, she oversees creation of educational content and advises educators world-wide on how to use Roblox in STEAM-based learning programs. Her work empowers students to pursue careers as entrepreneurs, engineers, and designers. Prior to Roblox, Johnson was educational content manager for iD Tech, a nationwide tech education program that reaches more than 50,000 students yearly, ages 6-18. While at iD Tech, she helped launch a successful all-girls STEAM program, and her team developed educational content for more than 60 technology-related courses, teaching a variety of subjects from coding to robotics and game design.

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you email, please be sure to include this book's title and author as well as your name, email address, and phone number. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: community@informit.com

Reader Services

Register your copy of *Roblox Game Development in 24 Hours* at www.informit.com/register for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account.* Enter the product ISBN (9780136829423) and click Submit.

*Be sure to check the box that you would like to hear from us to receive exclusive discounts on future editions of this product.

This page intentionally left blank

HOUR 1
Coding Your First Project

What You'll Learn in This Hour:

- ▶ Why Roblox and Lua are a perfect combination
- ▶ What Roblox Studio's main windows are
- ▶ How to say "Hello" to the world with your first code
- ▶ How to make a part explode
- ▶ How to check for errors
- ▶ How to leave a comment

Roblox is the world's most popular game development platform. All types of people come together to create amazing virtual experiences: artists, musicians, and—you guessed it—coders. Coding is what allows players to interact with the world that they see.

In Roblox, the coding language used is Lua. Lua is one of the easiest coding languages to learn, and when used with Roblox Studio, you can see the results of your code fast. For example, want to create an enormous explosion with a massive blast radius? You can do that with just a couple of lines of Lua.

Roblox Studio is the tool in which all Roblox games are created, and when paired with Lua, it offers seamless access to multiplayer servers, physics and lighting systems, world-building tools, monetization systems, and more. And even though Roblox provides the environment in which your program runs, you control the vision. You are the creator and artist. Roblox gives you the canvas and paints, and Lua the brushes and actions. But *you*, with some well-placed dabs of code, get to create your masterpiece. This first hour covers how to set up Roblox Studio, make your first script, and test your code.

Installing Roblox Studio

Before you get started, make sure you have Roblox Studio installed. It runs on Windows and MacOS, and you can grab a copy at https://roblox.com/create. Click Start Creating to begin. You'll need to create a Roblox account if you don't yet have one (see Figure 1.1).

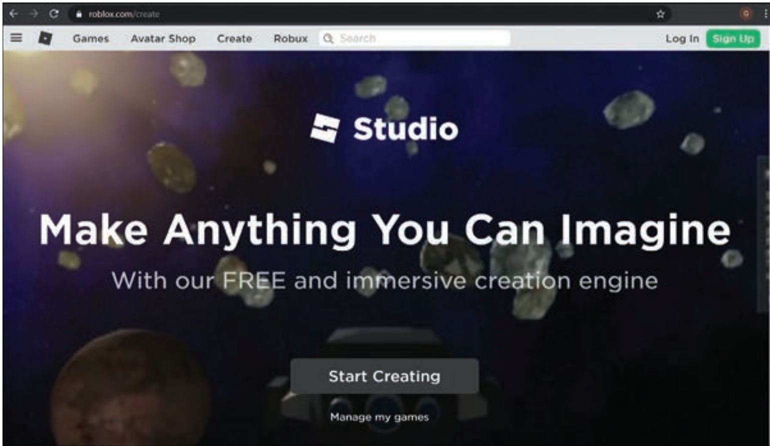


FIGURE 1.1
You need an account to use Roblox Studio. It's free and just a quick sign-up away.

Let's Take a Tour

Roblox Studio provides everything you need to create games. It includes assets such as character models, items to put in the world, graphics for the sky, soundtracks, and more.

Go ahead and launch Roblox Studio to see the window shown in Figure 1.2. Enter the login information for the account you created when you signed up on the Roblox website and click Log In.

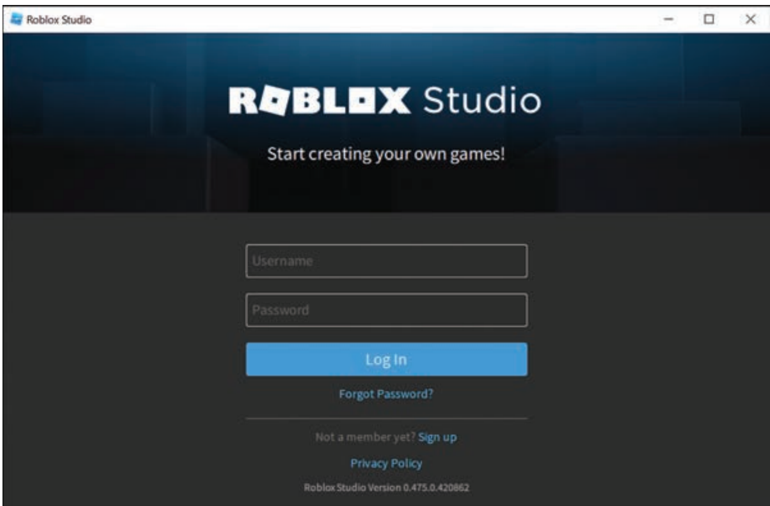


FIGURE 1.2
Enter your normal Roblox account information.

When you first open up Studio, you see templates. These are starting places you can use for your experiences. The simplest starting point for any project is the *Baseplate* template. Click on the Baseplate template, as shown in Figure 1.3.

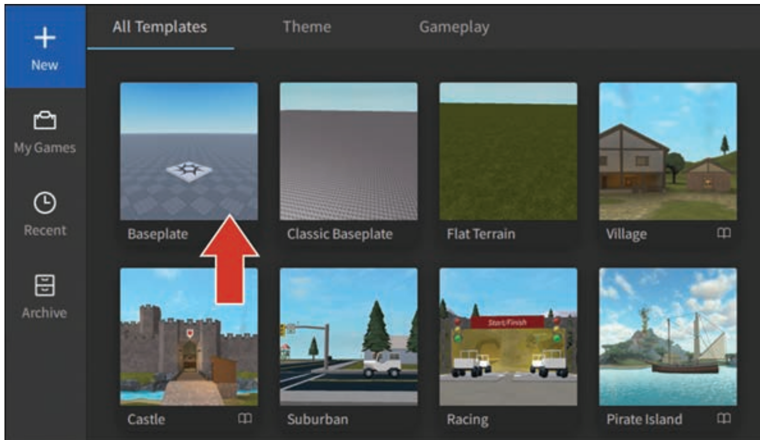


FIGURE 1.3
Studio offers template places you can use as starting points.

Let's start with a quick overview of the main parts of the screen in Figure 1.4, and then move straight into your first line of code:

1. The offerings in the Toolbar ribbon change according to the menu tab you've selected.
2. The Toolbox contains existing assets to add to your game. You can also create your own assets through a 3D modeling program such as Blender3D, and Studio includes a set of mesh-editing tools to customize the 3D models already available.
3. The 3D Editor provides a view of the world. Hold your right mouse button to turn the view, and use the WASD keys to reposition the camera. Table 1.1 describes the different controls to move the camera.
4. The Explorer window provides convenient access to every key asset or system in the game. You use this to insert objects into your experience.
5. Use the Properties window to make changes to objects in the game, such as color, scale, value, and attributes. Select an object in the Explorer to see available properties.

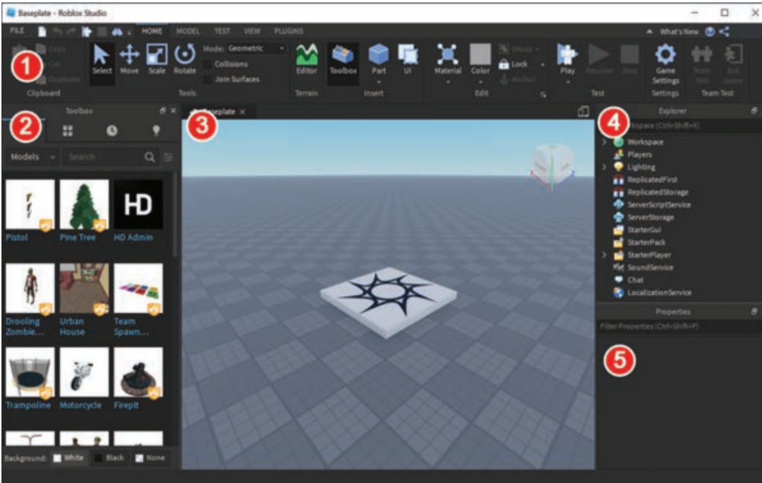


FIGURE 1.4
There are a number of panels, buttons, and lists in the Studio, and you'll quickly become familiar with them.

TABLE 1.1 Camera Controls

Key	Movement
W A S D	Move the camera up, left, down, or right
E	Move the camera
Q	Lower the camera down
Shift	Move the camera slower
Right mouse button (hold and drag mouse)	Turn the camera
Middle mouse button	Drag the camera
Mouse scroll wheel	Zoom the camera in or out
F	Focus on selected object

There are numerous ways to configure this main screen, including hiding different sections, rearranging their positioning to be more convenient, and changing their size.

Roblox Studio is a very complete game development environment that goes well beyond Lua. It's a big topic on its own, so you may want to check out our other book, *Roblox Game Development in 24 Hours*, for help.

Opening the Output Window

The Output window in Studio isn't open by default, but you need this before you continue so that you can see errors and messages that are related to your code.

Use the following steps to display the Output window:

1. Click the View tab (see Figure 1.5). If you ever close a window and need to reopen it, you can find it here.

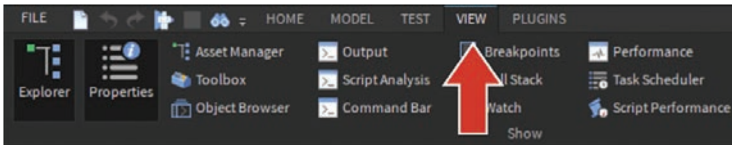


FIGURE 1.5
Use the View tab to control which windows are open.

2. Click Output (see Figure 1.6) to display the Output window at the bottom of your screen, as shown in Figure 1.7.

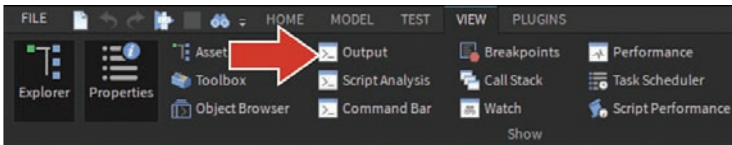


FIGURE 1.6
Click the Output option to open the Output window.

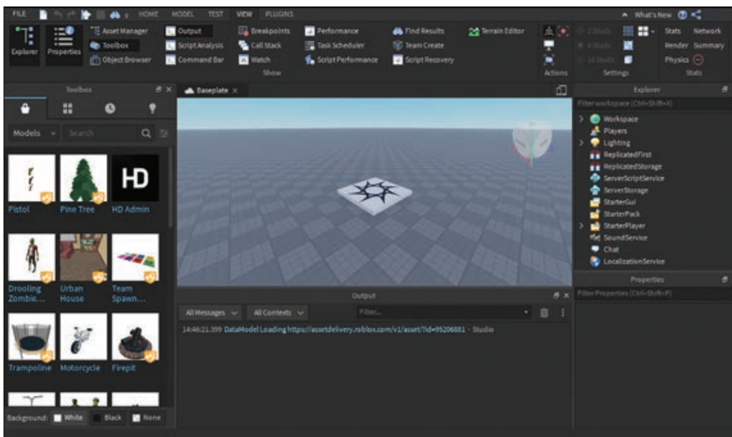


FIGURE 1.7
The Output window opens beneath the 3D Editor.

Writing Your First Script

On to coding! You need something to hold your code, and that's a script. You can insert scripts directly into objects within the world. In this case, you're inserting a script into a part.

Insert a Script into a Part

A part is the basic building block of Roblox. Parts can range in size from very tiny to extremely large. They can be different shapes such as a sphere or wedge, or they can be combined into more complex shapes.

1. Return to the Home tab and click Part (see Figure 1.8). The part appears in the 3D Editor at the center of your camera view.

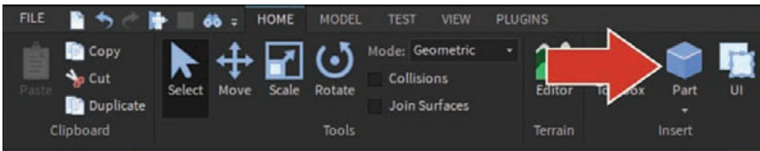


FIGURE 1.8
Click Part on the Home tab to insert a part.

2. To add a script, in Explorer, hover over the part and click the + symbol, and then select Script from the drop-down menu (see Figure 1.9).

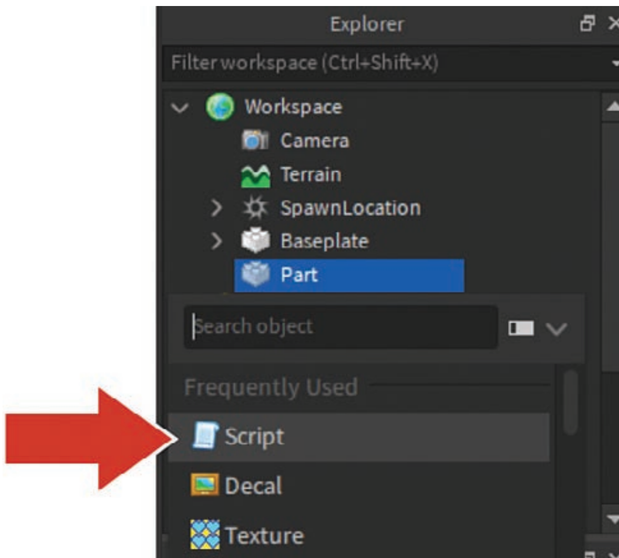


FIGURE 1.9
You use Explorer to insert a script into the part.

TIP

Finding Items Quickly

Typing the first letter (*S*, in this case) or two of the items you are adding filters the list so you can locate that item quickly.

The script automatically opens. At the top, you see words familiar to any coder: `"Hello world!"` (see Figure 1.10).

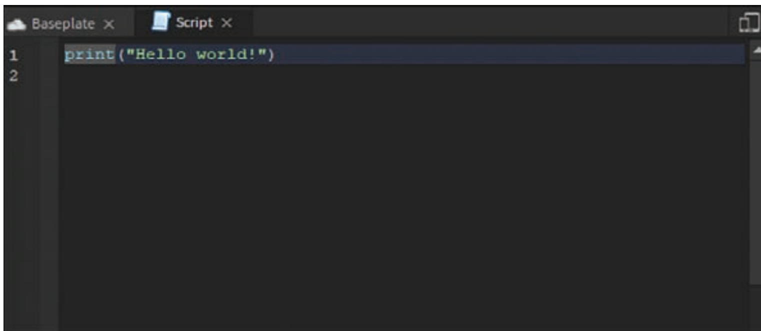


FIGURE 1.10
The window shows the default script and code.

Writing Some Code

Since the 1970s, `"Hello World!"` has been one of the first pieces of code people have learned. Here it's being used in the print function. Functions are chunks of code that serve a specific purpose. As you learn to code, you'll use prebuilt functions like `print()`, which displays messages in the Output window. You will, of course, also learn how to create functions of your own.

`print()` displays a string, which is a type of data usually used with letters and numbers that need to stay together. In this case, you're printing `"Hello world!"`:

1. Make this code your own by changing the message inside of the quotation marks to what you want for dinner tonight. Here's an example:

```
print("I want lots of pasta")
```

2. To test the code, in the Home tab, click Play (see Figure 1.11).

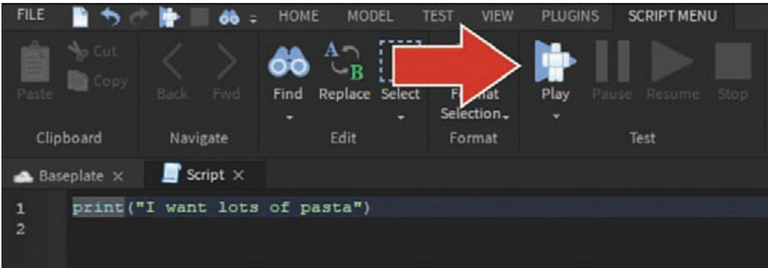


FIGURE 1.11
Click Play to test your script.

Your avatar will fall into the world, and you can see your dinner dreams displayed in the Output window, along with a note about which script that message came from (see Figure 1.12).

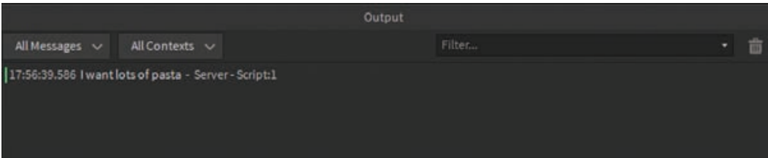


FIGURE 1.12
The string is displayed in Output.

3. To stop the playtest, click the Stop button (see Figure 1.13).

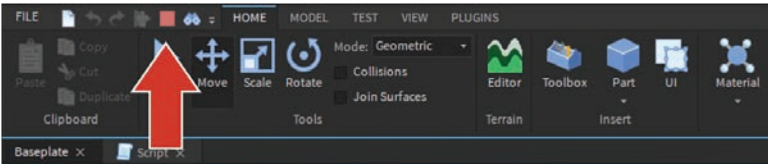


FIGURE 1.13
Click Stop to quit the playtest.

4. Return to your script by clicking on the tab above the 3D Editor, as shown in Figure 1.14.

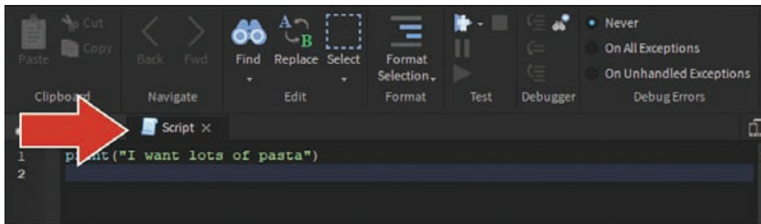


FIGURE 1.14
Click Script to return to the window where your script is visible.

Code an Explosion

Code of course can do more than just display messages to the output window. It can completely change how players interact with the world and make it come alive. Let's take a slightly longer piece of code and make the block in the Baseplate template destroy anything it touches:

1. Use the Move tool (see Figure 1.15) to move the block off the ground and away from the spawn point. The code you're going to write will destroy anything it touches, and you don't want it to go off prematurely.

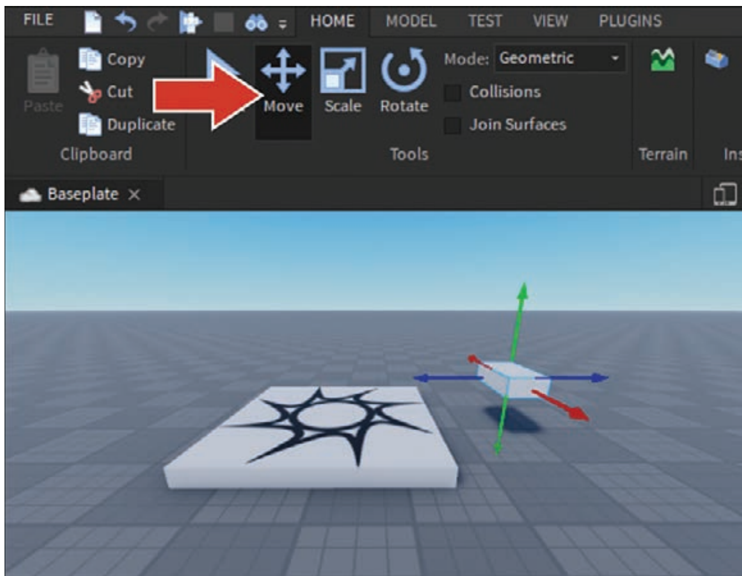


FIGURE 1.15
Move the part up and away from the spawn.

2. In the Properties window, scroll to Behavior and make sure Anchored (see Figure 1.16) is selected so the block doesn't fall when you click Play.

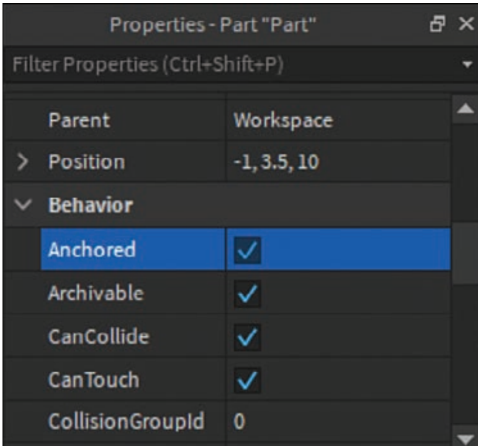


FIGURE 1.16
Check Anchored to keep the blocks from falling.

3. In the script, below the print function, add the following code:

```
print("I want lots of pasta!")

-- Destroys whatever touches the part
local trap = script.Parent
local function onTouch(partTouched)
    partTouched:Destroy()
end
trap.Touched:Connect(onTouch)
```

NOTE

Code Boxes

Code boxes for this book will be presented in light mode, unless specifically calling attention to Studio UX.

4. Click Play and run up and touch the part.

The result should be that your character breaks or parts of your avatar are destroyed. You may notice that this code only destroys what touches it directly, such as your feet. Try jumping on top

of the block or brushing against it with just a hand. You'll see only that part of your avatar is destroyed.

The reason is that code only does what you tell it, and you told the part to destroy only what it touches and nothing more. You have to tell it how to destroy the rest of the player. Throughout this book, you'll learn how to write additional instructions so that the code can handle more scenarios like this one. In Hour 4, "Parameters and Arguments," you'll learn how to make sure it destroys the entire player character.

Error Messages

What if the code didn't work? The truth is, all engineers make mistakes in their code. It's no big deal, and the editor and the output window can help you spot mistakes and fix them. Try making a couple of mistakes to learn how to better spot them later:

1. Delete the second parenthesis from the `print` function. A red line appears under `local`. (See Figure 1.17.) In the editor, red lines indicate a problem.

```
Baseplate ×   Script ×
1  print("I want lots of pasta!"
2
3  -- Destroys whatever touches the part
4  local trap = script.Parent
5  local function onTouch(partTouched)
6      partTouched:Destroy()
7  end
8  trap.Touched:Connect(onTouch)
```

FIGURE 1.17
A red line indicates Studio has spotted an error.

2. Hover over the red line, and the editor gives you a clue about what's gone wrong, as shown in Figure 1.18. But don't fix the mistake quite yet.

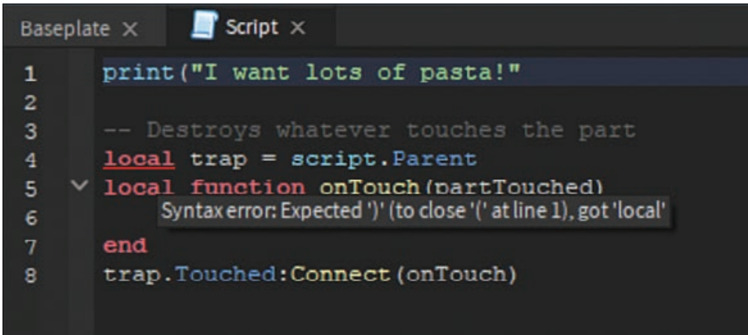


FIGURE 1.18
An error message displays when you hover over the red line.

3. Click Play, which causes an error message to display in the Output window, as shown in Figure 1.19. Click the red error, and Studio takes you to where it thinks the problem is.

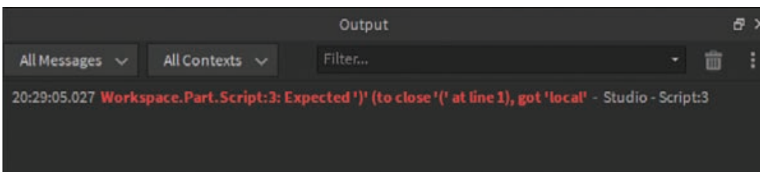


FIGURE 1.19
The error shows up as a clickable red message in the Output window.

Stop the playtest and fix the issue.

TIP

Changes Made While Playtesting Aren't Permanent

Be careful about making changes while in a playtest because the work you've done is not automatically saved. If you do make changes, be sure to click Preserve Changes when you stop the playtest.

Leaving Yourself Comments

In the previous code, you may notice the sentence `-- Destroys whatever touches the part`. This is a comment. Comments begin with two dashes. Anything on the same line as the dashes doesn't affect the script.

Coders use comments to leave notes to themselves and others about what the code does. Trust us: When you haven't looked at a piece of code in months, it's very easy to forget what it does.

The following code shows what it might look like to add a comment at the top of the script you wrote earlier in this hour:

```
-- What do I want for dinner?
print("I want lots of pasta!")
```

Summary

In just one hour, you've come a long way, particularly if this happened to be your first time coding or using Roblox Studio. This hour covered creating an account and opening Roblox for the first time. By using the + button, you were able to insert a script into a part, and then you added code that turned the part into a trap for anyone who happened to touch it.

In addition, you learned how to test code using the Play button and use the built-in error detection within the script editor and Output window to help you troubleshoot when something goes wrong.

Finally, you learned about comments, which are only readable in the script editor and can be used to leave notes about the purpose of the code.

Q&A

Q. Can you use Studio on a Chromebook?

A. To create, Studio must be run on a MacOS or Windows machine. Once a game has been published, it's available to be played on Android, Apple, Mac, PC, Chrome, and potentially even XBox Live.

Q. How do I reopen a script if I close it?

A. If you close out of the script editor, you can reopen it by double-clicking the script object in Explorer.

Q. How do I save my work?

A. Go to File, Publish to Roblox to save to the cloud, which makes your game accessible from any computer.

Q. Where do I go if I want additional information about how Roblox Studio works?

A. You can visit developer.roblox.com to find documentation on all of Studio's features and API.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. Roblox uses the _________ coding language.
2. Aspects of an object such as color, rotation, and anchored can be found in the ______ window.
3. Game objects are found in the _________ window.
4. To enable the Output window, which displays code messages and errors, enable it in the ________ tab.
5. True or false: Comments change the code to enable new functionality.
6. To force parts to stay in place, they need to be _______.

Answers

1. Lua
2. Properties
3. Explorer
4. View
5. False. Comments do not affect the code and are used to leave notes to yourself and other coders as to the purpose of the script.
5. Anchored

Exercise

Before moving on, take a moment to experiment with the creation tools by creating a mini obstacle course. It could be individual parts the player has to avoid, or it could be a lava floor like the one shown in Figure 1.20.

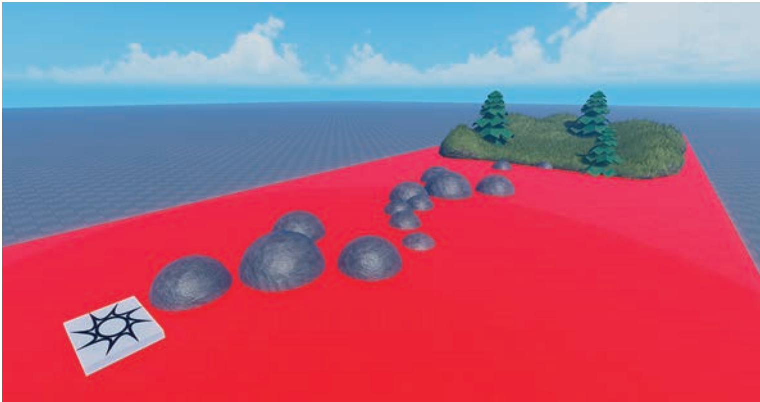


FIGURE 1.20
Use what you've learned so far to create a lava obstacle course.

Tips

- Create more parts and manipulate them with the Move, Translate, and Scale tools found on the Home tab (see Figure 1.21). You can also change the parts' appearance with Material and Color.

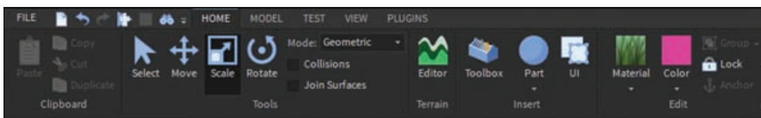


FIGURE 1.21
The Home tab has the tools you need to create and manipulate parts.

- Use a single large part and insert a script as you did earlier to turn it into lava.
- Additional models can be found in the Toolbox; just be aware that some models may already have scripts in them.
- Don't forget to anchor all parts and models.
- If you know how to use the terrain tools, you can work that into your obstacle course as well.

This page intentionally left blank

HOUR 2

Properties and Variables

What You'll Learn in This Hour:

- ▶ About the parent/child relationship of objects in the Explorer
- ▶ How to make changes to an object's properties
- ▶ How to create variables
- ▶ How to assign values to variables
- ▶ Which types of data variables can hold
- ▶ How to create instances of objects

In this hour, you learn how to find the objects you want to make changes to in the hierarchy and create an adorable NPC (Non Playable Character) guide like the one in Figure 2.1 that can warn players of upcoming danger. To create the guide, you use code to update a part's appearance and behavior.

FIGURE 2.1
An NPC warns players of upcoming danger.

Object Hierarchy

If you want to affect objects with code, you have to be able to say where those objects are within the game's hierarchy. As you look in the Explorer, you can see some game objects are nested inside of others. For example, in Figure 2.2, you can see that the Baseplate object is nested inside of Workspace. This makes Baseplate a *child* of Workspace, which makes Workspace the *parent* object. And even though you can't see it in Explorer, Workspace is a child of Game.

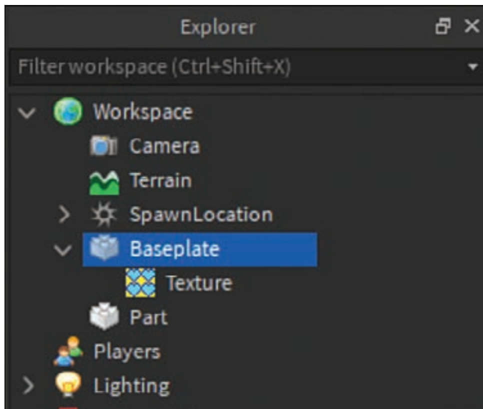


FIGURE 2.2
Baseplate is a child of Workspace.

In code, you can navigate the hierarchy of the game using the *dot operator*—for example, game.Workspace.Baseplate.

This is how you give directions in the script to tell the code what object to work with.

▼ TRY IT YOURSELF

Search and Destroy

Use the dot operator to search within Workspace for the baseplate and use `Destroy()`, which was also used in Hour 1 to get rid of the baseplate.

1. Insert a new script into Baseplate. Rename the script **`DestroyBaseplate`** by double-clicking it or pressing F2 (see Figure 2.3).

TIP

Rename Scripts and Objects

Renaming scripts and objects in your project is important for staying organized.

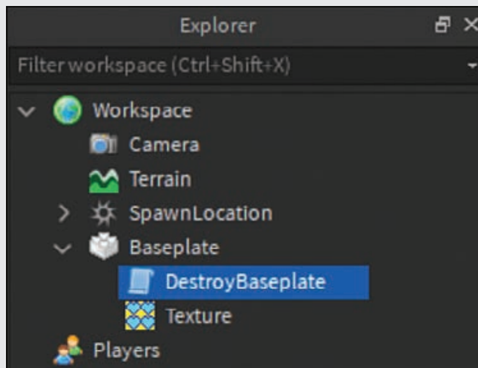


FIGURE 2.3
You can rename a script.

2. In the script, type `game.Workspace.Baseplate:Destroy()`.

3. Playtest the game, and the baseplate will be destroyed, possibly even before your character loads.

Keywords

Now let's talk about keywords. *Keywords* can be thought of as the words that make up a coding language. Each keyword serves a special purpose. Lua has fewer keywords than most coding languages, which makes it one of the easiest to learn. Some keywords are built into Lua automatically, and some have been added by Roblox to make things easier.

One keyword in Roblox Lua is `workspace`, lowercased, because `game.Workspace` was typed so much, the thoughtful Roblox engineers decided to supply a keyword to shorten it.

TRY IT YOURSELF

Use the `workspace` Keyword

Update the code you just wrote with the keyword `workspace` in place of `game.Workspace`.

1. In your prior code, replace `game.Workspace` with `workspace`.

TIP

Correct Capitalization Is Important

Keywords are case-sensitive, so make sure workspace is lowercased.

2. Playtest and verify the code still works.

Now back to hierarchy. Not only can the children of objects be accessed with the dot operator, but so can parent objects. This time, use the keyword `script`, which always represents the Script object no matter what the object is named, and use the dot operator to access the parent.

▼ TRY IT YOURSELF

Shorten the Code

You can actually shorten the code even more and get rid of the baseplate by using `Destroy()` with `script.Parent`:

1. In the same script as before, replace your code with `script.Parent:Destroy()`.

TIP

Take Advantage of Autocomplete

As you type, you may see suggested code appear. You can accept the suggestion by pressing Enter. This will save time on typing and minimize the risk of making typos.

2. Playtest and verify your code.

You can find a complete list of keywords in the appendix at the back of the book.

Properties

In addition to navigating the hierarchy, the dot operator also allows you access to the properties of an object. So what are properties? I'll explain with an example: Take a look at the flower in Figure 2.4. How would you describe it to someone?

Maybe you would start off saying that it's a plant. When pressed for more information, you might say that it's a green plant with yellow petals. An engineer might add additional details like it's a green plant with yellow petals, three units tall and two units wide. Someone else might mention it's on fire (see Figure 2.5).

FIGURE 2.4
How would you describe this flower?

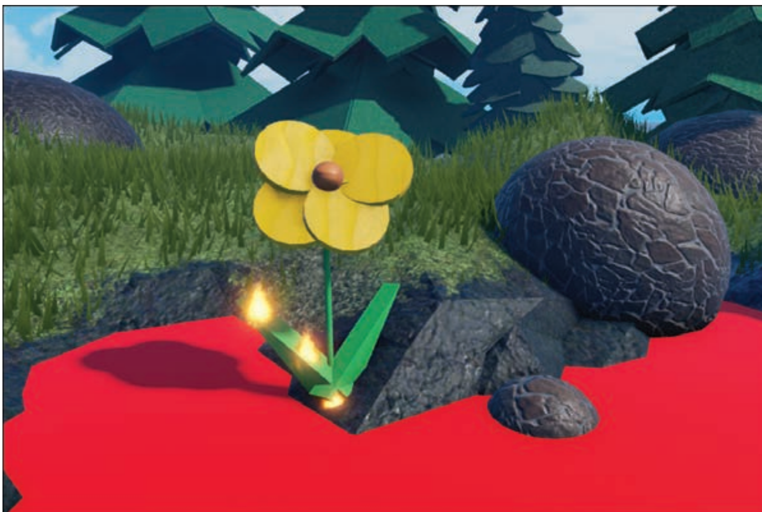


FIGURE 2.5
The flower is also flammable.

All of the ways you describe an object are its *properties*.

Finding Properties and Data Types

When you click an object in Explorer, the aptly named Properties window populates with different properties of the object that are changeable. The different formats in which properties track values are *data types*. Some important data types to start off with are the following:

- **Number:** Any real number—for example, `11.9`.
- **String:** A collection of letters and/or numbers sandwiched within quotation marks. Good for storing readable information. print() accepts string values—for example, `"99 bananas"`.
- **Boolean:** The values true and false. Properties that have states like on/off or checked/unchecked are often booleans.
- **Tables:** A set of information—for example, {`Amy, Bill, Cathleen`}.

For a larger list of data types, see the appendix.

Creating Variables

Now that you have an understanding of how to find objects in the hierarchy and how each property has its own specific value format called a data type, you can begin making variables.

Variables are placeholders for information. They can be used to keep track of objects and data types for use in your code. Once created, some variables can only be used in specific scripts or chunks of code. These are called *local variables*. Other variables are designed so that they can be used more broadly across scripts. Those are called *global variables*.

Unless you have a good reason, you almost always want to use local variables. Your code runs faster with local variables, and you're less likely to end up with clashing variable names in your code. Almost all of the variables you create in this book will be local variables.

To create a local variable, type `local` and then the desired name of the variable, for example:

```
local baseplate
```

Once the variable is created, you can assign, or set, the value of the variable using the equal sign, for example:

```
local baseplate = script.Parent
```

In your head, you can think of the equal sign as the word *is*. So, the prior variable would read *basePlate is script.Parent*. Once the variable is created, you can access the information being held as many times as you want with just the name, for example:

```
local basePlate = script.Parent
basePlate.Transparency = 0.5
```

Variables can be updated as often as you want. So if you're keeping score in a game, every time the player scores a new point, you can keep using the same variable and assign it the updated score, like so:

```
local playerScore = 10

print("playerScore is " .. playerScore)

local playerScore = playerScore + 1 -- Add one to current player score

print( "new playerScore is " .. playerScore)
```

In Output, you should see print messages similar to Figure 2.6.

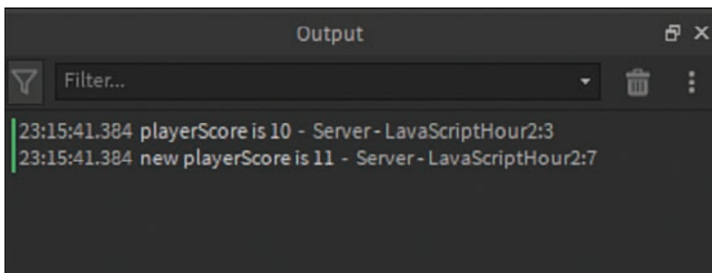


FIGURE 2.6
First, the original value of `playerScore` is printed, and then the updated `playerScore` prints.

TIP

Combining Strings and Variables

`print()` can accept both strings and variables, but they need to be combined with two dots. Combining values is called *concatenation*.

TRY IT YOURSELF ▼

Create an NPC

With just the knowledge you have so far, you can create an NPC guide that delivers a warning about the upcoming lava field to the player. This exercise will help you practice navigating hierarchy and properties using the dot operator, as well as using variables and data types.

First, you need to create the NPC:

1. Use the Part drop-down menu to create a sphere or any other type of part.
2. Rename the part to **`GuideNPC`**.
3. Insert a script into the sphere and rename it.

4. Insert a Dialog object into `GuideNPC`. Do not rename it. (See Figure 2.7.)

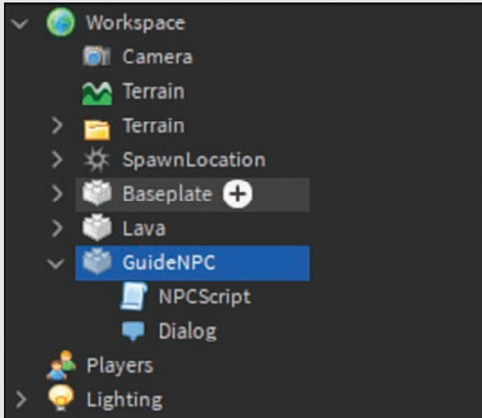


FIGURE 2.7
The NPC hierarchy includes a Script and a Dialog.

Code the Script

For this example, you create two different variables. The first variable navigates to the parent part, and the second variable holds the message that the spirit guide greets the player with when first prompted. You also add a bit of code to customize the appearance of the NPC.

1. Replace the default code in NPCScript with a new local variable named **`guideNPC`** that points at the script's parent.

```
local guide = script.Parent
```

TIP

Object Naming Conventions

For consistency, in-game objects are named using CamelCase with the first letter in uppercase, and variables named after them are pascalCased, with the first letter in lowercase.

2. Create a second variable holding the guide's message with a string value. The message can be anything you like as long as it's a string.

```
local guideNPC = script.Parent
local message  = "Danger ahead, stay on the rocks!"
```

3. Make the NPC more ghostly by accessing its properties and setting Transparency to `0.5`.

```
local guideNPC = script.Parent
local message  = "Danger ahead, stay on the rocks!"
guideNPC.Transparency = 0.5
```

4. Access the child Dialog object and its property InitialPrompt. Set InitialPrompt to `message`.

```
local guideNPC= script.Parent
local message  = "Danger ahead, stay on the rocks!"
guideNPC.Transparency = 0.5
guideNPC.Dialog.InitialPrompt  =  message
```

Playtest and click the question mark above the NPC's head to see the message.

Changing the Color Property

A property commonly changed in code is an object's color property. To change the color, you need to understand how light works. Every color on your screen is actually a product of just three types of light; red, green, and blue. The strength of each color ranges from 0 to 255. All three colors at full strength (255 , 255, 255) appear white onscreen. Each band turned all the way down (0, 0, 0) is black. Pure red is (255, 0, 0), and pure green is (0, 255, 0). So what do you think pure blue is?

Turn your NPC purple by mixing a little red with a lot of blue:

```
guideNPC.Color = Color3.fromRGB(40, 0, 160)
```

TIP

Use the Color Picker to Find the Right Values

As you type, a small color wheel will appear (see Figure 2.8). If you click it, you can select the color you want, and click OK to automatically set the correct RGB value.

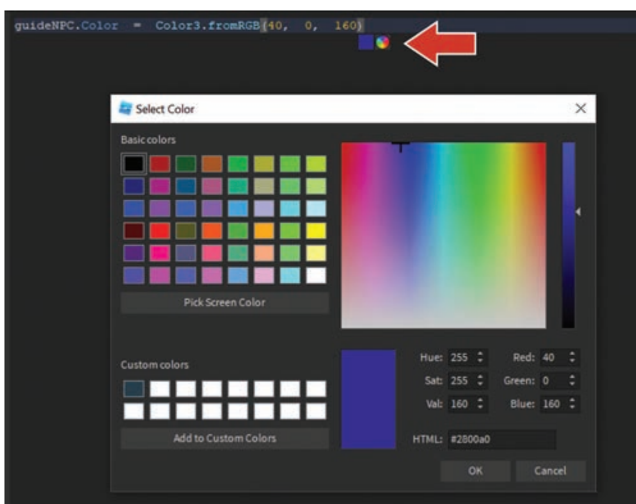


FIGURE 2.8
Click the color wheel to bring up the RGB color selector.

Instances

The last topic for this hour is instances. *Instances* are copies of game objects like parts, scripts, and sparkles.

Rather than using the + button like you have so far, instances can instead be created with the function `Instance.new()`, as shown here:

```
local part = Instance.new("Part")
```

Once you've created a part, you can access all of its properties like normal. Make any desired changes, and then parent it to the workspace.

▼ TRY IT YOURSELF

Create a New Part Instance

Instead of inserting a part directly into Explorer, use code to create the part, change the part's color, and then place it in workspace where it can be seen.

1. In ServerScriptService, add a new script.
2. Create an instance of a part; then set the color and finally the parent:

```
local part = Instance.new("Part")
part.Color = Color3.fromRGB(40, 0, 160)
part.Parent = workspace
```

You can even take this one step further by creating instances inside of instances:

```
local part = Instance.new("Part")
local particles = Instance.new("ParticleEmitter")
part.Color = Color3.fromRGB(40, 0, 160)
particle.Parent = part
part.Parent = workspace
```

TIP

New Part Instance Appears at the World's Center

When new parts are created via code, they appear at the very center of the world, where the default spawn point is. If you can't see your part when testing, try moving the spawn point and then testing again.

Summary

Every object in a game has properties like Color, Scale, and Transparency that determine how the object looks and behaves in game. Each property uses values formatted in a specific way called a data type. A few common data types are strings, booleans, and numbers.

Within code, dot notation is used to access properties of an object, as well as to find the object in the Explorer hierarchy.

Once you understand an object's properties and how to access them in the game's hierarchy, you can begin making changes using code.

Variables can be used as placeholders for information that you want the script to work with. There are two main types of variables, global and local. Of the two, local variables should always be used unless there is a specific reason not to.

Game objects such as Parts, Scripts, Dialogs, and ParticleEmitters can be created in a running script by using the function `Instance.new()`, which accepts the name of the object type as a string.

Q&A

Q. How do you know what data type a property accepts?

A. You can look up a game object, its properties, and their corresponding types on developer.Roblox.com. For example, in a search engine, type Roblox Dialog Properties and look for API results on the Roblox domain.

In Figure 2.9, you can see a portion of the Dialog API page. It has a short description and a list of properties with their matching data types. You can click each property and data type to learn more about how to use it.

APIREFERENCE > DIALOG

Dialog

Show deprecated

The Dialog object allows users to create non-player characters (NPCs) that players can talk to using a list of choices. The Dialog object can be inserted into a part such as a Humanoid's head, and then a player will see a speech bubble above the part that they can click on to start a conversation. The creator of a place can choose what choices the player can say by inserting DialogChoice objects into the dialog.

See Also:

- How to use Dialogs

Properties

DialogBehaviorType BehaviorType
Sets whether the Dialog can be used by multiple players at once.

float ConversationDistance
The furthest distance that I player can be from the Dialog's parent to start a conversation.

bool GoodbyeChoiceActive
Toggles whether the goodbye option will be displayed.

FIGURE 2.9
This snippet of the API page for Dialog shows a short description of properties and matching data types.

Q. Why should you not set a variable to include the property you want to change? Like *`local partColor = workspace.Part.Color`*?

A. The hierarchy information and the property information are two different types of data and can't be mixed.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. What type of data type only accepts the values true or false?
2. Variables are _____ for information.
3. If I were storing a player's name, which would be a good data type: a string, a boolean, an enum, or a float?
4. To access the script's parent, use _______.
5. A Dialog object inserted into a part is a _____ of the part?
6. The process of combining values for use by `print()` is called _____.

Answers

1. Booleans
2. Placeholders
3. String
4. `script.Parent`
5. Child
6. Concatenation

Exercises

A face would make your NPC much more personable. One can easily be added on by inserting a decal. For the texture, you can use the one in the link provided or upload your own.

Tips

- Give the spirit a face (see Figure 2.10) by inserting a decal instance and updating the texture property with the following string: `"http://www.roblox.com/asset?id=494290547"`

- You may have to rotate the NPC to get it to face the right way. Or you can try updating the decal's face property to change the placement of the decal.
- Find the code solution in the appendix.

FIGURE 2.10
The NPC now has a face, making it feel more alive to players.

- For the second exercise, see if you can create the spirit guide from start to finish only using code.

Tips

- Insert a new Script object into `ServerScriptService` to write your code in.
- Use `Instance.new()` to create the part that will act as the guide's body, the Dialog object.
- Don't forget to anchor the part. Anchored accepts boolean data types.
- Make all changes to the part, including adding the children objects before finally parenting the guide to the workspace.
- The NPC appears at the dead center of the world as a cube. In Hour 14, "Coding in 3D World Space," you learn how to work with the coordinate system to move objects to exactly where you would like them.
- See the appendix for an example code snippet.

This page intentionally left blank

HOUR 3
Creating and Using Functions

What You'll Learn in This Hour:

- How to create functions in Lua
- How to call functions to make them run
- How to use events to call functions
- How scope works

In Hours 1 and 2, you used the prebuilt functions `print()`, `destroy()`, and `new()`. This hour talks more about what functions actually are, how to create your own functions, and how to get your functions to run using events that happen in the world.

The second half of the hour talks a little bit about how code is organized so you can better understand how placement in a script matters when making sure that code will run.

Creating and Calling Functions

Functions are packaged bits of code designed for specific purposes that can be used when needed, as often as needed.

The code that you used in the last hour to create your NPC ran as soon as the playtest session started. But what if you don't want the code to run right away? Say if you only want the NPC to appear after the player clicks a button or completes a quest. Or what if you want to create multiple NPCs, but don't want to write all the same code again. These types of scenarios are great for functions. You can write the code the same as you did, package it in a function, and have it run whenever you want.

1. To create a function, type **`local function nameofFunction`**.
2. Press Enter to automatically close the function with `end`. Your code will look like this:

```
local function nameOfFunction()

end
```

3. Inside the function, add your code on an indented line. In this case, we used `print()` just for testing purposes. All of the code for the function must be typed before `end`:

```
local function nameOfFunction()
    print("Function Test")
end
```

TIP

Indent Your Code

The code will work if not properly indented. Even so, proper indentation makes code a lot easier for you and other people to read, so we highly recommend that you use indentation in your code.

4. Once the function is created, all that's left is to tell it to run. To do this, you have to call the function by typing the function name followed by a parenthesis:

```
local function nameOfFunction()
    print("Function Test")
end

nameOfFunction()
```

   If you don't call the function, it won't run.

5. The function will run as many times as you call it. Try calling it a few more times:

```
local function nameOfFunction()
    print("Function Test")
end

nameOfFunction()
nameOfFunction()
nameOfFunction()
```

The name of the function can be whatever you like, as long as it's followed by `()`, but let's take a second to think about properly naming functions. Here are some guidelines to follow:

- Names should tell you what the function does. For example, `destroy()` clearly destroys things.
- Function names in Lua are typically pascalCased. They begin lowercase, and each new word is capitalized.
- Do not include spaces or special characters in function names. This will cause errors.

TIP

Method: Another Name for *Function*

Functions that are prebuilt or belonging to existing world objects such as `print()`, `wait()`, and `destroy()` are often referred to as methods in other coding languages. Lua users tend to just say function regardless of whether it could be referred to as a method.

Understanding Scope

Just mentioned was that any code *not* between the first and last lines of a function won't run when the function is called. Code that is outside a function is *out of scope*. Scope is the information that a particular chunk of code, such as a function, can see and access.

If you run the following code, the `print` function inside of the function will run three times, and the `print` function on the outside will run only once:

```
local function scopeTest()
    print("This is in scope")
end

print("This is out of scope")

scopeTest()
scopeTest()
scopeTest()
```

Using Events to Call Functions

Just typing the name of a function is one way to call it, and that works well when you want it to be called at a particular point in the script. Sometimes, however, you don't know in advance when the function should be called. You want the function to run when something in particular happens in the experience. Here are some examples:

- Giving a user a sword when they click on a loot chest
- Assigning a player to a team when they've joined a game
- Destroying a piece of a bridge when a player has touched it

For these sorts of scenarios, you don't know in advance when they'll happen, but you do know what code you want to run when they do. What you're waiting for is a specific *event*. When the event happens, a signal is fired that can be used to tell code to run. To call a function whenever an event has been fired, use `Connect()` and pass in the name of the function to run, but leave off the `()`.

Here's an example:

```
partName.Touched:Connect(functionName)
```

The `Touched` event is built into parts, so it can be accessed using the dot operator like other children. The colon is then used to access the function named `Connect()`.

TRY IT YOURSELF

Create a Vanishing Bridge

Part objects have several built-in events, one of the most useful being `Touched`. The `Touched` event fires whenever its parent part has been collided with. Let's use the `Touched` event to create a bridge where the pieces become transparent half a second after being touched by an explorer:

1. Create a bridge piece (see Figure 3.1) using parts or models. Make sure to anchor the part in place.

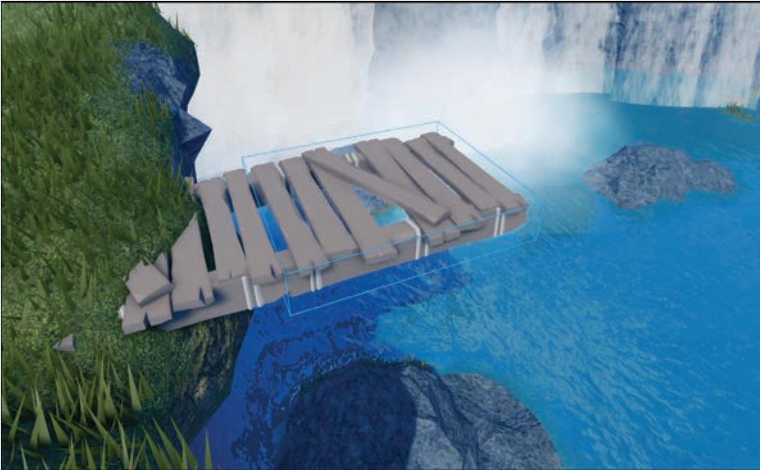


FIGURE 3.1
Use a model or a part to act as part of a bridge.

2. Insert a script into the part, and rename the script **`BridgeScript`** (see Figure 3.2).

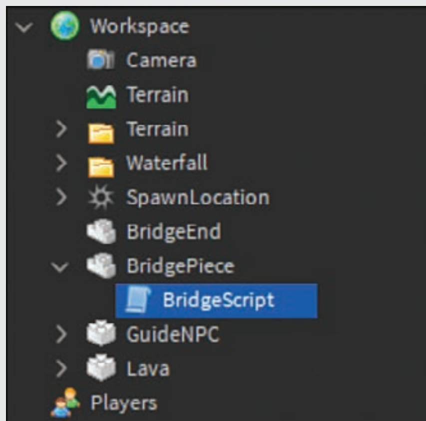


FIGURE 3.2
Insert a script into BridgePiece.

3. Assign the parent part to a local variable: `local bridgePart = script.Parent`.

4. Create a new local function named `onTouch`:

```
local bridgePart = script.Parent
local function onTouch()

end
```

TIP

Naming Functions Used With Events

A common naming pattern for functions that are called with events is `onBlank`, where `Blank` is the name of the event. Just another way of making your own code easy to read when you have to update it after a year.

5. Connect the function to the part's `Touched` event. Once that's done, you can use `print()` to test your code so far:

```
local bridgePart = script.Parent
local function onTouch()
    print("Touch event fired!")
end

bridgePart.Touched:Connect(onTouch)
```

6. Inside of the function, add the code that should run when the event fires. Here, the part will turn transparent, and in 0.5 seconds, anyone standing on the bridge will be dropped. If you added a `print` statement in the last step, go ahead and delete it:

```
local bridgePart = script.Parent

local function onTouch()
    bridgePart.Transparency = 0.5
    wait(0.5)
    bridgePart.CanCollide = false
end

bridgePart.Touched:Connect(onTouch)
```

TIP

Using Booleans

CanCollide is a boolean. When true, that object can interact with things in the world. When false, it can't. So in this case, when false, the bridge no longer supports the user.

Understanding Order and Placement

When creating variables and functions, it's important to remember that where they are located in the script matters. Scripts run code line by line, starting from the top and working to the bottom.

So if you try to use a variable or a function before it's been created in the script, you'll run into problems (see Figure 3.3 and Figure 3.4). Take a look at the BridgeScript you just completed:

```
local bridgePart = script.Parent

local function onTouch()
    bridgePart.Transparency = 0.5
    wait(0.5)
    bridgePart.CanCollide = false
end

bridgePart.Touched:Connect(onTouch)
```

If you move the first line to the bottom, then the previous mentions of the variable now show errors.

```
1
2
3  local function onTouch()
4      bridgePart.Transparency = 0.5
5      wait(0.5)
6      bridgePart.CanCollide = false
7  end
8
9  bridgePart.Touched:Connect(onTouch)
10
11 local bridgePart = script.Parent  -- Being at the bottom causes errors
```

FIGURE 3.3
Moving the creation of bridgePart to the bottom causes errors due to the unknown variable.

```
1  local bridgePart = script.Parent
2
3  onTouch() -- Function hasn't been assigned yet
4
5  local function onTouch()
6      bridgePart.Transparency = 0.5
7      wait(0.5)
8      bridgePart.CanCollide = false
9  end
10
11 bridgePart.Touched:Connect(onTouch)
12
13
```

FIGURE 3.4
Calling a function before it's assigned also causes issues.

In both of these examples, errors are caused because the script is trying to call something that isn't there yet.

Now that you know that order matters, let's talk about variables that are created inside of a function. Start with this basic set of three variables: one before the function, one inside, and one below:

```
local above = "above"

local function scopePractice()
    local inside = "inside"
end

local below = "below"
```

At the bottom of the script, try to print all three variables. What happens? `inside` will error (see Figure 3.5) despite having been assigned previously. That's because local variables inside of a function can't be accessed from the outside.

```
1  local above = "above"
2
3  local function scopePractice()
4      local inside = "inside"
5  end
6
7  local below = "below"
8
9  print(above)
10 print(inside)  -- Can't see inside of scopePractice
11 print(below)
```

FIGURE 3.5
Local variables inside of a function can't be accessed outside of the function.

To understand why this doesn't work, you need to understand that a script is a series of nested blocks of code. Each time you create a new function, you're creating a new block. Figure 3.6 illustrates how those blocks can overlap. The first block, block A, is the script itself. Inside is a function, shown as block B.

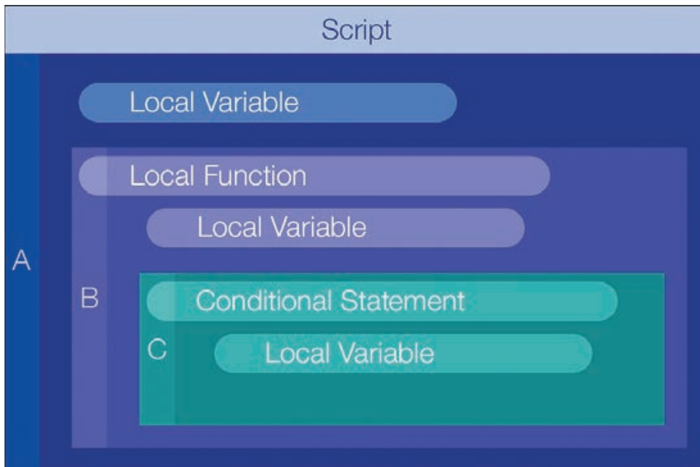


FIGURE 3.6
Functions create a new block of code within the script.

Within the function can be more blocks, created by conditional statements and other things that you'll learn about in a couple of hours. Each block can access local variables/functions in its parent block, but not those in any child blocks:

- Block B can access the local variable in block A.
- Block C can access the local function/variables in blocks A and B.
- Block A cannot access the local function/variables in blocks B or C.
- Block B cannot access the local variable in block C.

▼ TRY IT YOURSELF

Reactivate the Bridge

One thing to remember about Roblox experiences is that they are on live servers and are inherently multiuser, which means that lots of people can be in the same server at the same time. For that reason, you don't want a broken bridge like the one in Figure 3.7 permanently in your experience once a player has gone across it. Keep in mind what you've learned about scope and create a second function that reactivates the bridge.

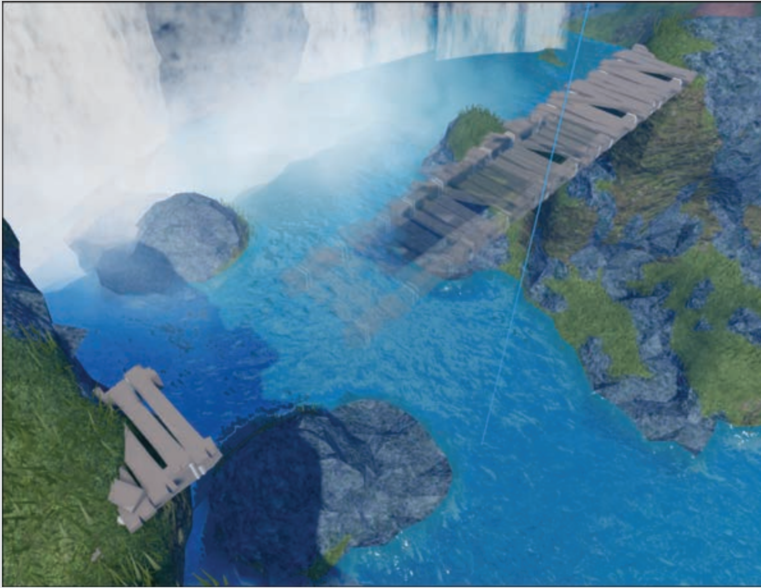


FIGURE 3.7
The bridge needs to be reset so users can cross it again.

1. In the same script you used in the earlier Try It Yourself, create a new function called `activateBridge()` above the `onTouch` function:

```
local bridgePart = script.Parent

local function activateBridge()

end

local function onTouch()
    bridgePart.Transparency = 0.5
    wait(0.5)
    bridgePart.CanCollide = false
end

bridgePart.Touched:Connect(onTouch)
```

2. In `activateBridge()`, reverse the changes to `CanCollide` and `Transparency`:

```
local function activateBridge()
    bridgePart.Transparency = 0
           bridgePart.CanCollide = true
end
```

3. Inside of `onTouch()`, call `activateBridge()` after a short amount of time:

```
local bridgePart = script.Parent

local function activateBridge()
    bridgePart.Transparency = 0
    bridgePart.CanCollide = true
end

local function onTouch()
    bridgePart.Transparency = 0.5
    wait(0.5)
    bridgePart.CanCollide = false
    wait(3.0)
    activateBridge()
end

bridgePart.Touched:Connect(onTouch)
```

TIP

Pay Attention to the Order of Your Functions

`activateBridge()` needed to come before `onTouch()` in order to keep it in scope.

Summary

Functions are reusable chunks of code that you can use again and again and again. Once defined, they can be called by simply typing `functionName()`. Or alternatively, if you don't know exactly when the function will be needed, you can connect them to an event. That way, the function will be called each time the event fires.

When creating a script, it's important to keep in mind what information a chunk of code has access to. Variables and functions need to be within the scope of a code chunk to be used. Code chunks can access information within their own chunk and within their parent chunk. Trying to access information that's out of scope results in errors in the script.

A good way to practice scope is take a piece of code that you have working, like the bridge script, and move around functions and variables to see when things break.

Q&A

Q. Can you create a variable without assigning it a value if you don't know it yet?

A. Yes, the variable can be created earlier and then later assigned a value.

Q. Can you have more than one function with a script?

A. Yes, you'll quite often have multiple functions created within a script.

Q. Why not make everything global and not worry about scope?

A. In addition to global variables running slower than local variables, there's a lot of time you'll have to create multiple functions within the same script. These functions will all need their own variables. If you don't make your variables local, it's very easy to accidentally overwrite a variable when you meant to make a new one.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. What's another word for telling a function to run?
2. What function is used to run a function when an event is fired?
3. An object's events are accessed using ___________.
4. What are two reasons for using local variables instead of global variables?
5. True or false: If a local variable is inside of a function, it can be accessed by all functions further down the script.
6. What symbol is used to run a function associated with an object? Hint, think of how `Connect()` and `Destroy()` were used.

Answers

1. Call
2. `Connect()`
3. Dot notation
4. Local variables run faster and prevent accidentally overwriting values in the case of duplicate names.
5. False. Variables are only accessible within their own code block and child code blocks.
6. A colon is used to call a function associated with an object—for example, `part:Destroy()`.

Exercise

Instead of having the bridge collapse after being touched, try creating a bridge or track piece that solidifies after a player touches a button and then resets. Create one function that activates the bridge when touched and a second function that deactivates the bridge (see Figure 3.8).

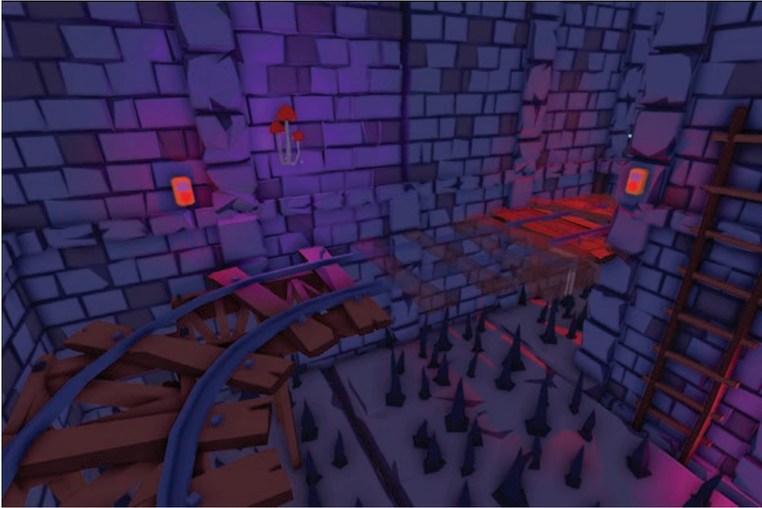


FIGURE 3.8
Users need to touch the buttons to activate the missing bridge piece.

Tips

- It'll be easier to use a single bridge part rather than multiple parts.
- Place the script in the button to take advantage of the Touched event.
- Use `wait()` to control how long the bridge is active.
- Turn the button green while the bridge is active.
- Make sure the bridge piece starts off disabled so that it can be enabled by the script.

The code solution is in the appendix.

HOUR 4
Working with Parameters and Arguments

What You'll Learn in This Hour:

- How to use parameters
- How to use multiple parameters and arguments
- How to return values from functions
- How to use anonymous functions

Functions can not only perform tasks; they can perform like little factory machines that take things in, transform them, and then gives back the results. This hour covers the information that goes inside of the parentheses—parameters and arguments—and what a function can do with that info.

Giving Functions Information to Use

Functions don't have to be stand-alone chunks of code; they can actually take in information from the outside to use. You've seen this done with `print("Hello")`, which takes in messages to display, and `wait(3)` which takes in a number of seconds to pause a script.

The values that get passed into a function through the parentheses are called *arguments*. When creating your own functions where information will get passed in, you need to create placeholders for the arguments. Those placeholders are called *parameters*.

To create your own parameters, add a variable name within the parentheses when you define the function, like so:

```
local function functionName(parameterName)

end
```

The parameter can then be used within the function just like any other variable.

▼ TRY IT YOURSELF

Create a Painting Function

The side of the building in Figure 4.1 will be changed by creating a new function named `paint()`, which will have a parameter for taking in what color the wall should be repainted. Instead of using a building like this, you can practice with a regular old part or an untextured model.

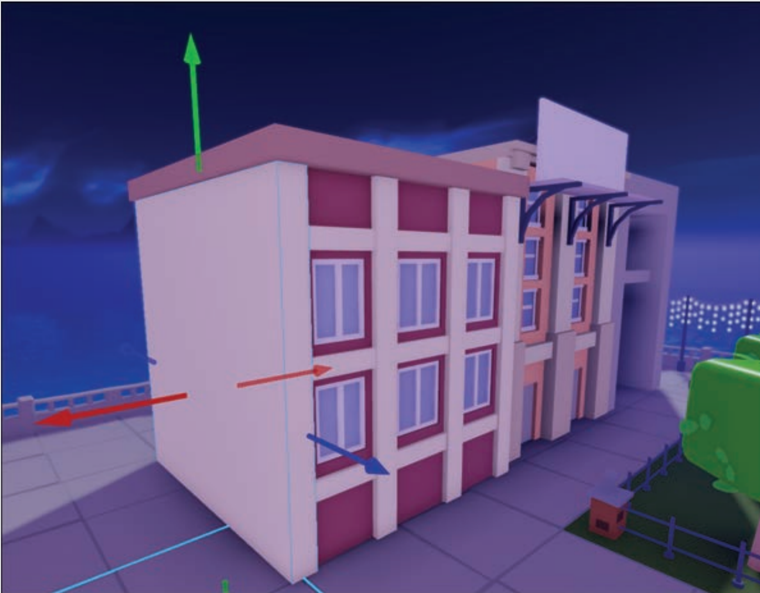


FIGURE 4.1
Use a function to repaint part of the building.

1. Inside of a part or model, add a new script named Paint.
2. Create a new local variable assigned to the parent part, and then a new local function named `paint()`:

```
local wall = script.Parent
local function paint()

end
```

3. Create a parameter named `paintColor` that will act as a placeholder for what color to paint the wall with:

```
local wall = script.Parent
local function paint(paintColor)

end
```

4. Inside the function, set the color of the wall to the `paintColor` placeholder:

```
local wall = script.Parent
local function paint(paintColor)
    wall.Color = paintColor
end
```

5. Add a variable or two with different RGB colors you might want to paint the wall:

```
local wall = script.Parent

local blue = Color3.fromRGB(29, 121, 160)
local yellow = Color3.fromRGB(219, 223, 128)

local function paint(paintColor)
    wall.Color = paintColor
end
```

TIP

Variable Placement

You may have already noticed, but variables usually go at the top of the script or the code chunk they belong to.

6. Call the paint function and pass in one of the color variables:

```
local wall = script.Parent

local blue = Color3.fromRGB(29, 121, 160)
local yellow = Color3.fromRGB(219, 223, 128)

local function paint(paintColor)
    wall.Color = paintColor
end

paint(blue)
```

Test it out, and whatever part you painted will be the color you picked!

Working with Multiple Parameters and Arguments

The previous Try It Yourself let you pass in the colors you wanted to paint with, but the object to be painted was hard-coded in. In other words, the code would only work with that particular object.

Hard-coding really limits the use of the function unless you plan on changing the color of that one wall a lot. Luckily, you can pass more than one argument into a function. All you have to do is create more than one parameter.

Multiple parameter names can be separated with a comma when defining the function, as shown here:

```
local function functionName(firstParameter,secondParameter)
    print(firstParameter .." and ".. secondParameter)
end
```

TIP

How Many Is Too Many?

There's no technical limit to how many parameters you can have, but most people agree that no more than three is a good rule of thumb.

The arguments that get passed in always fill up the parameters in order. The first argument always goes through the first parameter, and the second argument always goes through the second parameter:

```
local first = "first"
local second = "second"

local function practice(firstParameter,secondParameter)
    print(firstParameter .. " and " .. secondParameter)
end

practice(first, second) -- Prints "first and second"
practice(second, first) -- Prints "second and first"
```

▼ TRY IT YOURSELF

Pass in What Color and What Object

Make the paint function more useful by creating a variable that takes in both an object to paint and a color to paint with. Figure 4.2 shows a car and a building currently painted white, which is incredibly boring and doesn't match the scene. Take the same painter code as before but make it so you can pass in the object you want to paint. That way, the code can be used on both the building and the car:

FIGURE 4.2
A function with two parameters can be used to designate both what object to paint and what color to use.

1. In ServerScriptService, create a new script.
2. Assign variables for two different colors and two different objects:

```
-- Available colors
local red = Color3.fromRGB(170, 0, 0)
local olive = Color3.fromRGB(151, 15, 156)

-- Objects to paint
local car = workspace.Car
local restaurant = workspace.Buildings.Restaurant
```

TIP

Finding Embedded Objects

Note that for the second object in the example, the restaurant was in a folder, so dot notation was used to navigate one more step down the hierarchy.

3. Create a function that has a parameter to take in an object to paint and the color to paint it:

```
-- Paints objects
local function painter(objectToPaint, paintColor)
    objectToPaint.Color = paintColor
end
```

4. Call the function and pass in which object should be painted and what color:

```
-- Available colors
local red = Color3.fromRGB(170, 0, 0)
local olive = Color3.fromRGB(151, 15, 156)

-- Objects to paint
local car = workspace.Car
local restaurant = workspace.Buildings.Restaurant

-- Paints objects
local function painter(objectToPaint, paintColor)
   objectToPaint.Color = paintColor
end

painter(restaurant, olive)
painter(car, red)
```

5. Test your code. Instead of Play, in the drop-down menu, select Run (see Figure 4.3) if you want to see the changes to the world without playing the experience.

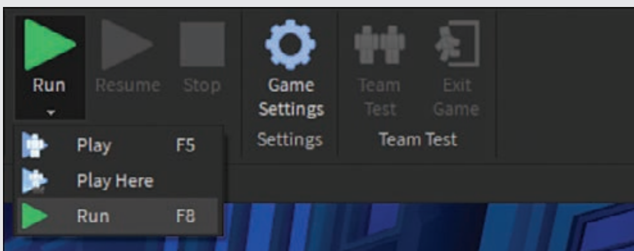


FIGURE 4.3
Use Run to test code without loading a character avatar.

Figure 4.4 shows the finished car and building, which are no longer startlingly white.

FIGURE 4.4
Once the script runs, both the building and the car have had their color changed.

Returning Values from Functions

Not only can values be passed into a function, but they can also be passed back. A classic example is a calculator, just like the one on your phone. Values go in, and the result is **returned**. In the following code example, a function is assigned to a variable. When the variable is used, the function runs, and the result is sent back using the keyword `return`:

```
-- Adds any two numbers together
local function add(firstNumber, secondNumber)
    local sum = firstNumber + secondNumber
    return sum -- Sends sum back to where the function was called
end

-- Some numbers to use
local rent = 3500
local electricity = 128

-- Use add() to add rent and electricity and return the result
local costOfLiving = add(rent, electricity)
print("Rent in New York is " .. costOfLiving)
```

Returning Multiple Values

Sometimes you may want multiple values returned from a function. An example may be returning how many wins, losses, and ties a user has. To return multiple values, use `return` as normal and separate the values with a comma.

TRY IT YOURSELF

Return Player's Wins, Losses, and Ties

Follow the steps to create a custom function which, when called, returns a player's wins, losses, and ties. Assign the returned values to a variable:

1. Create a custom function with variables for wins, losses, and ties.
2. Type `return` followed by the desired variable. Use a comma to separate them:

```
local function getWinRate()
    local wins = 4
    local losses = 0
    local ties = 1
    return wins, losses, ties
end
```

3. Rather than creating variables for each received value on separate lines, create them on the same line as in the following example. They'll be filled in order with the returned values:

```
local function getWinRate()
    local wins = 4
    local losses = 0
    local ties = 1
    return wins, losses, ties
end
local userWins, userLosses, userTies = getWinRate()
```

4. Print the variables to see the results.

```
local function getWinRate()
    local wins = 4
    local losses = 0
    local ties = 1
    return wins, losses, ties
end

local userWins, userLosses, userTies = getWinRate()
print("Your wins, losses, and ties are: " .. userWins .. " , " .. userLosses
.. " , " .. userTies)
```

Returning Nil

Nil means something can't be found or doesn't exist. If you see `nil` printed instead of the expected output, do the following:

- Check that the number of values received is the same as those returned.
- Verify that values returned and received are separated by commas.
- Confirm nothing else is wrong with the function.

TRY IT YOURSELF ▼

Returning Something That Doesn't Exist

If you try to use a variable or function that doesn't exist, you can see the keyword nil displayed in Output, as well as where the error occurred.

1. In any script, pass a fake variable name like `doesntExist` into `print()`.
2. Run the code and check Output. You should see the nil alongside the name of the script and the line number of where the variable couldn't be found, such as the error shown in Figure 4.5.

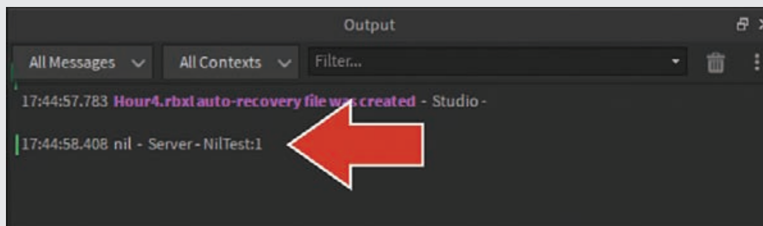


FIGURE 4.5
The error message shows that a value couldn't be found at line 1 of the script named NilTest.

Dealing with Mismatched Arguments and Parameters

It's important to be aware of what will happen if the wrong number of values are passed into a function or are returned. Having the wrong number can cause your code to error and freeze.

If insufficient arguments are passed into a function, an error occurs when the function reaches the nil value:

```
local function whoWon(first, second)
    print("First place is " .. first .. "Second place is ")
end


whoWon("AngelicaIsTheBest") -- Will error because there is not a second value
```

If more values are passed back than there are available variables, the values fill up in order, and any variables left over will be dropped and lost. In this example, three values are passed back, but there are only spaces for two:

```
local function giveBack()
    local a = "Apple"
    local b = "Banana"
    local c = "Carrot"
    return a, b, c
end

local a, b = giveBack() -- c is lost


print(a, b, c) -- Will print Apple, Banana, nil
```

`"Carrot"` only exists within the scope of the function and is never returned, so nil is printed for the third value.

Working with Anonymous Functions

Anonymous functions are, as the name implies, functions. What makes them special is that when they are first defined, they go unnamed. This means they can be defined in the same place as they are called. Compare the following two code samples for our familiar simple trap connected to the `Touched` event. The `Touched` event returns the name of the triggering part, which is then destroyed.

First, here is the script where a named function is created and then called whenever the `Touched` event is fired:

Named Function Example

```
local trap = script.Parent

local function onTouch(otherPart)
    otherPart:Destroy()
end

trap.Touched:Connect(onTouch)
```

Now here is code that does the same thing, but the function is created in the same place where it's being called:

Anonymous Function Example

```
local part = script.Parent

part.Touched:Connect(function(otherPart)otherPart:Destroy() end)
```

If you were to run both pieces of code, they do the exact same thing: destroy anything that touches the script's parent. So why wouldn't you use an anonymous function? The table shows some pros and cons for unnamed functions.

Pros	Cons
Faster to type.	More difficult to read.
Can be used with functions that don't return values otherwise.	Trickier to update and reuse.
	Can't be called from elsewhere, because they don't have a name to call them by.

TIP

Named Functions Makes Collaborating Easier

The *Roblox Lua Style Guide* discourages the use of anonymous functions when not necessary because most projects will have multiple coders, and anonymous functions make code much more difficult to read and update.

Summary

Functions can be used and reused in a lot of different ways. They can be used to create something, like the NPC in Hour 2. They can be used to make changes to an object by updating properties or even destroying them altogether, as with a trap part. To do so, they can take in values from outside of the function by passing those values through parameters. The actual chunks of information that get passed through parameters are called arguments.

When the function completes its work, information can be passed back and used by the script. A classic example of information being returned is the calculator on your phone. If you use the calculator to add two numbers, it'll pass back the answer. Another example we've seen is whenever the `Touched` event is fired, `Touched` passes back the name of the object that caused it to fire.

Sometimes, if there's nothing to return, you might use an anonymous function and create the function in the same place as it's called. This can be convenient, but it also makes your code a lot harder to read, which possibly means slowing down teammates who are working with the script. It can even make it more challenging for yourself if you want to make updates to the code later.

Q&A

Q. Is there a maximum amount of parameters a function can have?

A. There's not a strict maximum, but most of the time, you'll want to limit it to just three. The more parameters you have, the harder it becomes to remember what each one is for, and easier to mess up the order.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. Handing information from outside of a function to the inside is called ______.
2. What keyword allows values to be handed back when a function is done?
3. The placeholder for values that will later be used by a function are called _______.
4. The actual values used by functions are called ______.
5. The keyword that is used when values can't be found or don't exist is ____.

Answers

1. Passing
2. `return`
3. Parameters
4. Arguments
5. `0bnil`

Exercise

All coders will at times research how other people have made something. However, the code you find online might not do exactly what you want or be formatted in a way that makes it easy for your team members to read. It's important that you take the time to examine borrowed code and make improvements where you can. In this exercise, practice by taking an anonymous function and trying to reformat it as a named function:

```
script.Parent.Touched:Connect(function(otherPart) local fire = Instance.new"Fire"
fire.Parent = otherPart end)
```

See the appendix for the solution.

This page intentionally left blank

HOUR 5
Conditional Structures

What You'll Learn in This Hour:

- How to use `if/then` statements
- How to work with operators
- How to use multiple conditions with `elseif` and `else`
- How to find the Humanoid

Have you ever told someone you would do something—on one condition? For example, you'll help them move to a new place, but they have to help you study for finals. That's a conditional structure. You'll do something *if* something else happens.

The same thing can happen in scripts. You can set up code so that it'll run only *if* something else is true. Figure 5.1 shows a flowchart of how a conditional structure works.

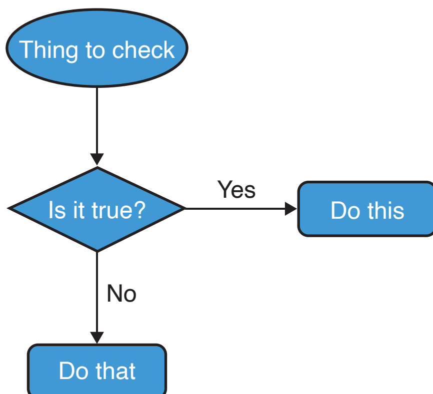


FIGURE 5.1
The code decides which option to take based on whether something is true or false.

This hour explores the world of conditional structures in which code runs only if certain conditions are met.

if/then Statements

The most common conditional structure is probably an `if/then` statement. If something is true, then the code will do something.

Here are a few examples:

- *If* a key is found, *then* a new area can be explored.
- *If* a quest is completed, *then* the user will receive a free pet.
- *If* someone says happy birthday in chat, *then* make a burst of balloons on screen.

In code, it looks like this:

```
if somethingIsTrue then
    -- Do something
    print("It's true!")
end
```

If the first line is true, then the `print` command in the indented code will run.

Conditional statements can use operators to evaluate if something is true. *Operators* are symbols that give directions about how to evaluate something. Table 5.1 shows some of the most common operators. For a complete list, refer to the appendix.

TABLE 5.1 Common Operators

Operator	Description	Examples of Being True
==	Is equal	`If 3 == 3 then`
+	Addition	`If 3 + 3 == 6 then`
–	Subtraction	`If 3 - 3 == 0 then`
*	Multiplication	`If 3 * 3 == 9 then`

Pay close attention to the double equal sign used as an operator as compared to a single equal sign used to assign a value to a variable. A double equal sign == is used to check if something is equal.

The following evaluates as true, and the code will run:

```
local health = 10

if health == 10 then
    print("You're at full health")
end
```

The following evaluates as false, and the code won't run:

```
local health = 5

if health == 10 then
    print("You're at full health")
end
```

What if the player has temporary bonus health? You can check for that with the operator for greater than or equal to (>=):

```
local health = 12

if health >= 10 then
    print("You're at full health")
end
```

You can also check for the presence of a value if no operator is used. The following code snippet checks to see if a roof is on fire:

```
local roof = script.Parent

local fire = roof:FindFirstChildWhichIsA("Fire")

if fire then -- Checks if fire is not nil
    print("The roof is on fire!")
    fire:Destroy()
end
```

It uses `FindFirstChildWhichIsA()` to see if any of the roof's children objects are Fire objects. `FindFirstChildWhichIsA()` only gets the first object it finds that matches its search.

TRY IT YOURSELF ▼

Introducing Humanoids

The lava in Hour 1 had a major flaw: It only destroyed whatever touched it directly, which means that users could possibly be running around without feet or hands if they managed only to brush against the trap, as in Figure 5.2.

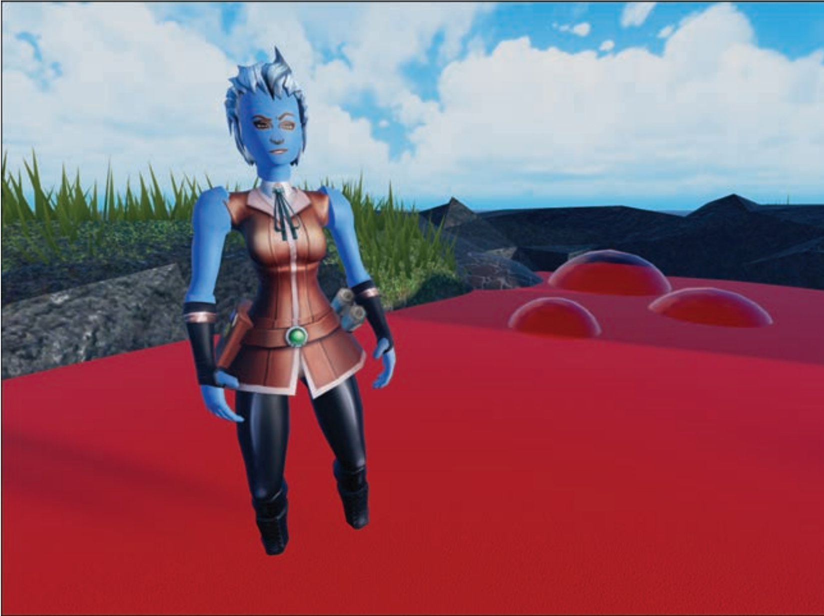


FIGURE 5.2
This person has lost their feet.

To reset the user completely, you need to find the object that controls the user's health. In Roblox, by default that's the Humanoid object. If you use the Humanoid to set the user's health to 0, they'll be forced to respawn—legs, feet, hands, and all.

1. Create a part and insert a new script. You can use your lava from Hour 1 as long as you delete the old script.

2. Create a variable assigned to the trap part itself.

3. Start a function named `onTouch` with a parameter for `otherPart`.

4. Inside of the function, create a variable named **`character`** to find `otherPart`'s parent, if it has one:

```
local trap = script.Parent

local function onTouch(otherPart)
    local character = otherPart.Parent
    local humanoid = character:FindFirstChildWhichIsA("Humanoid")

end
```

5. The next step is to check whether the character has a Humanoid. If it does, it's most likely a user or an NPC:

```
local trap = script.Parent

local function onTouch(otherPart)
    local character = otherPart.Parent
    local humanoid = character:FindFirstChildWhichIsA("Humanoid")

    if humanoid then

    end
end
```

6. Set the user's health to `0`:

```
local trap = script.Parent

local function onTouch(otherPart)
    local character = otherPart.Parent
    local humanoid = character:FindFirstChildWhichIsA("Humanoid")

    if humanoid then
        humanoid.Health = 0
    end
end
```

7. Connect `onTouch` to the trap's `Touched` event:

```
local trap = script.Parent

local function onTouch(otherPart)
    local character = otherPart.Parent
    local humanoid = character:FindFirstChildWhichIsA("Humanoid")

    if humanoid then
        humanoid.Health = 0
    end
end


trap.Touched:Connect(onTouch)
```

You can find the API page for Humanoids at https://developer.roblox.com/api-reference/class/Humanoid.

elseif

OK, but what if you want the code to check for more than one scenario? For example, you want the code to do one thing if the user's health is full, but a different thing if the user's health is not full. In these scenarios, add a second conditional called with the keyword `elseif`:

```
local health = 5

if health >= 10 then
    print ("You're at full health")

elseif health < 10 then -- Check if health is less than 10
    print("Find something to eat to regain health!")
end
```

The `elseif` is still part of the same code block as `if/then`. It does not have its own `end`.

Logical Operators

A few special operators aren't symbols; instead, *logical operators* are the words `and`, `or`, and `not`. `and` and `or` allow you to check against multiple conditions at the same time. `not` lets you make sure something isn't something else. Table 5.2 explains how these operators are evaluated.

TABLE 5.2 Logical Operators

Operator	Description
`and`	Evaluates as true only if both conditions are true.
`or`	Evaluates as true if either condition is true.
`not`	Evaluates as the opposite of the condition.

These operators consider both false and nil as "false" and anything else as "true."

In the following snippet, `and` is used to check for a range rather than a single value. In this scenario, we imagine the user has a maximum of 10 health, and at 0 health respawns. So the user needs to eat only if they are less than full health:

```
local health = 1
ssssssssss
if health >= 10 then
    print ("You're at full health")

elseif health >= 1 and health  < 10 then -- Check if health is in a specific range
    print("Find something to eat to regain health!")
end
```

You can keep creating code for specific scenarios as needed with additional `elseif` statements:

```
local health = 1

if health >= 10 then
    print ("You're at full health") -- Runs if health is 10 or higher

elseif health >= 5 and health < 10 then -- Runs if health is 5 - 9
    print("Find something to eat to regain health!")

elseif health >= 1 and health <= 4 then -- Runs if health is 1 - 4
    print("You are very hungry, better eat soon!")
end
```

else

Finally, it's always wise to tell the script what to do if none of the other contingencies are met. Use the keyword `else` to mark what should be done if no other conditions are met:

```
local health = 0

if health >= 10 then
    print ("You're at full health") -- Runs if health is 10 or higher

elseif health >= 5 and health < 10 then -- Runs if health is 5 - 9
    print("Find something to eat to regain health!")

elseif health >= 1 and health <= 4 then -- Runs if health is 1 - 4
    print("You are very hungry, better eat soon!")

else -- Runs if none of the conditions have been true.
    print("You ran out of food, you'll need to restart")
end
```

Once again, `else` is not its own code block; the entire conditional should only use the keyword `end` once.

TRY IT YOURSELF ▼

Make a Portal with Attributes and Services

Practice using `if/then` and `elseif` statements by creating a portal that only allows players to pass into the tunnels on the other side *if* they've activated a special keystone nearby. (See Figure 5.3.) To create the portal, you get to learn about the ProximityPromptService and custom attributes.

FIGURE 5.3
A keystone (left) that needs to be activated before people can use the portal (right).

First, you need to set up the portal and keystone using either parts or models. In Figure 5.3, a model was used for the portal arch, but the portal itself is just a black part acting as a barrier. The arch is just for show.

Both the portal and keystone will need a new attribute to make the script work. *Attributes* are custom properties that you can name and set a value type for. An attribute named `Activated` will be created for each part to track whether the key has been found and if the portal can be used:

1. Set up parts or meshes named **`Portal`** and **`KeyStone`**.
2. Select Portal, and insert a ProximityPrompt (see Figure 5.4). ProximityPrompts enable users to click and interact with parts instead of only being able to run up and touch them.

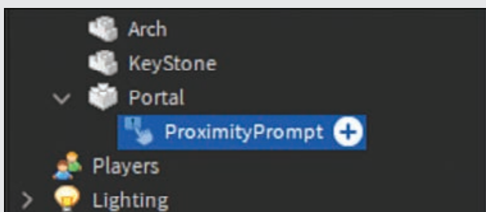


FIGURE 5.4
Insert a ProximityPrompt object.

3. With Portal still selected, in Properties, scroll all the way down and click Add Attribute (see Figure 5.5).

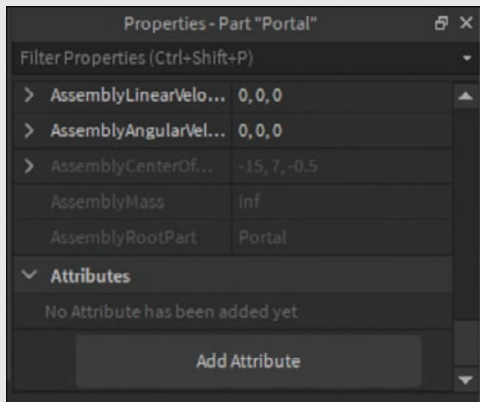


FIGURE 5.5
Click Add Attribute.

4. Name the attribute **`Activated`**, set the type to boolean, and then click Save (see Figure 5.6).

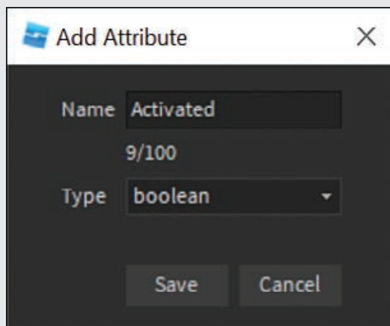


FIGURE 5.6
Name the attribute `Activated` and set the type to boolean.

5. Select KeyStone, create another new attribute named **`Activated`**, and set the type to boolean.

WARNING

Leave the New Attributes Unchecked

Make sure to leave both new attributes disabled, as in not checked. Enabled/checked attributes and properties are `true`, whereas disabled/unchecked are `false`.

Next, you create two scripts: one for the key and one for the portal.

Getting Attributes in the KeyStone Script

The KeyStone's attribute, `Activated`, should currently be `false`. As long as the key isn't activated, the portal doesn't allow people to pass through. The KeyStone's script is used to turn on the key when touched, and set `Activated` to `true`:

1. Select KeyStone and insert a new script.
2. Create a variable to reference the script's parent and a function named `onTouch` connected to the KeyStone's `Touched` Event. Include a parameter for the touching part.
3. Check to see if a person has touched the part by looking for a Humanoid. You don't want the code to be triggered by a touching baseplate or something similar. Refer to the code earlier in the hour if you can't remember how.
4. Inside the function, use `SetAttribute()` to pass in `Activated` and change the value to `true` as shown:

```
local keyStone = script.Parent

local function onTouch(otherPart)
    local character = otherPart.Parent
    local humanoid = character:FindFirstChildWhichIsA("Humanoid")

    if humanoid then
        keyStone:SetAttribute("Activated", true)
    end
end

keyStone.Touched:Connect(onTouch)
```

5. Change the KeyStone's material to Neon to show the user that the KeyStone has been activated:

```
local keyStone = script.Parent

local function onTouch(otherPart)
    local character = otherPart.Parent
    local humanoid = character:FindFirstChildWhichIsA("Humanoid")
```

```
        if humanoid then
            keyStone:SetAttribute("Activated", true)
            keyStone.Material = Enum.Material.Neon
        end
    end

    keyStone.Touched:Connect(onTouch)
```

TIP

Alternatives for Textured Parts

If your part has a texture, changes to materials and color won't show up. You can delete the texture so these properties show or enable a particle when activated. The important thing here is to always show users when they have an interaction with a part.

6. Test and make sure the part changes to neon, as shown in Figure 5.7.

FIGURE 5.7
The KeyStone (left) glows neon blue (right) after it's activated so users know it's working.

Portal Script

Back to the portal itself: When the user walks up to the portal, the ProximityPrompt displays a message showing the barrier can be interacted with, as shown in Figure 5.8.

FIGURE 5.8
A ProximityPrompt's default message is shown to users when they get close enough.

ProximityPrompts have a number of associated functions that can be used, but they aren't automatically included with the functions normally available in a script. We can make these functions available to us by adding the ProximityPromptService to the script. Services are optional sets of code that provide additional functions for your script to use. These code sets can be made available for use by assigning them to a variable with `GetService()`. Here's an example:

```
local ProximityPromptService= game:GetService("ProximityPromptService")
```

TIP

Using Colons with Methods

As a reminder, when accessing methods—that is, functions associated with an object—you use colons. Here, `GetService()` is associated with the top-level object, `game`:

1. Select Portal and insert a new script.
2. Create a variable to get `ProximityPromptService`.
3. Create variables to reference Portal, KeyStone, and ProximityPrompt.
4. Create a new function named **`onPromptTriggered`**:

```
local ProximityPromptService = game:GetService("ProximityPromptService")

local portal = script.Parent
local keyStone = workspace.KeyStone
```

```
local proximityPrompt = portal.ProximityPrompt

local function onPromptTriggered()

end
```

TIP

Naming Functions

You may have noticed that the formula of `on` and the name of the event is a common way to name functions.

5. Connect the function to the `PromptTriggered` event that comes with `ProximityPromptService`:

```
local function onPromptTriggered()

end

ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)
```

6. Inside of the function, get the current value of the KeyStone's attribute, `Activated`:

```
local function onPromptTriggered()
    local KeyActivated = keyStone:GetAttribute("Activated")
end
```

7. If KeyStone is Activated, make the part transparent and disable CanCollide:

```
local function onPromptTriggered()
    local KeyActivated = keyStone:GetAttribute("Activated")

    if KeyActivated == true then
        portal.Transparency = 0.8
        portal.CanCollide = false
        print("Come on through")
    end
end
```

8. Otherwise, make the door blink red:

```
local ProximityPromptService = game:GetService("ProximityPromptService")

local portal = script.Parent
local keyStone = workspace.KeyStone
local proximityPrompt = portal.ProximityPrompt
local originalColor = portal.Color

local function onPromptTriggered()
     local KeyActivated = keyStone:GetAttribute("Activated")
```

```
        if KeyActivated == true then
            portal.Transparency = 0.8
            portal.CanCollide = false
            print("Come on through")

        else
            portal.Color = Color3.fromRGB(255, 0, 0)
            wait(1)
            portal.Color = originalColor
            print("Activate the key stone to pass through the portal")
        end
    end

    ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)
```

TIP

Taking into Account Multiple Player Interactions

A variable is used to get the portal's original color at the top of the script instead of in the function. Otherwise, if you spam the interaction, the function might start while the portal is still red. The variable would then be assigned red instead of the original color.

The script is finished! Test it out and make sure people can use the portal. In the Property window, you can even customize the text of the ProximityPrompt (see Figure 5.9) and how far away players have to be.

FIGURE 5.9
You can customize the ProximityPrompt text.

Summary

Using conditional statements can really make your world start to come to life, allowing you to set up cause-and-effect reactions in the world. If people touch something dangerous, then they lose health. If they touch something else, magical powers can be given or new doors can open.

The keywords `if/then`, `elseif`, and `else` are what allow you to create the flowchart for what code should run under what circumstances. The script checks each condition starting from the top, and if the condition is true, the code for that section is run. The rest of the code in the `if/then` statement is skipped. If nothing is true, an `else` can be used to say what the code should do.

While setting up these interactions, always think of the people who will be experiencing the world you're creating. Include visual clues such as color changes or special effects to make sure the user understands an object is working as intended.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. A double equal sign (==) means ___ _____.
2. What's wrong with the following code snippet?

```
local health = 5
if health >= 10 then
    print ("You're at full health")

                elseif health < 10 then
                   print("Find something to eat to regain health!")
                end
end
```

3. Make additional sets of code available in your script with ______.
4. What operator means less than or equal to?
5. Operators that are not symbols are called ______.
6. What does the operator `or` do?

Answers

1. Is equal
2. `elseif` shouldn't be its own code chunk. It should be on the same indent level as `if`, and there should only be one `end`.

3. GetService()
4. <=
5. Logical operators
6. Evaluates as true if either condition is true.

Exercise

Give a player super powers by making them go fast when they touch a speed booster! A Humanoid's default WalkSpeed property is set to 16. Not so bad, but it'd be a lot cooler to go a lot faster. Make a part that temporarily allows a user to go way faster and then returns them to their original speed after a few seconds.

To do so, you can use the onTouch pattern you've been working with and some if/then statements. As an added twist, use a ParticleEmitter object to stream sparkles behind them while they're powered up (see Figure 5.10).

Tips

- ParticleEmitters can be stored in ServerStorage.
- Get ServerStorage using GetService().
- Use if/then to check for a Humanoid.
- Change the Humanoid's WalkSpeed property from the default value of 16 to 50.
- Use the method Clone() to make a copy of the particle and then parent it to the runner.
- After a few seconds, reset WalkSpeed to 16, and destroy the ParticleEmitter.

FIGURE 5.10
Stars stream behind this running ninja while they're powered up.

You can find the code solution in the appendix.

HOUR 6

Debouncing and Debugging

What You'll Learn in This Hour:

- How to create debounce systems
- What string debugging is
- How to pull out values for easier testing
- How to create image and text labels

Now that you know what a humanoid is, and how to check for one using `if`, you can start creating code that doesn't just destroy things or set a user's health all the way to 0. Instead, you can start making things happen *incrementally*—that is, only by a certain amount at a time. Instead of taking a user's health all the way to 0, you can make it go down only part way, or a user's wealth can go up by one gold every time they mine a piece of ore.

The second half of this hour shows methods for checking and improving existing code. You'll use string statements to check where your code might have gone wrong, see how to set up systems that prevent individual users from spamming interactions, and find out how to start bringing more of the design process into your coding.

Don't Destroy, Debounce

Let's explore setting up a trap that removes 10 health points from a user at a time. A humanoid's default max health is 100. Using what you know, the easiest way to set up a trap that takes the player's current health and subtracts 10 might be as follows:

```
local trap = script.Parent
local function damageUser(otherPart)
    local partParent = otherPart.Parent
    local humanoid = partParent:FindFirstChildWhichIsA("Humanoid")
    if humanoid then
        humanoid.Health = humanoid.Health - 10
        print("Ouch! Current health is " ..  humanoid.Health)
    end
end
trap.Touched:Connect(damageUser)
```

The problem is that—because of how the physics engine handles collisions—the code will trigger multiple near-simultaneous events and cause more damage than intended. In Figure 6.1, you can see from the time stamp on the left that the person's current health went down very quickly.

```
13:17:28.043 Ouch! It's lava! Curent health is 90 - Server - SubtractHealth:9
13:17:28.360 Ouch! It's lava! Curent health is 80 - Server - SubtractHealth:9
13:17:28.577 Ouch! It's lava! Curent health is 70 - Server - SubtractHealth:9
13:17:28.676 Ouch! It's lava! Curent health is 60 - Server - SubtractHealth:9
13:17:28.992 Ouch! It's lava! Curent health is 50 - Server - SubtractHealth:9
13:17:29.108 Ouch! It's lava! Curent health is 41.016143798828 - Server - SubtractHealth:9
13:17:29.209 Ouch! It's lava! Curent health is 31.016143798828 - Server - SubtractHealth:9
13:17:39.209 Ouch! It's lava! Curent health is 31.115489959717 - Server - SubtractHealth:9
13:17:39.726 Ouch! It's lava! Curent health is 21.115489959717 - Server - SubtractHealth:9
13:17:39.960 Ouch! It's lava! Curent health is 11.115489959717 - Server - SubtractHealth:9
13:17:40.308 Ouch! It's lava! Curent health is 2.132438659668 - Server - SubtractHealth:9
13:17:40.509 ▸ Ouch! It's lava! Curent health is 0 (x69) - Server - SubtractHealth:9
```

FIGURE 6.1
The output message shows the trap was activated more often than anticipated.

We don't want the code to run so many times so fast. We want to make sure it runs only once and doesn't run again until we say it can. Ensuring that an action is triggered only once when it would otherwise be triggered multiple times is known as *debouncing*.

Here's the previous code snippet but with a debounce system that deactivates the trap for a set amount of time:

```lua
local trap = script.Parent
local RESET_SECONDS = 1 -- How long the trap will be disabled
local enabled = true -- Needs to be true to damage user
local function damageUser(otherPart)
    local partParent = otherPart.Parent
    local humanoid = partParent:FindFirstChildWhichIsA("Humanoid")
    if humanoid then
        if enabled == true then  -- Check that trap is currently enabled
            enabled = false -- Set variable to false to disable the trap
            humanoid.Health = humanoid.Health - 10
            print("OUCH!")
            wait(RESET_SECONDS)  -- Wait for reset time duration
            enabled = true -- Re-arms trap
        end
    end
end
trap.Touched:Connect(damageUser)
```

With this system, the player will only be harmed if `enabled` is equal to `true`.

TRY IT YOURSELF ▼

Mining Simulator

An excellent example of when you don’t want code to run more often than intended is when giving users gold or points. Here you create a mining simulator where users get gold every time they mine a pile of ore like the one in Figure 6.2. This Try It Yourself uses ProximityPrompts for the mining mechanic and a leaderboard where people can see how much gold they’ve collected so far.

FIGURE 6.2
Sparkling gold ore, just waiting to be mined.

Set Up the Scoreboard

You’re going to use the leaderboard that’s built into Roblox. This is the leaderboard you see in the top right of many Roblox games (see Figure 6.3). It can be used to keep track of more than just scores. It can also be used to track what level a player is on, how many resources they have, or what team they’re on.

FIGURE 6.3
The leaderboard in the top-right corner shows people's names and how much gold they've collected so far.

Whenever a player enters the game, they should be added to the leaderboard. This can be done as follows:

1. In ServerScriptService, add a new script. (See Figure 6.4.)

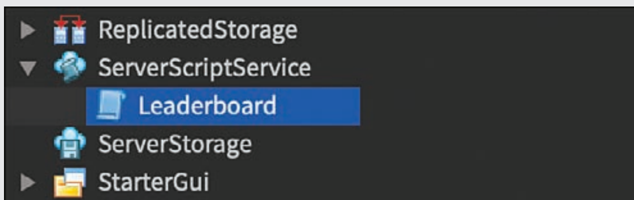


FIGURE 6.4
The Leaderboard script should be placed in ServerScriptService.

2. Get the `Players` service and connect a function to the `PlayerAdded` event:

```
local Players = game:GetService("Players")
local function leaderboardSetup(player)

end
-- Connect the "leaderboardSetup()" function to the "PlayerAdded" event
Players.PlayerAdded:Connect(leaderboardSetup)
```

3. Inside the connected function, create a new Folder instance, name it **leaderstats**, and parent it to the player:

```
local function leaderboardSetup(player)
    local leaderstats = Instance.new("Folder")
    leaderstats.Name = "leaderstats"
    leaderstats.Parent = player
end
```

TIP

Make Sure the Name Is `leaderstats`

It's really important that the folder is named `leaderstats` (all lowercase). Roblox won't add the player to the leaderboard if any other name variation is used.

4. Set up the actual stat that you see in the corner of the screen. Do your best to follow the steps without looking at the code box:

   a. Use a local variable named **gold** to create a new IntValue instance.

   b. Name the IntValue **Gold**. What you type here is exactly what will be shown to users.

   c. Set the IntValue's `Value` property to `0`.

   d. Parent the IntValue to leaderstats.

```
local Players = game:GetService("Players")

local function leaderboardSetup(player)
    local leaderstats = Instance.new("Folder")
    leaderstats.Name = "leaderstats"
    leaderstats.Parent = player

    local gold = Instance.new("IntValue")
    gold.Name = "Gold"
    gold.Value = 0
    gold.Parent = leaderstats
end
Players.PlayerAdded:Connect(leaderboardSetup)
```

TIP

IntValue Objects Can Help Track Values

IntValues are special objects that only accept integers—that is, whole numbers. That way, you don't accidentally end up with something like 6.7 points.

Set Up the Gold Ore Object

For the gold ore, you can use a part or mesh. Remember, you can always copy meshes that you see in any of Roblox Studio's templates and paste them into your file. As with the portal from the last hour, you use a ProximityPrompt to allow people to interact with the ore, and a new attribute is created. One of the cool things about ProximityPrompts is that you can modify them to include their own debounce by increasing the `HoldDuration` property:

1. Pick a part or a mesh to act as the gold ore deposit.
2. Insert a ProximityPrompt.
3. Name the ProximityPrompt **`GoldOre`**. This is important because we're going to use the name to check whether we have the right proximity prompt.
4. In ProximityPrompt's properties, change the following, as shown in Figure 6.5:
   - **ActionText:** Mine
   - **HoldDuration:** 1 (This is the amount of time users need to hold the interaction to mine the ore.)
   - **ObjectText:** Gold Ore

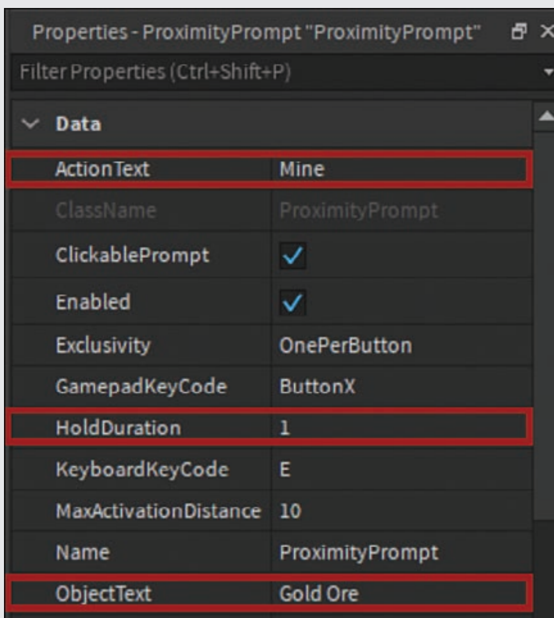


FIGURE 6.5
Modify the ActionText, HoldDuration, and ObjectText properties.

5. Select the gold ore part, and add a new attribute named ResourceType, set to string.
6. Set ResourceType to Gold, as shown in Figure 6.6.

TIP

Attributes Can Make Your Code Reusable

Using an attribute to tag the ResourceType means you can use this same script for other collectible objects.

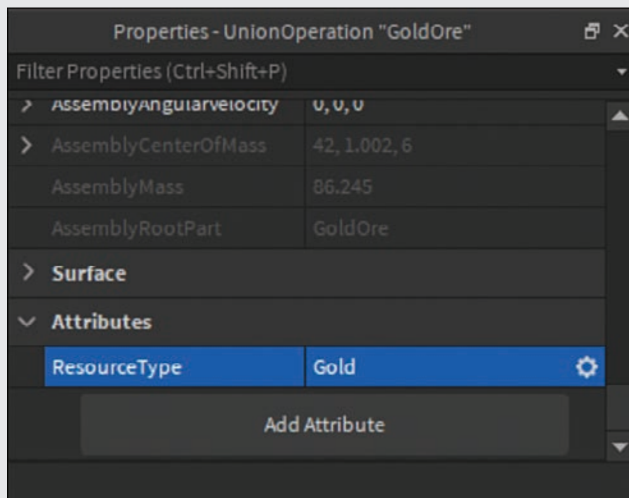


FIGURE 6.6
Add a new attribute named Resource with its value set to Gold.

Set Up the Gold Ore Script

The next step is to set up the interactivity for the ProximityPrompt, but this time you're going to put the script into ServerScriptService. This will let you have lots of gold mines that all use the same script.

With the PromptTriggered event, you can tell if the player has held the button the required amount of time:

1. Insert a new script into ServerScriptService.
2. At the top of the script, get `ProximityPromptService`. Then, create a variable for how long the prompt will be disabled once it is used. See if you can remember how to do it without referring to the code example in step 3.

3. Create a new function connected to the PromptTriggered event with parameters for `prompt` and for `player`, in that order. This way, you know when the user is done holding the button:

```
local Players = game:GetService("Players")
local ProximityPromptService = game:GetService("ProximityPromptService")

local isEnabled = true -- Debounce variable
local DISABLED_DURATION = 4
local function onPromptTriggered(prompt, player)

end
ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)
```

TIP

You Need to Account for Both Arguments

When the prompt is triggered, both the specific prompt that triggered it and the player who triggers it is returned. You only need to know the player, but remember that returned values are always returned in order. So if you want the second returned value, you need two placeholders.

4. There can be lots of proximity prompts in your game, so find the prompt's parent and see if it has an attribute named `ResourceType`:

```
local ProximityPromptService = game:GetService("ProximityPromptService")

local DISABLED_DURATION = 4

local function onPromptTriggered(prompt, player)
    local node = prompt.Parent
    local resourceType = node:GetAttribute("ResourceType")
end

ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)
```

5. If there is a resourceType, and `prompt.Enabled` is equal to true, disable the prompt:

```
local function onPromptTriggered(prompt, player)
    local node = prompt.Parent
    local resourceType = node:GetAttribute("ResourceType")
    if resourceType and prompt.Enabled then
        prompt.Enabled = false
    end
end
```

6. Find the player's leaderstats and then use the `resourceType` to update the leaderboard stats as shown:

```
local function onPromptTriggered(prompt, player)
    local node = prompt.Parent
    local resourceType = node:GetAttribute("ResourceType")
    if resourceType and prompt.Enabled then
        prompt.Enabled = false

        local leaderstats = player.leaderstats
        local resourceStat = leaderstats:FindFirstChild(resourceType)
        resourceStat.Value += 1

    end
end
```

7. After a certain amount of time has passed, re-enable the prompt so that it can be used again:

```
local ProximityPromptService = game:GetService("ProximityPromptService")
local DISABLED_DURATION = 4

local function onPromptTriggered(prompt, player)
    local node = prompt.Parent
    local resourceType = node:GetAttribute("ResourceType")
    if resourceType and prompt.Enabled then
        prompt.Enabled = false

        local leaderstats = player.leaderstats
        local resourceStat = leaderstats:FindFirstChild(resourceType)
        resourceStat.Value += 1

        wait(DISABLED_DURATION)

        prompt.Enabled = true

    end
end

ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)
```

You've finished! Add some visual indicators when the ore is disabled, like changing the transparency or color of the ore. (See Figure 6.7.) When it's working as you like, go ahead and duplicate the ore as many times as you want.

The cool thing is that because you just have the single script in ServerScriptService, if you need to make changes to the script later, it's really easy no matter how many copies of the ore you add to your game.

FIGURE 6.7
Now you have a dark disabled gold ore in the foreground and an enabled bright gold ore in the background.

TIP

Saving Player Data

With the code you have so far, the user has to start over every time they join the game. Hour 17 explores how to save users' data between sessions.

Figuring Out Where Things Go Wrong

We all make mistakes. Even pro Roblox developers who have been coding for years make mistakes every day. The key is to develop a detectivelike attitude toward both what went wrong with your code and how your code might encounter unexpected situations when users interact with it in the experience you build. The second half of this hour covers some techniques you can use to test your code and iterate on it to make a better experience for the people who visit your Roblox worlds.

Using String Debugging

As your coding skill grows and you continue to challenge yourself, you'll quite often not be sure why your code didn't run on the first try. The most obvious thing to check for first is underlined errors in the editor and Output window. Sometimes, though, that's not enough.

The next step in figuring out where things went wrong is trying to find where the code didn't run as expected. Maybe a function wasn't actually called or the given values weren't what you'd expect. One way to narrow things down is combining print statements with your knowledge of scope. Use print statements to verify variables are what you expect and code is running when you would expect.

For example, if you want to make sure a function was called, put a `print` statement right at the beginning of the function:

```
local speedBoost = script.Parent

local function onTouch(otherPart)
    print("onTouch was called!")
    -- Code Body
end

speedBoost.Touched:Connect(onTouch)
```

If for some reason you don't see `"onTouch was called!"` in the output window, you know the function was never called. Maybe the event didn't fire, or it's not connected to the function. If you do see the message, you need to check whether the problem was with the next code chunk and verify if code chunks are running when expected. The following code snippet is for creating a speed boost. The code can be placed within a script inserted into a part.

A `print` statement is used to verify the user's walk speed before the conditional, when the walk speed is supposed to have been changed, and after it's set back to normal.

This way, you can verify the code is running, and `WalkSpeed` is changing as you would expect:

```
local speedBoost = script.Parent

local function onTouch(otherPart)
    print("onTouch was called!")
    local character = otherPart.Parent
    local humanoid = character:FindFirstChildWhichIsA ("Humanoid")
    if humanoid and humanoid.WalkSpeed <= 16 then
    -- Checks for Humanoid without speed boost
    print("Original walk speed is " .. humanoid.WalkSpeed)
        humanoid.WalkSpeed = 30
        print("New walk speed is " .. humanoid.WalkSpeed)
        wait(1) -- Duration of boost
        humanoid.WalkSpeed = 16
        print("Walk speed is returned to " .. humanoid.WalkSpeed)
    end
end

speedBoost.Touched:Connect(onTouch)
```

Once you're done testing your code, always go back and delete all unnecessary `print` statements. Every line of code that runs makes the script just a little bit slower because there is more for the script to do. Deleting unnecessary code keeps things running as quickly as possible.

Moving Values for Easier Testing

Even if your code runs perfectly, you may still need to tweak things. Maybe in the last code snippet, you're not exactly sure how fast you want the player to run or how long the buff should last. It's a great practice to move important variables that affect user experience to the top of the script, making it easier for you and team members to tweak as needed.

This code snippet does the same thing as the previous snippet, but variables have been created at the top for how fast players will go, and for how long the boost will last:

```
local speedBoost = script.Parent

local BOOSTED_SPEED = 20
local BOOST_DURATION = 1

local function onTouch(otherPart)
    local character = otherPart.Parent
    local humanoid = character:FindFirstChildWhichIsA ("Humanoid")
    if humanoid and humanoid.WalkSpeed <= 16 then
    print("Original walk speed is " .. humanoid.WalkSpeed)
        humanoid.WalkSpeed = BOOSTED_SPEED
        print("New walk speed is " .. humanoid.WalkSpeed)
        wait(BOOST_DURATION) -- Duration of boost
        humanoid.WalkSpeed = 16
        print("Walk speed is returned to " .. humanoid.WalkSpeed)
    end
end

speedBoost.Touched:Connect(onTouch)
```

In a very long script, this saves you a lot of time with updating while experimenting to find just the right value to use—particularly if the same value is used in multiple places.

Variables like the ones created for `NEW_SPEED` and `BOOST_DURATION`, which never change value throughout the entirety of a script, are called *constants*. Unlike normal variables, they are typed in ALL_CAPS with words separated by an underscore (_).

TRY IT YOURSELF ▼

Tweak the SpeedBoost

Take the previous code snippet and adjust the values until you feel like the speed and duration are just right. One technique you can use while experimenting with values is *doubling and halving*. It's particularly good if you're not sure what a great number to use is.

1. Add a new part or mesh and insert a script into it, as shown in Figure 6.8.

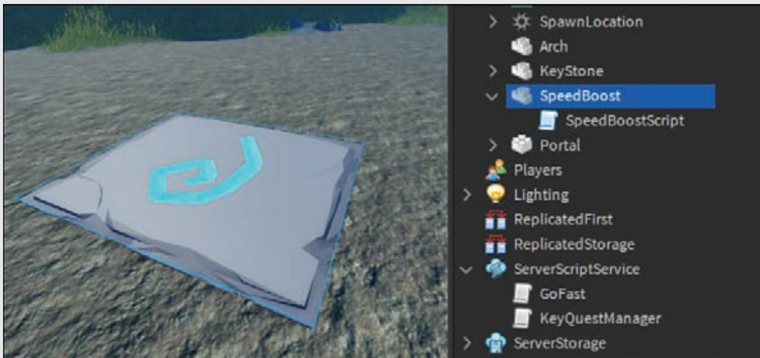


FIGURE 6.8
Insert a new script into a mesh or part.

2. Give people a temporary speed boost after having touched the part. You can use the code snippets from earlier in this chapter, or if you did the Try It Yourself in Hour 5, use that and modify the code to use constants.
3. Experiment with the values for `BOOSTED_SPEED` and `BOOST_DURATION` by doubling the values for both `BOOSTED_SPEED` and `BOOST_DURATION`.
4. Test and see the results. If it doesn't seem fast enough or boosted for long enough, double the amount again. If it feels like too much, subtract half of what you added.

Checking Attribute Values

Most variable values can be printed, but attributes behave a little bit differently. You need to assign the attribute's value to a variable first:

```
local activatedValue = weapon:GetAttribute("Activated")
print(activatedValue)
```

Keep this in mind while confirming attribute values.

Getting the Right Types of Values

You also need to be careful with what types of values are being passed back into the functions.

Good code takes into account that errors happen when incorrect value types are passed in. If you were to try to pass a string into `wait()`, the string is ignored, and a built-in default value of a thirtieth of a second is used:

```
local part = script.Parent
wait("twenty") -- Will use default value because strings aren't accepted
part.Color = Color3.fromRGB(170, 0, 255)
```

Summary

There are a lot of ways to make sure your code runs only once using different types of debounce systems. One way you may have used before is deleting a part as soon as something touches it. Two other ways used in this hour are setting up a debounce variable and using a proximity prompt with a long hold. No matter what way is used, always think about how your choices will affect your end user.

A big part of being a coder is thinking about all the possible scenarios that might come up and trying to create code that won't break while still giving your users the best possible experience. It is, of course, totally normal for things to go wrong. It happens to the very best coders—even the ones who make your favorite Roblox experiences.

If things aren't going as anticipated, you can use your knowledge of scope, functions, and events to narrow down the problem. A few well-placed `print` statements can help you verify whether your functions are called and if values are what you expect.

Q&A

Q. In leaderboards, can you use values other than IntValues?

A. Yes, you can create other types of values. For example, in Figure 6.9, a StringValue was used to display the faction name of the character.

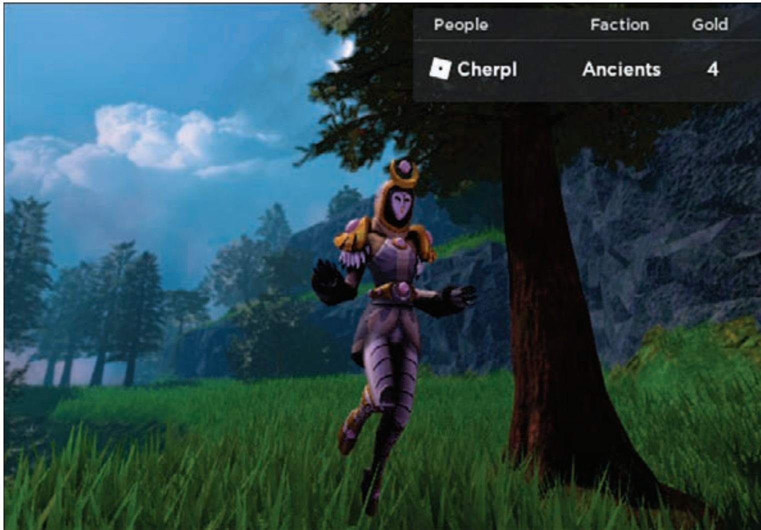


FIGURE 6.9
The leaderboard at the top right uses strings to display faction names and integers to display owned gold.

Q. Is there a maximum amount of stats that can be displayed?

A. A maximum of four stats can be displayed, although additional stats can still be tracked.

Q. Can you create your own custom leaderboards?

A. You can! In later hours, you learn more about how to display information to individual people and to everyone in the server as a whole.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. What is it called when you make sure code can only be triggered once and not multiple times?
2. What's a name for variables whose values don't change as the script runs?
3. How are the variables in the preceding question formatted compared to other variables?
4. What's an easy technique for figuring out what value numbers to use when tweaking code for a better user experience?
5. How do you print the value of an attribute?

Answers

1. Debouncing
2. Constants
3. ALL_CAPS, with an underscore (_) between words
4. Doubling and halving
5. You need to first assign the attribute to a variable, and then print the variable:

```
local armorValue = Helm:GetAttribute("Armor")
print(armorValue)
```

Exercises

A large part of any engineer's job is thinking about both what can go wrong and how to make things better. Think about the code you've created in all of the hours up until now, and for this first exercise, write down at least three ways in which the code could be better. It might be things that can go wrong with the code, or features that would enable people to better enjoy your experience.

You might not be able to code the solutions just yet, but you should get in the habit of being critical of the code you're creating.

You can find possible answers in the appendix.

For the second exercise, make two pickups: one that makes you smaller and one that makes you larger (see Figure 6.10). Instead of setting three specific sizes, use a multiplier to change the avatar's current scale.

FIGURE 6.10
A giant mech avatar strolls through a city after being enlarged.

Remember to create a debounce variable to control how fast the person is allowed to grow and shrink. This is one case where if the function is triggered more often than intended, your experience *will* crash.

Tips

- You can modify the user's default scale with the following properties:
  - `Humanoid.HeadScale`: Scale of the avatar's head.
  - `Humanoid.BodyDepthScale`: Scale of the body's depth.
  - `Humanoid.BodyWidthScale`: Scale of the body's width.
  - `Humanoid.BodyHeightScale`: Scale of the body's height.
- Set up a simple debounce before experimenting with the avatar's scale to avoid crashes.
- Save your work before testing! If the avatar's scale gets too big, it *will* crash your experience.

This page intentionally left blank

HOUR 7
while Loops

What You'll Learn in This Hour:

- What a while loop is
- How to make tasks repeat forever and ever
- How to create a fire that requires fuel to stay burning
- How to plan for scope with while loops.

Do you ever feel like you're stuck in a loop where you just keep doing the same thing over and over and over and over? Get up, eat breakfast, work hard, go back to bed, and then the same thing all over again the next day. We see loops throughout our world. The minutes on our clocks loop through 60 minutes, and the hours loop 24 times to create a day.

Scripts have loops as well. When they're inside of the loop, they keep doing the same task until something makes them stop. This hour covers just one kind of loop that can be found in code: while loops.

Repeat Forever, while true do

The first kind of loop in this hour is a while loop. These loops are typically used to check on the state of something and run indefinitely until a condition is met. They can even run forever! The following code snippet shows you how a while loop is formatted:

```
local isHungry = true

while isHungry == true do
    print("I should eat something")
    wait(2.0)
end
```

The main keywords here are while and do. In the middle of those keywords is the condition for the while loop to check against. As long as that condition is true, the code keeps running. In fact, if you want the code to run forever, you can simply set the condition to true:

```
while true do
    print(count)
```

```
    count = count + 1
    wait(1.0)
end
```

The preceding example would count every second and display the result in the output until you stop the playtest.

Some Things to Keep in Mind

There's a couple of things to keep in mind when working with `while` loops. One is that every `while` loop should include a `wait` function. If you don't, there's a good chance the loop will run so quickly that your experience will end up either slowing down or crashing. The other thing to keep in mind is that the next loop is started as soon as the previous loop finishes.

▼ TRY IT YOURSELF

Create a Disco Dance Floor

Take a quick example where you're creating a disco dance floor, and you want the floor pieces to loop through a series of specific colors—in this case, a pattern of blue and orange:

1. Use a part to act as a section of the floor and insert a script with the following code:

```
local discoPiece = script.Parent

while true do
    discoPiece.Color = Color3.fromRGB(0, 0, 255)
    wait(1.0)
    discoPiece.Color = Color3.fromRGB(255, 170, 0)
end
```

2. Run the code. The only color you see while the code runs is blue. Because the next loop starts immediately, the orange blinks so fast it's not even visible.

3. Fix this by adding a second `wait` function after the color change:

```
local discoPiece = script.Parent

while true do
    discoPiece.Color = Color3.fromRGB(0, 0, 255)
    wait(1.0)
    discoPiece.Color = Color3.fromRGB(255, 170, 0)
    wait(1.0)
end
```

If the entire loop only needs a single `wait` function, it can be worked into the condition. This is demonstrated in the following code chunk that assigns a new random color every second for a floor like the one in Figure 7.1:

```
local discoPiece = script.Parent

while wait(1.0) do
    -- Get random values for RGB
    local red = math.random(0, 255)
    local green = math.random(0, 255)
    local blue = math.random(0, 255)
    -- Assigns color values
    discoPiece.Color = Color3.fromRGB(red, green, blue)
end
```

FIGURE 7.1
A `while` loop and random number generation are used to create an ever-changing disco floor.

TRY IT YOURSELF ▼

Keep the Campfire Burning

This Try It uses a `while` loop to keep track of how many fuel logs are on the fire. The fire is disabled until someone uses the ProximityPrompt to add fuel. Then the fire burns for a little while before going out again. (See Figure 7.2.)

FIGURE 7.2
A fire where fuel will be spent over time using a `while` loop.

Set Up

First, set up the fire and the ProximityPrompt. Once everything is scripted, the fire can be copied into any environment or model you please:

1. Use an invisible part to hold your fire.
2. Add a new attribute to the CampFire part:

   Name: **Fuel**

   Type: **Number**

3. Insert a Particle Emitter named **`Fire`**, and a ProximityPrompt named **`AddFuel`**. (See Figure 7.3.)

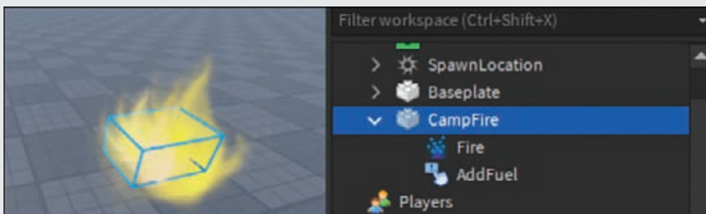


FIGURE 7.3
Use an invisible part with a ParticleEmitter and ProximityPrompt inserted.

TIP

Designing a Fire

For the fire particles, setting the Texture property to 4797593940 and Speed to 0 will help you get a particle like the one shown in the example. After that, try playing with the color, drag, and lifetime values.

4. In the ParticleEmitter's properties, uncheck Enabled because it'll be turned on within the script.

5. In the ProximityPrompt's properties, change HoldDuration to `2`.

The Script

When the ProximityPrompt is triggered, fuel is added to the fire, and the fire is enabled. A `while` loop spends fuel every second, and when the fuel reaches 0, the fire is disabled:

1. In ServerScriptService, add a new script.

2. Get `ProximityPromptService` and set up a function that is called when the prompt is triggered. Inside, make sure the prompt is enabled and confirm that the triggering prompt is `"AddFuel"`:

```
local ProximityPromptService = game:GetService("ProximityPromptService")

local BURN_DURATION = 3

local function onPromptTriggered(prompt, player)
    if prompt.Enabled and prompt.Name == "AddFuel" then

    end
end

ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)
```

3. Create a constant to control how long the fire will burn; inside the `if` statement, create variables for the campfire part and the fire particles:

```
local ProximityPromptService = game:GetService("ProximityPromptService")

local BURN_DURATION = 3

local function onPromptTriggered(prompt, player)
    if prompt.Enabled and prompt.Name == "AddFuel" then
        local campfire = prompt.Parent
        local fire = campfire.Fire -- This should be the particle emitter
    end
end

ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)
```

4. Get the current value of the `Fuel` attribute, and add 1:

```
local function onPromptTriggered(prompt, player)
    if prompt.Enabled and prompt.Name == "AddFuel" then
        local campfire = prompt.Parent
        local fire = campfire.Fire -- This should be the particle emitter
```

```
            local currentFuel = campfire:GetAttribute("Fuel")
            campfire:SetAttribute("Fuel", currentFuel + 1)

        end
    end
```

5. Use another `if` to check whether the particles are off, and if so, turn the particles on:

```
local function onPromptTriggered(prompt, player)
    if prompt.Enabled and prompt.Name == "AddFuel" then
        local campfire = prompt.Parent
        local fire = campfire.Fire -- This should be the particle emitter

        local currentFuel = campfire:GetAttribute("Fuel")
        campfire:SetAttribute("Fuel", currentFuel + 1)

        if not fire.Enabled then
            fire.Enabled = true
        end
    end
end
```

6. Burn off one piece of fuel at a time with a `while` loop, and then disable the particles:

```
local ProximityPromptService = game:GetService("ProximityPromptService")

local BURN_DURATION = 3

local function onPromptTriggered(prompt, player)
    if prompt.Enabled and prompt.Name == "AddFuel" then
        local campfire = prompt.Parent
        local fire = campfire.Fire -- This should be the particle emitter

        local currentFuel = campfire:GetAttribute("Fuel")
        campfire:SetAttribute("Fuel", currentFuel + 1)

        if not fire.Enabled then
            fire.Enabled = true
            while campfire:GetAttribute("Fuel") > 0 do
                local currentFuel = campfire:GetAttribute("Fuel")
                campfire:SetAttribute("Fuel", currentFuel - 1)
                wait(BURN_DURATION)
            end
            fire.Enabled = false
        end
    end
end

ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)
```

Check your work. If the UI is getting in the way of seeing the fire, you can move it higher in the prompt's properties by using the UIOffset (see Figure 7.4).

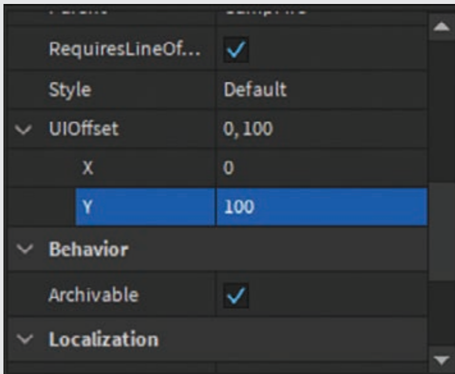


FIGURE 7.4
The prompt's UIOffset property can be used to move it out of the way.

Once you know the campfire works as intended, you can add it to fancier environments like those shown in Figure 7.5.

FIGURE 7.5
Fire inserted into a great chalice on the left and a hearth on the right.

If you want to expand on this, you could have players collect wood from nearby trees before being able to light the fire.

`while` Loops and Scope

One last thing you have to know about `while` loops is that any code beneath a `while` loop will never run unless the loop is broken:

```
print("The loop hasn't started yet") -- Will run once
while wait(1.0) do
     print("while loop has looped") -- Will run until the server stops
end
print("The while loop has stopped looping ") -- Will never run
```

Summary

As you create more experiences, you'll find more instances of when you want code to keep repeating forever or under certain circumstances. Some loops will be small and quick, like a loop creating a flickering light. Other loops will be longer and control the entire flow of a game—for example, the loops found in round-based games where people wait in a lobby for a certain amount of time and then are transported to wherever the action is. At the end of the round, everything is cleaned up, people are sent back to the lobby, and then the loop starts over again.

Of course, there are things to keep in mind when using `while` loops. Because a `while` loop runs forever, code beneath the loop will never be reached unless the loop stops. It's also its own code chunk, so you need to keep in mind how that affects scope.

If you don't want the loop to begin as soon as the server is launched, you can always wrap the `while` loop in a function if you want to control when it starts.

Q&A

Q. What if you want a piece of code to repeat only a certain number of times?

A. If you want a piece of code to repeat a certain number of times—for example, if you want to create exactly ten trees—you can use what's called a `for` loop. `for` loops are covered in Hour 8.

Q. What if you want a loop to run while something is false instead of while true?

A. If you want a piece of code to run while a condition is false, you have a couple of options. The first is that you can set a condition such as `while NumberOfPlayers ~= 0 do`. Here, a piece of code runs as long as the number of players *is not equal* to zero. Alternatively, you can use `repeat action until(condition)`, which instructs a piece of code to repeat indefinitely until a condition becomes true.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. How long will a `while` loop run?
2. What must always be included in a `while` loop and why?
3. How often will the following loop print hello?

```
while wait(1.0) do
    print("hello")
    wait(1.0)
end
```

4. How many colors will the `discoFloor` referred to in this code turn?

```
local discoFloor = script.Parent

while wait(2.0) do
    print("hello")
end

while true do
    discofloor.Color = Color3.fromRGB(0, 0, 255) -- Blue
    wait(1.0)
    discofloor.Color = Color3.fromRGB(255, 255, 0) -- Yellow
end

discofloor.Color = Color3.fromRGB(255, 0, 127) -- Pink
```

Answers

1. Until the given condition is false.
2. A `wait` function must always be included; otherwise, the code loop will run faster than the engine can handle and crash.
3. `Hello` will print every 2 seconds. There's a one-second wait in the condition, and a one-second wait in the loop. The second wait isn't needed, however. It could just be a two-second wait in the condition.
4. The floor will never change colors. The first loop prevents the second loop from ever running. If that wasn't there, however, it would appear blue. Yellow would flash by too quickly to see, and pink is outside the scope of the loop.

Exercises

In this first exercise, modify the code so that people have to collect wood for the fire rather than being able to simply walk up to a fire and light it (see Figure 7.6).

FIGURE 7.6
Logs can be collected from the tree and used to fuel the campfire.

Tips

- Use the leaderboard to track how much wood the player has.
- You can use nearly the exact code and set up that was used for ore in the last hour to collect the logs.
- Modify the campfire script so that it takes logs from the player to use as fuel.

A universal truth to coding and design is that you're going to find yourself wanting to update things later. The more copies you have of something in your game, the harder making updates becomes, whether scripts, particles, or models. For the second exercise, try updating the fire script so that instead of enabling an existing particle emitter, it inserts a cloned particle emitter into the campfire.

Tips

- You still need a part to hold the ProximityPrompt.
- See if you can remember how to clone things out of ReplicatedStorage.

HOUR 8
`for` Loops

What You'll Learn in This Hour:

- How to repeat tasks with `for` loops
- How to use nested loops
- How to exit nested loops
- How to create displays for information
- How to do damage over time

So far, we've covered one type of loop—the `while` loop, which can go forever and ever and ever if that's what you want it to do.

If you want to make sure that code updates only a certain number of times, you use a different kind of loop: a `for` loop. Unlike `while` loops, `for` loops repeat themselves a certain number of times until a goal is reached.

Figure 8.1 shows a `for` loop being used to count down until an expected meteor collision.

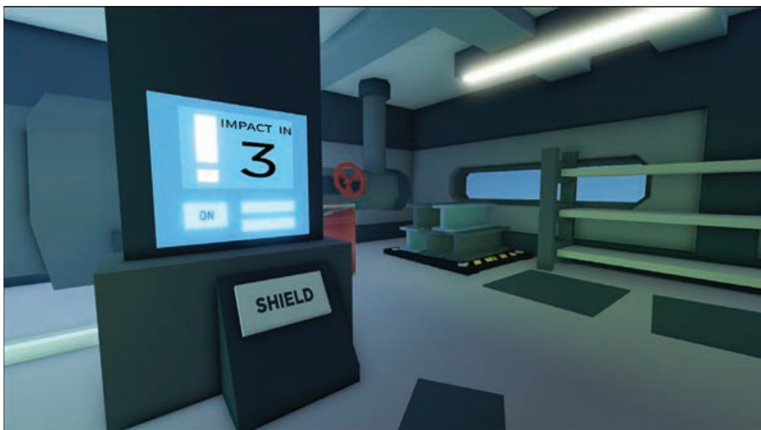


FIGURE 8.1
A clock uses a `for` loop to show three seconds until impact.

▼ TRY IT YOURSELF

Create a Countdown

Test out this simple `for` loop that counts down to 0. The individual parts of the code will be explained in the next section:

1. In any script, copy the following:

```
for countDown = 10, 0, -1 do
    print(countDown)
    wait(1.0)
end
```

2. Run the code. In the Output window, you should see a countdown like the one in Figure 8.2.

```
Output
Filter...
21:48:42.992 Western auto-recovery file was created - Studio -
21:48:43.791 10 - Server - CountDown:2
21:48:44.867 9 - Server - CountDown:2
21:48:45.870 8 - Server - CountDown:2
21:48:46.872 7 - Server - CountDown:2
21:48:47.885 6 - Server - CountDown:2
21:48:48.886 5 - Server - CountDown:2
21:48:49.902 4 - Server - CountDown:2
21:48:50.918 3 - Server - CountDown:2
21:48:51.937 2 - Server - CountDown:2
21:48:52.954 1 - Server - CountDown:2
21:48:53.958 0 - Server - CountDown:2
```

FIGURE 8.2
Numbers count down one by one from 10 to 0.

How `for` Loops Work

A `for` loop uses three values to control how many times it runs, which are formatted as shown in Figure 8.3:

- **Control variable:** Tracks the current value. The assigned value marks the starting place. A control variable can be any acceptable variable name. Like other variable names, a control variable name should be clear and descriptive about what the `for` loop is doing.
- **End or goal value:** The value at which the loop should stop running. The script checks the control variable against the end value before starting the next loop.
- **Increment value:** The amount by which the control variable changes every time. Positive increment values count up; negative increment values count down.

Control Variable — End Value — Increment Value

```
for count = 0, 10, 1 do
    print("This is a for loop")
end
```

FIGURE 8.3
The three values that control how many times a `for` loop runs are the control value, the end value, and the increment value.

Beginning at the initial value of the control variable, the `for` loop counts toward the ending goal value, stopping once the goal value is reached:

1. The `for` loop compares the control variable with the end value. (See Figure 8.4.)

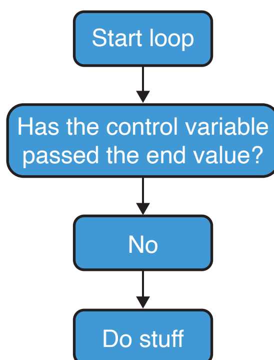


FIGURE 8.4
Before executing the code in the loop, the control variable is checked against the goal value.

2. After running the code, the increment value is added to the control variable. The loop then checks the control variable and starts over. (See Figure 8.5.)

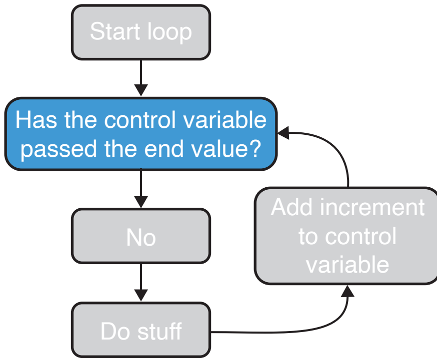


FIGURE 8.5
At the end of the loop, the increment value is added to the control variable.

3. Once the control variable passes the end value, the loop will stop. For example, if a loop has an end value of 10, once the control variable has passed 10, the `for` loop will stop (see Figure 8.6).

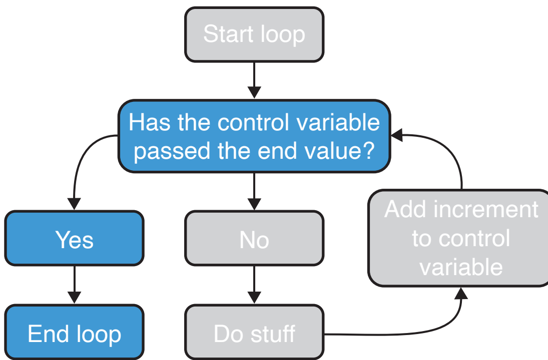


FIGURE 8.6
This is the flow of a complete `for` loop process.

Let's take another look at the Output shown in the Try It Yourself, displayed in Figure 8.7.

FIGURE 8.7
This output of a `for` loop counts down every second.

The loop that ran each time a number was printed is called an iteration. An *iteration* is the complete process of checking the control value, running code, and updating the increment value. Since the count started at 0 and ended after 10, the code actually went through eleven iterations.

Keep this in mind as you design your loops. If it's important for a count to go a specific number of times, you'll probably want the starting value to be 1 instead of 0.

Increments Are Optional

If an increment value isn't included, the default value of 1 is used. The code snippet begins at 0 and counts upward to 10:

```
for countUp = 0, 10 do
    print(countUp)
    wait(1.0)
end
```

Different `for` Loop Examples

Changing the values of the control variable, end goal, and increment changes how the loop functions. The `for` loop you just wrote could instead count up to 10 or count down in odd numbers. The following are examples of `for` loops with different start, end, and increment values.

Counting Up by One

```
for count = 0, 5, 1 do
    print(count)
    wait(1.0)
end
```

Counting Up in Even Numbers

```
for count = 0, 10, 2 do
    print(count)
    wait(1.0)
end
```

Be careful not to reverse the starting and goal values, like so:

```
for count = 10, 0, 1 do
    print(count)
    wait(1.0)
end
```

If the control variable starts out beyond the end value, like in the earlier example, the `for` loop doesn't run at all. In this case, the `for` loop is counting up and checking if `count` is greater than 0. When the `for` loop does its first check, it sees that 10 is greater than 0, so it stops the loop without printing anything.

▼ TRY IT YOURSELF

In-World Countdown

So far, messages have only been displayed within the Output window. Now it's time to start communicating information to people in your environments. In this Try It Yourself, you use a graphical user interface (GUI) to display information where everyone can see it. GUIs are like sticker labels that can be used to display information within the world.

Setup

For the setup, you create a SurfaceGui and TextLabel and size them to the part to display the countdown. Since this is a coding book, we won't get too much into how these work. If you want to know more, you can find more detailed explanations on the Roblox Developer Hub:

1. Create a new part.
2. Insert a SurfaceGui object into the part. Nothing obvious happens, but SurfaceGUI objects act as containers for anything you want to display.

3. Select SurfaceGui and insert a TextLabel object. This displays the actual text. (See Figure 8.8.)

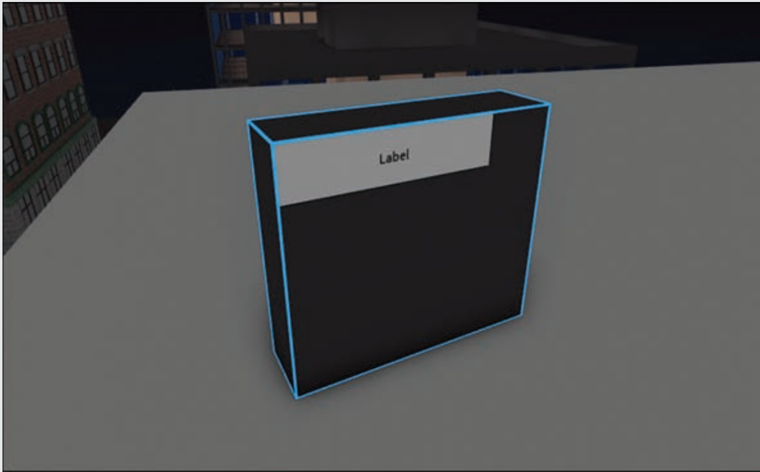


FIGURE 8.8
The TextLabel is added on the front of a part.

TIP

Finding the TextLabel

If you can't see the TextLabel, it probably appeared on a different side of the part. You can rotate the part or change the SurfaceGui's Face property to fix it.

4. Select the TextLabel. In Properties, expand Size. For X Scale, type `1`, and in Offset, type `0`. Do the same for Y. This should make the TextLabel take up the entire side of the part. (See Figure 8.9.)

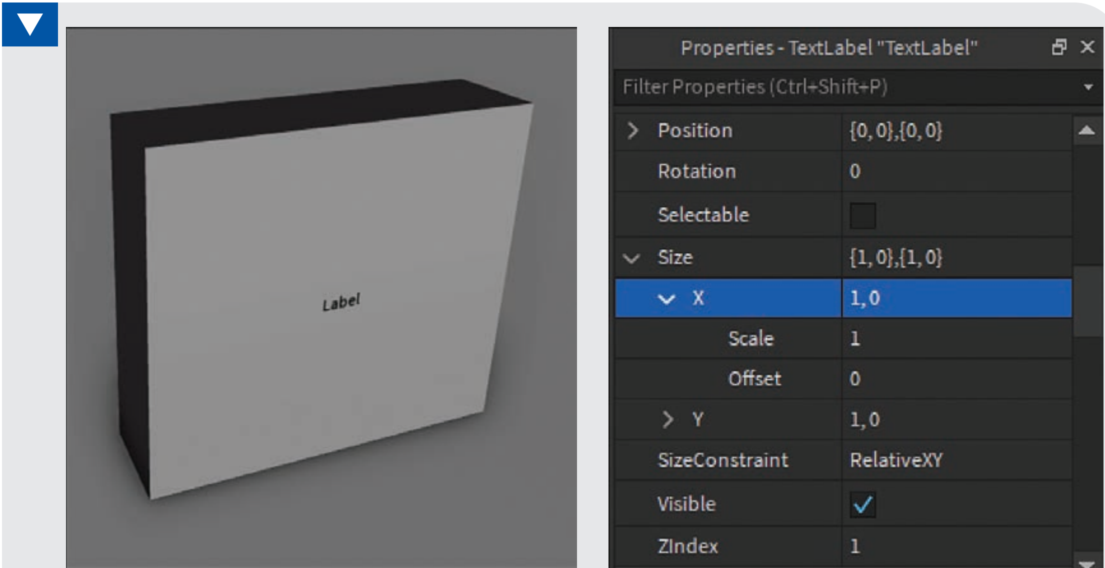


FIGURE 8.9
The TextLabel takes up the entirety of the side.

5. Still in TextLabel's properties, scroll almost to the bottom to TextScaled and enable it. This sizes the font to fit as shown in Figure 8.10.

FIGURE 8.10
The text is automatically scaled to fit the entire TextLabel.

Code the Countdown

You use a script to change what the TextLabel displays:

1. Select the sign part and insert a new script.
2. Use variables to reference the script's parent and the TextLabel. *Hint: You can go down the hierarchy a couple of times.*
3. Create a new `for` loop that counts down every second:

```
local sign = script.Parent
local textLabel = sign.SurfaceGui.TextLabel

for countDown = 10, 1, -1 do
    print(countDown)
    wait(1.0)
end
```

4. Within the loop, set the TextLabel's Text property to the current value of the countdown:

```
local sign = script.Parent
local textLabel = sign.SurfaceGui.TextLabel

for countDown = 10, 1, -1 do
    textLabel.Text = countDown
    print(countDown)
    wait(1.0)
end
```

5. Test your code.

TIP

A Note on Load Times

You may notice that sometimes the count seems to start in the middle. That's because the script started before your character and camera loaded all the way in. You can verify that the countdown ran correctly by using a `print` statement or delay the beginning of the count with a small pause at the beginning of the script. As you create more scripts, you begin having to take into account load times more often.

Nested Loops

Loops can be used within loops. One of the most common ways you see this done is placing a `for` loop inside of a `while` loop. This way, you can repeat events that repeat every so often, such as firework shows:

```
while true do
    for countDown = 10, 1, -1 do
```

```
        textLabel.Text = countDown
        print(countDown)
        wait(1.0)
    end

    print("Launch the rockets!")
    wait(2.0)
end
```

When loops are nested, the script starts from the top line and works its way down. When a new loop is reached, that loop runs to completion before continuing with the next lines of code.

Breaking Out of Loops

If for some reason you need to leave a loop, use the keyword `break`:

```
local goodToGo = true

while wait(1.0) do
    if goodToGo == true then
        print("Keep going")
    else
        break -- will stop loop if goodToGo changes to false
    end
end
```

Summary

Loops are everywhere in code. They can run forever or a set amount of times; it just depends on what type of loop you use. `while` loops keep running unless the initial condition becomes `false` or the keyword `break` is used. This type of loop gets used for things like a day/night cycle, which only ends when the world ends.

On the other hand, `for` loops are best used when you're trying to reach a specific value, like counting down to midnight on New Year's Eve.

Q&A

Q. Why do some people just type *i*?

A. There's a bit of controversy over exactly what `i` stands for, but a common theory is that it originally stood for integer. `i` was first used as a stand in for unknown numbers by ancient mathematicians and then by early computer programmers who had to keep their code very, very brief. In short, it's just a common control variable name, which is why you might sometimes see `for` loops that look like this:

```
for i = 1, 10 do
print(i)
end
```

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. How long will a `for` loop run?
2. What is an increment?
3. How many times will this code loop (backward starting values)?

```
for count = 10, 0, 1 do
    print(count)
end
```

4. True or false: Increment values are optional.

Answers

1. Until the given condition is reached.
2. The amount by which a value changes.
3. Zero times. The starting value of 10 is greater than the goal value of 0.
4. True. The default value of 1 is used if no increment value is given.

Exercises

The concept of Damage Over Time (DoT) is used in lots of experiences. With DoT, people take ongoing damage for a certain amount of time rather than taking it all at once. Common examples include encountering poison or taking burn damage after touching a fire.

Because you already have a fire from an earlier Try It Yourself, for this first exercise, use the same model to temporarily inflict burn damage to anyone who happens to touch it.

Tips

- Use the same fire you created previously or use a part to act as a stand-in.
- Insert a new invisible part named HitBox. Scale it to encompass the fire. (See Figure 8.11.)
- If somebody touches HitBox, use a `for` loop to inflict 10 points of damage every second for three seconds.

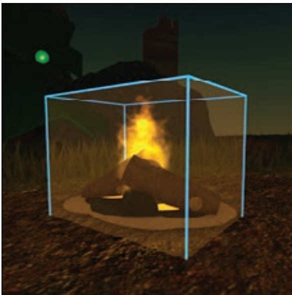


FIGURE 8.11
An invisible box is used to mark the boundaries of the fire.

For the second exercise, take a moment to think of at least five other ways that you can use `for` loops and `while` loops in your 3D Experiences. Don't worry about whether you know how to create the code. The important thing here is to be able to start recognizing where they might be found.

Solutions for the exercises are in the back of the book.

HOUR 9

Working with Arrays

What You'll Learn in This Hour:

- How to create and use arrays
- How to loop through arrays with `ipairs`
- How to make changes to arrays

Now it's time to work with multiple objects at once so you can do things like give every member of the team a shiny new weapon or modify every item in a folder. You handle tasks like this with tables. Tables allow you to organize multiple pieces of data or objects into groups, such as groups of players or a list of item requirements for a recipe.

This hour covers the first of two different table types: arrays. You'll learn how to make changes to a whole folder full of objects by turning on multiple lights at once rather than making people turn them on individually.

What Are Arrays?

Arrays create a numbered list of items that can be used to keep track of information, such as who's in first place or a folder full of different parts.

Every item on the list has a specific number assigned to it, called an *index*. If you had a grocery list, it might look something like the following table:

GroceryList

Index	1	2	3
Value	Apples	Bananas	Carrots

Creating an array is the same as creating other variables; the only difference is it gets assigned curly brackets, like so:

```
local myArray = {}
```

The curly brackets are what make it a table data type. Items can be added to the array by listing them within the brackets, although you need to be sure to separate values with commas. The index number is assigned automatically in the order in which the values are added. Here's an example of three-item array:

```
local groceryList = {"Apples", "Bananas", "Carrots"}
```

Arrays can hold any value type—even other arrays. The third array in the following example contains the first two arrays, and a fourth unnamed array is assigned to index 3:

```
local firstArray = {1, 2, 3}
local secondArray = {"first", "second", "third"}
local thirdArray = {firstArray, secondArray, {"unnamed array"}}
```

Adding Items Later

You can add an item to an already-created array by using `table.insert(array, valueTo-Insert)`. So, adding a new item to the previous array looks like this:

```
local groceryList = {"Apples", "Bananas", "Carrots"}
table.insert(groceryList, "Mangos")

print(groceryList)
```

New items get added to the end of the array.

Getting Information from a Specific Index

You can test the list by printing out a few indexed values. To use a value at a specific index, add the index after the array's name without any spaces, like `arrayName[1]`:

```
local groceryList = {"Apples", "Bananas", "Carrots"}
table.insert(groceryList, "Mangos")

print(groceryList[1], groceryList[4], groceryList[5])
```

As you can see in Figure 9.1, the value at index 1—and the value at index 4, which was added to the table later—were both printed. No value was found at index 5, so nil was returned.

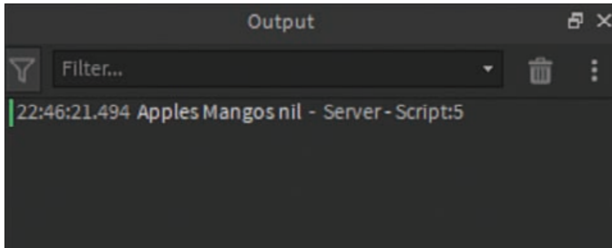


FIGURE 9.1
The first two array values are displayed, but the third value is nil because it doesn't exist.

Printing an Entire List with `ipairs()`

The easiest way to print out the entirety of the list is with a special type of `for` loop that uses the function `ipairs()`. The pattern looks like this:

```
for index, value in ipairs(arrayName) do
    -- Do something
end
```

The components of the pattern are as follows:

- `index`: This references the current index the loop is working through. It can be any valid variable name. People often just use the lowercase letter `i`.
- `value`: References the value of the current index. This can also be any valid variable name.
- `in ipairs(arrayName)`: `in` is a keyword and can't be changed. `ipairs()` takes in the name of the array you want to work with.

So, if you have a list of player names, and you want to print them out in order, it might look like this:

```
local players = {"Ali", "Ben", "Cammy"}
for playerIndex, playerName in ipairs(players) do
   print(playerIndex .. " is " .. playerName)
end
```

TIP

Generic Loops

Sometimes you'll see this type of loop referred to as a generic loop.

Folders and `ipairs()`

A really handy way to use `ipairs()` is to modify everything in a folder. You can get a list of every object in a folder, in order, using `GetChildren()`, which returns an array.

Let's say you have a folder full of parts, and you want every part in that folder to turn a different color. You can use something like this code snippet:

```
local folder = workspace.Folder -- Make sure to use the name of your folder

local arrayTest = folder:GetChildren() -- GetChildren() returns an array

for index, value in ipairs(arrayTest) do
    if value:IsA("BasePart") then -- checks to see it's a part
        value.Color = Color3.fromRGB(0, 0, 255)
        print( "Object " .. index .. " is now blue")
    end
end
```

TRY IT YOURSELF

Turn On the Kitchen Lights

In this Try It Yourself, you have a number of lights in a kitchen (see Figure 9.2) that should all turn on using the same switch. You've learned before that putting a script into every single one of these lights gets messy and makes things difficult to update. You could put a proximity prompt into each light, but then people would have to go around and turn all the lights on one-by-one.

One way to organize objects is to put them into a folder and use a `for` loop to update everything in that folder at once when somebody flips the switch.

FIGURE 9.2
A kitchen scene softly lit with track lighting. All the lights are controlled by a single switch.

1. Find a part to act as a light. In Figure 9.3, a small glass cylinder part is acting as a prop for the lens of the light. Insert a SpotLight into the part.

TIP

SpotLights

SpotLights shine a cone of light, like a flashlight.

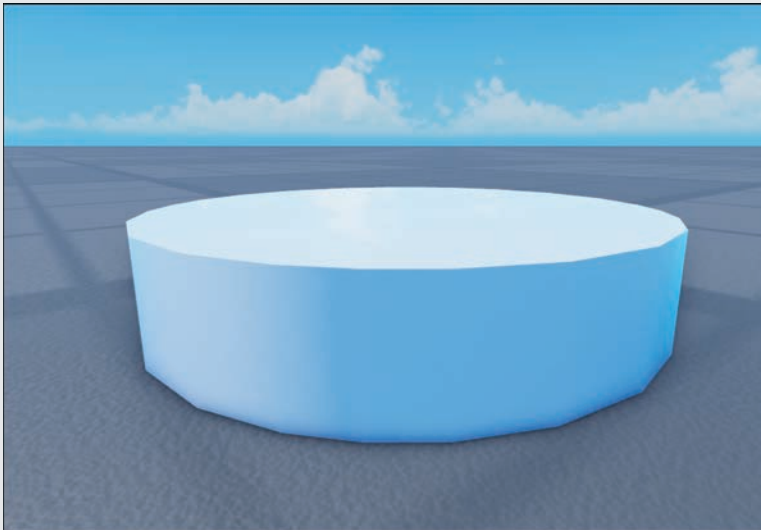


FIGURE 9.3
A tiny one-stud-wide glass disk that can be used as a light source.

2. To modify which direction the SpotLight shines, in Properties > Face, use the drop-down menu to select the correct face that makes the light appear to shine downward. For this example, that's Left. (See Figure 9.4.) Yours may be different.

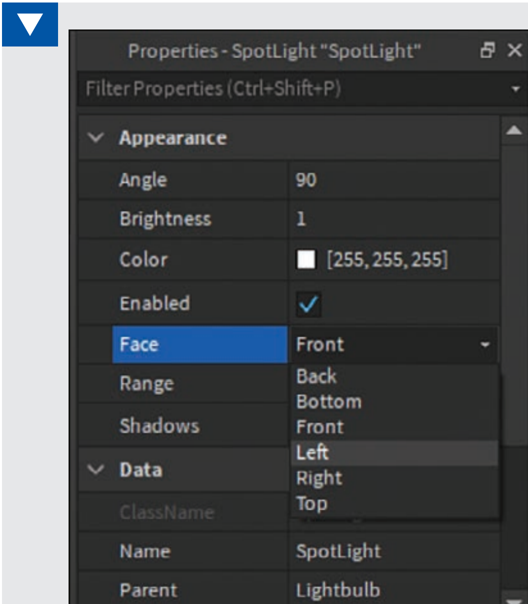


FIGURE 9.4
Use SpotLight's Face property to control which direction the light shines in.

3. With SpotLight still selected, in Properties, increase Brightness and Range until it's right for the scene. (See Figure 9.5.)

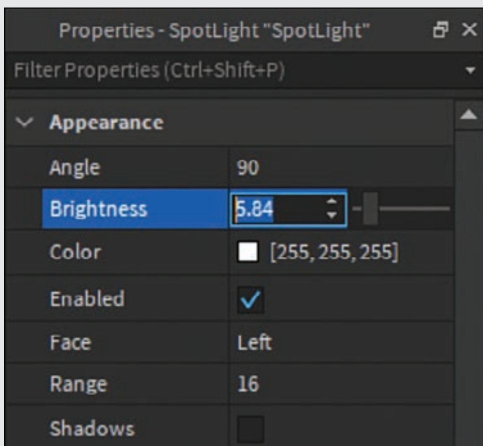


FIGURE 9.5
Increase the Brightness of the SpotLight to make it brighter and the Range to make the light reach farther.

4. Duplicate the light around your scene. You can even use different models. In Figure 9.6, the disk has been copied into track lighting around the ceiling of this kitchen.

FIGURE 9.6
Light props have been placed into track lighting within a kitchen scene.

5. Create a new folder called Lights, and move all of the lightbulbs into the folder. (See Figure 9.7.)

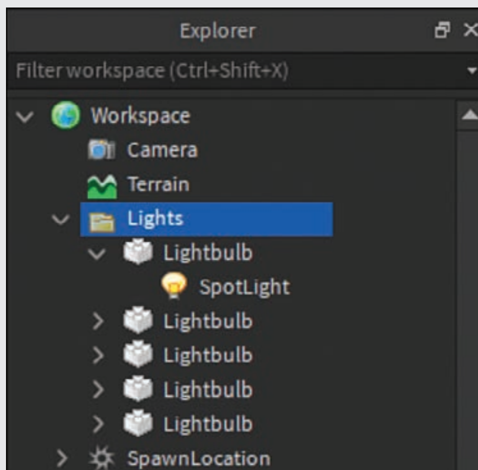


FIGURE 9.7
All of the lightbulbs have been moved into a single folder.

Turn the Lights On and Off

This script goes through each object in the Lights folder and checks to see if it has a spotlight. If it finds a spotlight, the script turns it on or off.

1. In ServerScriptService, create a new script.
2. Create a new variable that references the Lights folder.
3. Create a second variable to get an array of all of the folder's children:

```
local lightsFolder = workspace.Lights
local lights = lightsFolder:GetChildren()
```

4. Create a new `for` loop using `ipairs()` and pass in the array:

```
local lightsFolder = workspace.Lights
local lights = LightsFolder:GetChildren()

for index, lightBulb in ipairs(lights) do

end
```

5. Inside of the `for` loop, use `FindFirstChildWhichIsA()` to find the SpotLight nested inside of the lightbulb:

```
local lightsFolder = workspace.Lights
local lights = LightsFolder:GetChildren()

for index, lightBulb in ipairs(lights) do
    local spotLight = lightBulb:FindFirstChildWhichIsA("SpotLight")

end
```

6. Set up the following three conditions:

   a. If the spotlight is found and the light is off, enable the SpotLight.

TIP

Glowing Spotlight

If you're using a part, you can also change the material to neon to make it appear to be glowing.

   b. If the spotlight is found, and the light is on, disable the SpotLight.

   c. If the loop finds something in the folder that does not have a SpotLight, print "Not a lightbulb."

Try to do this on your own before looking at the following code:

```
local lightsFolder = workspace.Lights
local lights = LightsFolder:GetChildren()

for index, lightBulb in ipairs(lights) do
    local spotLight = lightBulb:FindFirstChildWhichIsA("SpotLight")

    if spotLight and not spotLight.Enabled then
        spotLight.Enabled = true
        lightBulb.Material = Enum.Material.Neon -- Makes it look glowy

    elseif spotLight and spotLight.Enabled then
        lightBulb.Material = Enum.Material.Glass
        spotLight.Enabled = false

    else
        print ("Not a light")
    end
end
```

Test your code by turning on some of the lights, disabling others, and throwing a random part into the Lights folder. If it works as intended, place your code inside of a function to be run when somebody interacts with a proximity prompt, as shown in the last few hours. If you've forgotten how, look at your previous code, or look in the appendix at the end of the book.

Finding a Value on the List and Printing the Index

Say you have a bunch of customers in line waiting for their table. One of them walks up and wants to know their place in line. You know the customer's name, but not their number. In this case, the waiting list is just another array. You can use `ipairs` again to look up the customer's place by checking for the matching value:

```
local waitingList = {"Ana", "Bruce", "Casey"}

-- Let's find Casey's place in line

for placeInLine, customer in ipairs(waitingList) do
    if customer == "Casey" then
        print(customer .. " is " .. placeInLine)
    end
end
```

Removing Values from an Array

To remove a value, like if a player used an item or someone in a list of active players leaves a game, use `table.remove(arrayName, index)`. This function either removes the last value of a table or removes it at a specific index depending on whether both parameters are used.

```
local playerInventory = {}
table.insert(playerInventory, "Health Pack")
table.insert(playerInventory, "Stamina Booster")
table.insert(playerInventory, "Cell Key")

table.remove(playerInventory)  -- No index, so last item will be removed
table.remove(playerInventory, 2) -- Will remove the second item
```

The second parameter for `table.remove()` only accepts a numerical index. Typing something like `table.remove(playerItems, "Health Pack")` returns an error. You can try printing the results of the table to confirm everything works as expected.

When an item is removed from an array, the rest of the values will shift to fill in the gap. You can test this by printing the array before and after the item is removed. Of course, we don't want to type the code for printing an array more than once, so in the following code snippet, it's part of a function that can be called as often as you want:

```
local function printArray(arrayToPrint)
    for index, value in ipairs(arrayToPrint) do
        print("Index " .. index .. " is " .. value)
    end
end

local playerInventory = {"Health Pack", "Stamina Booster", "Cell Key"}
printArray(playerInventory)

table.remove(playerInventory, 2) -- Will remove the second item

printArray(playerInventory)
```

In Figure 9.8, you can see that originally index 2 is `Stamina Booster`, but once the value is removed, index 2 becomes `Cell Key`.

```
Output
2 Types    All Contexts    Filter...
18:30:25.290
18:30:25.290  Index 1 is Health Pack
18:30:25.290  Index 2 is Stamina Booster
18:30:25.290  Index 3 is Cell Key
18:30:25.291  Updated array
18:30:25.291  Index 1 is Health Pack
18:30:25.291  Index 2 is Cell Key
```

FIGURE 9.8
First, the original array prints. Then, the updated array without Stamina Booster prints.

Numeric `for` Loops and Arrays

Mentioned earlier was that one name of a loop using `ipairs()` is a generic `for` loop. The type of `for` loop you used in Hour 8 is called a numeric `for` loop. If it helps you remember which is which, remember that numeric `for` loops use numbers to control when to start and stop.

Numeric `for` loops can easily be used with arrays as well. Let's go through a couple of examples.

Finding and Removing All of a Value with a `for` Loop

While the previous code could only remove the first instance of a value found, this code snippet will find and remove all occurrences of a value from an array.

Remember, removing items causes later indexes to shift. Instead of starting at the beginning of the array, start at the end to avoid accidentally skipping values. By starting at the last index, you won't change the indexes of the values before it.

The size of the array can be found using `#arrayName` and used as the starting index number:

```
local playerInventory = {"Gold Coin", "Health Pack", "Stamina Booster", "Cell Key",
"Gold Coin", "Gold Coin"}

for index = #playerInventory, 1, -1 do
    if playerInventory[index] == "Gold Coin" then
        table.remove(playerInventory, index)
    end
end



print(playerInventory)
```

Searching Only a Section of an Array

Another time you probably want to use a numeric `for` loop is when you only want to go over part of an array. Say you need to find the names of the first three ships to cross the finish line in a space race:

```
local shipsRaced = {"A Bucket of Bolts", "Blue Moon", "Cats In Space",
"DarkAvenger12"}

local fastestThree = {}

for index = 1, 3 do
    table.insert(fastestThree, shipsRaced[index])
end

print(fastestThree)
```

The preceding code snippet takes the first three values of `shipsRaced` and adds them to `fastestThree`.

Summary

Tables, of which arrays are one type, let you organize your experience. With arrays, you can make a list of every player in your game and give each of them a new avatar item or weapon. You can also use arrays to create a list of every item in a folder that needs to have changes made to it.

Once you have all the items you want in an array, you can use a `for` loop to iterate over the array for whatever purpose you would like. You could print the names in the list, you could update the color of every object in the array, or execute much more complicated code. There are two types of `for` loops that can be used with an array. The `for` loop you used in the last hour is called a numeric `for` loop. It's good at times when you want to make changes to only a portion of the array or are working with very large arrays. The second type of `for` loop is called a generic `for`. In this case, `ipairs()` is used to go over the complete array, in order.

Q&A

Q. What are some other reasons for using a numeric `for` loop?

A. Over a very long lists of objects, numeric `for` loops run slightly faster. If you need to iterate through hundreds and hundreds of parts, it's worth keeping in mind.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. Arrays are a type of _______.
2. The number assigned to an item on an array is an _____ number.
3. In Lua, the index starts at the number ___.
4. `GetChildren()` returns an ___.
5. Is `ipairs()` used to create a generic loop or a numeric loop?

Answers

1. Table
2. Index
3. One. Other coding languages may start at 0 instead.
4. Array
5. Generic

Exercises

One way developers make their experiences feel more tied to the real world is by updating assets as seasons go by. In this first exercise, see if you can figure out how to make the pine tree in Figure 9.9 go from a summery green to a wintery white.

FIGURE 9.9
Updating your world to match the seasons makes your world feel more alive.

Tips

- This pine tree can easily be found in the ToolBox (see Figure 9.10). Search *tree* if you don't see it. We picked it for this exercise because it's a known good model made out of several parts, so you don't have to worry about swapping textures or it having extra scripts inside.

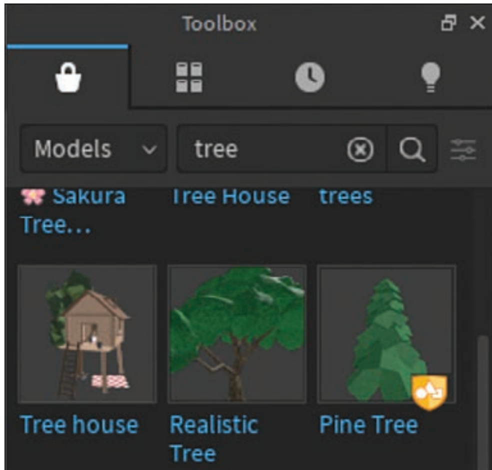


FIGURE 9.10
A pine tree with the endorsed model logo. The model is made out of base parts, so you'll easily be able to update the color of each leaf.

- Assume there's a whole forest of trees in your game.
- You're going to need more than one loop.

You can find the code solution in the appendix at the end of the book.

HOUR 10
Working with Dictionaries

What You'll Learn in This Hour:

- How to create a dictionary
- Run through dictionaries with `pairs()`
- How to return values from tables
- Test code designed for multiple participants
- Create a voting simulator

The second type of table you'll learn about in this book is *dictionaries.* This type of table enables you to gather information into groups and tag individual entries with something other than just a number, which opens up a whole world of possibilities. This hour covers creating dictionaries, adding and removing values, and iterating through dictionaries using `pairs()`.

One way you'll use dictionaries this hour is to keep track of who has the most votes in a voting simulator. The person with the most votes will be kicked off the island. This will give you practice using both arrays and dictionaries to keep track of participants, and what their votes are.

Intro to Dictionaries

Dictionaries are table objects that use a key to identify values instead of numbered indexes. The key can be a person's ID number, properties like Health or Stamina, or any valid data type. The following table is what a dictionary of player names and their respective scores might look like.

activePlayer Dictionary

Player's name as a key	Agatha	Billie	Mary Sue
Score as a value	1000	150	1200

In dictionary form, the same table might look like this:

```
local activePlayer = {
    Agatha = 1000,
    Billie = 150,
    ["Mary Sue"]  = 1200,
}
```

Use dictionaries when you need to label values, not just list them in a specific order as an array does.

Coding a Dictionary

Like arrays, dictionaries are created with curly brackets ({ }).

When you're constructing a new dictionary, you'll often see the brackets separated so that people can tell it apart from arrays, as shown in the following snippet:

```
local newDictionary = {
}
```

Key-value pairs are stored on separate lines followed by a comma. Keys and values can be any data type, including strings, numbers, instances, and other tables. The following dictionary uses strings as keys:

```
local inventory = {
   Batteries = 4,
   ["Ammo Packs"] = 1,
   ["Emergency Rations"] = 0,
}
```

Formatting Keys

How a key is formatted depends on if it is a string, instance, or something else. If strings are used as the key, they don't need to be in brackets unless there are spaces in the string. Then they must be enclosed in quotation marks *and* brackets:

```
local seedInventory = {
    -- String keys with no spaces
    Wheat = 1,
    Rice = 4,
    -- String key with spaces
    ["Sweet Potatoes"] = 3,
}
```

However, if the keys are an instance such as a part or someone in the game, then brackets should be used to mark that. In the following example, a dictionary uses boolean values to track whether all of the required portalStones are activated before opening the master portal:

```
local eastStone = workspace.EastStone
local westStone = workspace.WestStone
local northStone = workspace.NorthStone
local southStone = workspace.SouthStone

-- Each portal stone is an instance of a part, so it's marked in brackets
local requiredPortalStones = {
    [eastStone] = true,
    [westStone] = true,
    [northStone] = true,
    [southStone] = false,
}
```

Dictionaries are often used for organizing information for a character or object where they're used to label properties like name or level. In this case, neither the brackets nor the quotation marks are needed.

The following example uses a dictionary to track a character's name and level:

```
local hero = {
    Name = "Maria",
    Level = 1000,
}
```

WARNING

Don't Mix Keys and Indexes

Once you create a table, be consistent with using either key-value pairs or indexed values. Never use both within the same table. Combining keys and indexes in the same table can lead to errors.

Using Dictionary Values

To use individual dictionary values in code, type the name of the dictionary followed by the key in brackets, just like you did with arrays—for example, `dictionaryName[key]`. Or, if you're working with strings, you can also use dot notation:

```
local hero = {
    Name = "Maria",
    Level = 1000,
}
-- Remember that Name is a string and can be accessed with brackets
print ( "The hero's name is "  .. hero["Name"] )
```

```
-- Or you can use dot notation
print ( "The hero's name is "  .. hero.Name )
```

TIP

Dot Notation Only Works with String Keys

Once again, dot notation only works with strings, but it's something you'll see quite a bit.

Use Unique Keys

Lua won't stop you from trying to reuse the same key. Keep this in mind as you code. In the following example, the original value for the key `Name` will be overwritten, and the second value given for the key `Name` will be printed:

```
local hero = {
    Name = "Maria",
    Level = 1000,
    Name = "Aya",
}
-- Will print Aya. The first value has been overwritten.
print ( "The hero's name is "  .. hero.Name)
```

Adding and Removing from Dictionaries

To add a key-value pair to an existing dictionary, the formula is:

```
dictionaryName[key] = value
```

Or if working with strings:

```
dictionaryName.String = value
```

Adding players to a dictionary when they join the game, and then starting them off with 0 points, might look like: `playerPoints.Points = 0`

Be careful! As mentioned earlier, if the key already exists, the existing value will be overwritten.

Removing Key-Value Pairs

To remove a key-value pair from a dictionary, set the key's value to nil. This deletes the key:

```
local lightBulb = model.SpotLight

local flashLight = {
```

```
    Brightness = 6,
    [lightBulb] = "Enabled",
}

-- Remove string keys
flashLight.Brightness = nil

-- Remove other keys
flashLight[lightBulb] = nil
```

This also means if you are ever trying to get a value from a dictionary, and you only get `nil`, that means you're looking for something that doesn't exist.

TRY IT YOURSELF ▼

Add New Players to a Dictionary

In this Try It Yourself, you add a player's name to a dictionary when they join and then assign them to a team. If you're using a key-value pair that hasn't been added previously, it'll be added automatically.

1. In ServerScriptService, create a new script.
2. Get the Players service and create an empty dictionary:

```
Players = game:GetService("Players")
-- Empty dictionary
local teams= {
}
```

3. Add a new function for assigning teams and include a parameter for a new player. Connect the function to the `Players.PlayerAdded` event:

```
Players = game:GetService("Players")
local teams= {
}
-- Assign player to "Red" team
local function assignTeam(newPlayer)
end
Players.PlayerAdded:Connect(assignTeam)
```

4. In the function, add a variable to get the player's name:

```
-- Assign player to "Red" team
local function assignTeam(newPlayer)
    local name = newPlayer.Name
end
Players.PlayerAdded:Connect(assignTeam)
```

5. Insert the name into the teamAssignments dictionary as a key, and set the value to "Red":

```
-- Assign player to "Red" team
local function assignTeam(newPlayer)
    local name = newPlayer.Name
    teams.name = "Red"
end
Players.PlayerAdded:Connect(assignTeam)
```

6. Use `name` to print the name of the player and `teamAssignment[name]` to print the value of the key:

```
Players = game:GetService("Players")
local teams = {
}
-- Assign player to "Red" team
local function assignTeam(newPlayer)
    local name = newPlayer.Name
    teams.name = "Red"
    print( name .. " is on " .. teams.name .. " team.")
end
Players.PlayerAdded:Connect(assignTeam)
```

Working with Dictionaries and Pairs

`pairs()` can be used to work with a dictionary element's key, value, or both. In the following `for` loop, the first variable is the key. The second variable is the value. The dictionary that you want to work with is passed into `pairs()`:

```
local inventory = {
    ["Gold Bricks"] = 43,
    Carrots = 3,
    Torches  = 2,
}

print("You have:")
for itemName, itemValue in pairs(inventory) do
    print(itemValue, itemName)
end
```

TIP

Comma Instead of Dots

If just printing two variables, you can use a comma instead of two dots.

Returning Values from Tables

You can search a table using `pairs()` or `ipairs()` for half of any table element, such as the key or value, to find and return the other half. The following code snippet searches a dictionary of names to find the spy among them:

```
local friendOrSpy = {
    Angel = "Friend",
    Beth = "Spy",
    Cai = "Friend",
    Danny = "Friend",
}
-- Searches a given dictionary to find the spy
local function findTheSpy(dictionaryName)
    for name, loyalty in pairs(dictionaryName) do
        if loyalty == "Spy" then
            return name
        end
    end
end

local spyName = findTheSpy(friendOrSpy)

print("The spy is " .. spyName)
```

TRY IT YOURSELF ▼

Vote Them Off the Island!

In this Try It Yourself, you're going to pretend to vote somebody off an island. The end goal of this exercise is to take the name of every player in the experience and then create a way everyone can vote on who should be kicked off the imaginary island.

To start, begin mentally breaking down the problems you need to solve to create this script. As you start working on longer scripts, a to-do list of what needs to be done can be helpful.

Here are some problems to solve for:

- There needs to be enough time for all of the players to join before voting.
- Each player's name needs to be represented in some way that players can interact with.
- The votes for each person need to be kept track of.
- The results need to be shown at the end of voting.

There are other things you could possibly solve for, but this is enough of a list to work with for now.

Set Up

The first problem will be solved by allowing players to click a button when they're ready to begin voting. In a more complex experience, the voting might happen after a series of mini-games or something like that. To solve the second problem, once the voting starts, a new set of buttons representing each player will appear. (See Figure 10.1.)

FIGURE 10.1
One button will start the vote, and then more buttons will be created to represent everyone on the island.

For this example, the names of the ProximityPrompts are important. Differently named prompts will be used to do different things:

1. Set up a part to act as the first button that starts the voting:
   a. Insert a ProximityPrompt named `StartVote`.
   b. Set HoldDuration to 1.
2. Set up a second part to act as the button that will hold the player's name:
   a. Insert a ProximityPrompt named `AddVote`.
   b. Set HoldDuration to 0.5.
   c. Once set up, move the button to ServerStorage where copies can be made. (See Figure 10.2.) Do not move the StartVote button; that needs to be where people can see it.

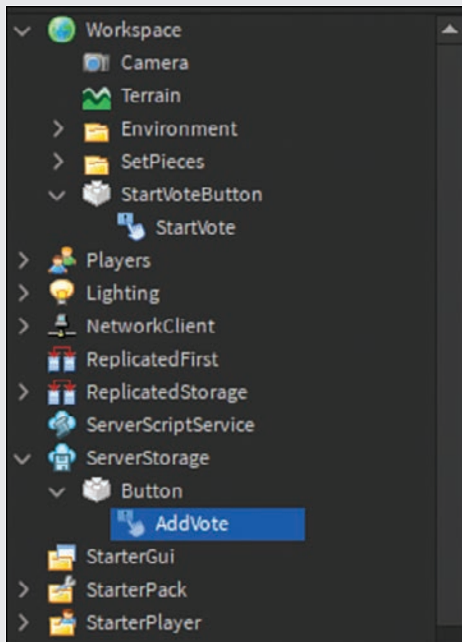


FIGURE 10.2
The button to start the vote stays in Workspace, whereas the button with AddVote goes into ServerStorage.

Coding the Script

As you work through this exercise, you use multiple tables, both arrays and dictionaries. Newcomers to the experience will be added to an array named activePlayers. Once voting starts, anyone who receives a vote will be added to a dictionary along with how many votes they have.

Set Up the Buttons

Remember the different problems mentioned earlier that need to be solved for? As you start working on larger scripts, it's better to break the script into sections with individual functions designed to solve unique problems. You start out by getting the names of all the players and creating buttons for each player:

1. In ServerScriptService, add a new script.

2. Create variables for the following:

   a. ServerStorage

   b. ProximityPromptService

   c. Players service

   d. Amount of time players have to cast their votes

   e. An array to hold all of the active players

   f. A dictionary to hold the votes cast

```
local ServerStorage = game:GetService("ServerStorage")
local ProximityPromptService = game:GetService("ProximityPromptService")
local PlayersService = game:GetService("Players")

local VOTING_DURATION = 30

local activePlayers = {}
local votes = {

}
```

3. At this point, you start breaking your code into smaller solutions. Create a new function that adds players to the `activePlayers` array when they are added to the experience. Use the `PlayerAdded` event to call the function:

```
local function onPlayerAdded(player)
    table.insert(activePlayers, player)
end

PlayersService.PlayerAdded:Connect(onPlayerAdded)
```

4. Create a new function that creates a button for each player instance in the `active-Players` array. This function is called later in the script with the `StartVote` prompt:

```
local function onPlayerAdded(player)
    table.insert(activePlayers, player)
end

local function makeButtons()
    for index, player in pairs(activePlayers) do
        -- Use the name of your button in the next line
        local newBooth = ServerStorage.Button:Clone()

        newBooth.Parent = workspace
    end
end

PlayersService.PlayerAdded:Connect(onPlayerAdded)
```

5. Find the ProximityPrompt within the button, and set `ActionText` to match the player's name:

```
local function makeButtons()
    for index, player in pairs(activePlayers) do
        local newBooth = ServerStorage.VotingBooth:Clone()

        local proximityPrompt =
            newBooth:FindFirstChildWhichIsA("ProximityPrompt")
        local playerName = player.Name
        proximityPrompt.ActionText = playerName

        newBooth.Parent = workspace
    end
end
```

6. Add the highlighted code additions to space the buttons apart a little. You learn more about positioning objects in Hour 14, "Coding in 3D World Space":

```
local function makeButtons()
    local position = Vector3.new(0,1,0)
    local DISTANCE_APART = Vector3.new(0,0,5)

    for index, player in pairs(activePlayers) do
        local newBooth = ServerStorage.Button:Clone()

        local proximityPrompt =
            newBooth:FindFirstChildWhichIsA("ProximityPrompt")
        local playerName = player.Name
        proximityPrompt.ActionText = playerName

        position = position + DISTANCE_APART
        newBooth.Position = position

        newBooth.Parent = workspace
    end
end
```

7. Add a third function connected to the `PromptTriggered` event. Inside, use the StartVote proximity prompt to call `makeButtons()`:

```
local function makeButtons()
    -- Earlier code
end

local function onPromptTriggered(prompt, player)
    if prompt.Name == "StartVote" then
        makeButtons()
```

```
        end
    end

    PlayersService.PlayerAdded:Connect(onPlayerAdded)
    ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)
```

TIP

Keep Event Connections Together When Possible

Notice that all of the event connections are at the bottom of the script. This keeps everything organized.

Testing for Multiple People

When testing code that's meant to be used for multiple players, you need to use Network Simulator instead of just Play or Play Here. Network Simulator enables you to set up as many fake people as you want, which you can then control to test your game:

1. In the Test tab, find the section titled Clients and Servers.
2. Set the bottom drop-down menu to two or more players, as shown in Figure 10.3.

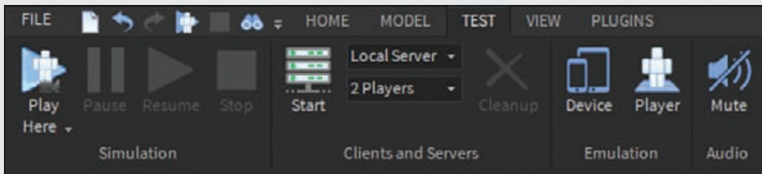


FIGURE 10.3
Use the drop-down menu to select at least two players.

3. Clicking Start will bring up a new Studio instance representing the server and an additional window for each pretend player. Player windows have a blue outline (see Figure 10.4), whereas the Server window has a green outline.
4. Click any of the blue windows to control that dummy character. While testing, any errors and printed messages show up in the Server Output window.
5. Interact with StartVote and make sure that a button is spawned for each test player.

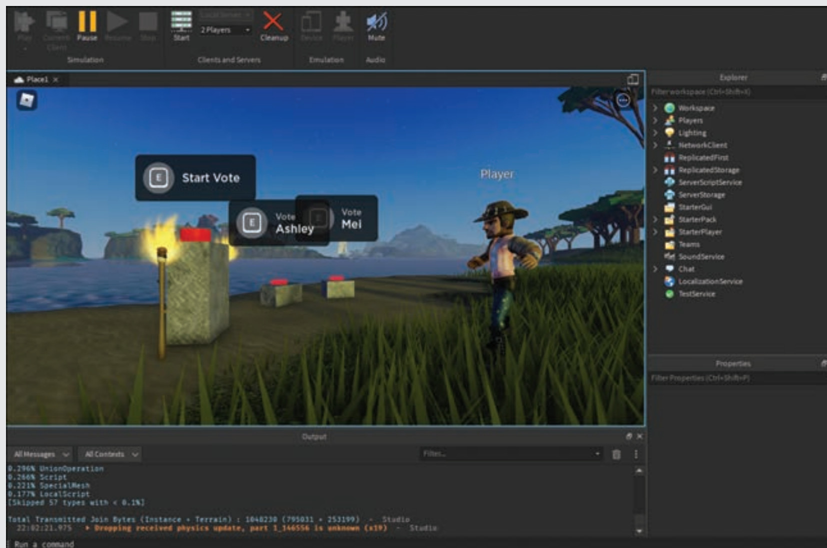


FIGURE 10.4
The blue outline indicates this is one of the player instances.

TIP

Positioning Objects

The buttons should appear just slightly above the center of your experience, 0,0,0. Later in the book, you'll learn how to control the position of objects when added to the workspace.

6. To stop the test, click Cleanup. (See Figure 10.5.)

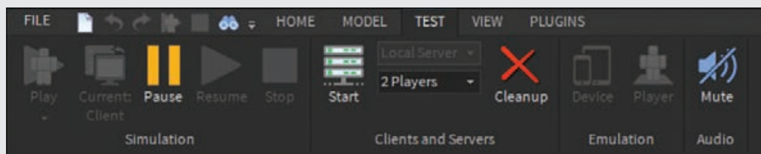


FIGURE 10.5
Click the red X to close the extra Studio instances.

Adding and Counting Votes

Once the voting buttons are up and running, the votes need to be tracked and the results shown when the voting is done. In this example, you give a set amount of time that players are allowed to vote and then show the results:

1. In the same script, above `onPromptTriggered()`, create a new function called `showVotes` that prints all the values in the votes dictionary:

```
local function showVotes()
    for playerName, value in pairs(votes) do
        print(playerName .. " has " .. value .. " votes.")
    end
end
```

2. In `onPromptTriggered()`, begin a countdown once voting starts, and call `showVotes` when it's done:

```
local function onPromptTriggered(prompt, player)
    if prompt.Name == "StartVote" then
        makeButtons()

        for countdown = VOTING_DURATION, 0, -1 do
            print(countdown .. " seconds left")
            wait(1.0)
        end

        showVotes()
    end
end
```

TIP

Further Code Organization

If you wanted to, you could make the countdown its own function as well. That would allow it to be called by other means than just a prompt.

3. Also in `onPromptTriggered()`, add a second condition that listens for Proximity Prompts named `AddVote`:

```
local function onPromptTriggered(prompt, player)
    if prompt.Name == "StartVote" then
        makeButtons()
        -- Countdown code
        showVotes()
    elseif prompt.Name == "AddVote" then

    end
end
```

4. Get the name of the player who was voted for from `ActionText`, where it was used to label the button:

```
local function onPromptTriggered(prompt, player)
if prompt.Name == "StartVote" then
        makeButtons()
        -- Countdown code
        showVotes()
    elseif prompt.Name == "AddVote" then
        local chosenPlayer = prompt.ActionText
    end
end
```

5. If the votes dictionary doesn't already have an entry by that name, add the player's name as a key and set their points to 1. If a key does exist, take the current value and add one to it:

```
local function onPromptTriggered(prompt, player)
    if prompt.Name == "StartVote" then
        makeButtons()

        for countdown = VOTING_DURATION, 0, -1 do
            print(countdown .. " seconds left")
            wait(1.0)
        end

        showVotes()

     elseif prompt.Name == "AddVote" then
        local chosenPlayer = prompt.ActionText
        print("A vote for " .. chosenPlayer)

        if not votes[chosenPlayer] then
            votes[chosenPlayer] = 1
        else
            votes[chosenPlayer] = votes[chosenPlayer] + 1
        end

    -- Optional check for debugging purposes
    else
        print("Prompt not found")
    end
end
```

TIP

If All Else Fails

Included are a couple of print statements and an else that can be used for testing the code. A final else that runs if no other condition proves true can be quite helpful for making sure that the function was called as expected.

6. Use Network Simulator with at least two players to test the code, and look for the results in the Server Output.

Summary

In all Roblox experiences, tables are behind the scenes tracking information. Arrays are used to create lists of objects, and the information stored will always be in order. While dictionaries are used to track information about objects and properties, and unlike arrays, the entries within aren't guaranteed to stay in any particular order.

To iterate through a dictionary, you want to use `pairs()` instead of `ipairs()`. The two functions are very similar, but `ipairs()` only works with arrays.

Q&A

Q. I've seen `pairs()` used with arrays, so why not just use `pairs()` with both arrays and dictionaries?

A. One of the benefits of using arrays is that it stores things in order. `pairs()` is not guaranteed to return every object in order, whereas `ipairs()` is.

Q. If `pairs()` can technically work with arrays, why can't `pairs()` work with dictionaries?

A. `ipairs()` requires an ordered index to work. Dictionaries don't have that. On the other hand, pairs accept any valid datatype as a key, including indexes.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. Instead of using indexes, dictionaries use ___.
2. True or false: Dictionaries store information in a particular order.

3. To iterate through a dictionary, use the ____ functions.
4. If an instance is being used as a key, does it need to have brackets or quotation marks?
5. To remove a key-value pair from a dictionary, set the value to ___.
6. Why does `showVotes()` have to be above `onPromptTriggered()`?

Answers

1. Keys
2. False. Although dictionaries might sometimes return values in the order in which they were stored, it's by no means guaranteed.
3. `pairs()`
4. If an instance is being used as a key, it only needs the brackets.
5. Nil
6. Because code is read from top to bottom, `showVotes()` needs to be created before it's called in `onPromptTriggered()`.

Exercise

Earlier in this hour, a person was assigned to the `"Red"` team upon joining the experience. For this exercise, can you figure out how to alternate team assignments between `"Red"` and `"Blue"`? Print the members of each team.

Tips

- Test using Network Simulator.
- Instances can't concatenate with strings, but the name of the instance will.

This page intentionally left blank

HOUR 11
Client Versus Server

What You'll Learn in This Hour:

- ▶ What the server/client divide is
- ▶ How to set up serverwide messages
- ▶ How to create player-specific messages with GUIs
- ▶ How to test code
- ▶ How to use RemoteFunctions for two-way communication between the server and client

There are two sides to every Roblox experience. One side is where people interact with the experience, and the other side is in the cloud, controlling everything. This hour covers how these two sides work together and how messages are sent between them. At the end of the hour, you create a shop where players can click a button to buy firewood for the resource game created in Hour 9.

Understanding the Client and the Server

The first side, the side where people and players are interacting with the world, is the *client* side. A client is the individual device somebody uses to join a game, whether it's a Mac, PC, phone, tablet, or even a VR console.

Some things about the experience are calculated on the individual client device, whereas other things are taken care of by super powerful Roblox hardware called the *server*. The server and the client are always talking to each other. The server tells the client what the overall world is like, and the client tells the server what a person is doing within the world.

Typically, you want important information like scores, in-game money, and progress levels to be handled by the server. The server is more secure than the client and is harder to hack into. Meanwhile, the client handles things that apply only to the particular person using the device or for when it's important to have the least amount of lag possible, such as for showing them their own score or when controlling the camera.

Working with GUIs

So far, we've only worked with server-side code typed within Script objects, and everyone in the world sees the same thing. The next step is to start creating code that shows the person on each client information that only they can see, like their current score, quest progress, health level, and how much money they have. Information like this is displayed in what's called a Graphical User Interface or GUI, like the one shown on the left in Figure 11.1.

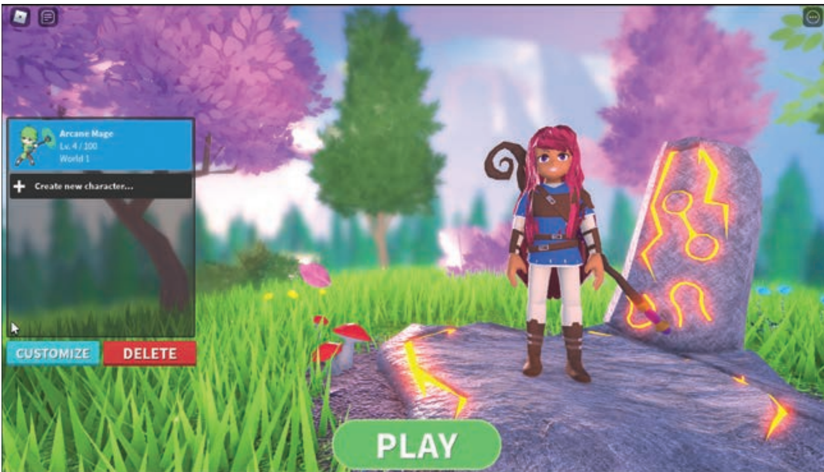


FIGURE 11.1
In World//Zero by Red Manta Studio, returning players are greeted by GUI showing their characters' levels and current locations. Additional GUI elements allow for customization and deletion, and a big green button starts the game.

With GUIs, you can also create onscreen buttons that allow you to build out things like shops.

The majority of GUI items that can only be seen by the local client should be placed in Starter-GUI, and you type the code into a LocalScript object instead of a Script object. Anything in StarterGUI is duplicated to anyone who joins the experience.

▼ TRY IT YOURSELF

Create a GUI with the Player's Name

To show you what it can look like when everyone in the server sees information custom tailored just for them, in this Try It Yourself, you create a GUI with the player's name.

Set Up

1. In Explorer, select StarterGUI.
2. Insert a new ScreenGui (see Figure 11.2). This will be the container for any buttons and labels you want to create.

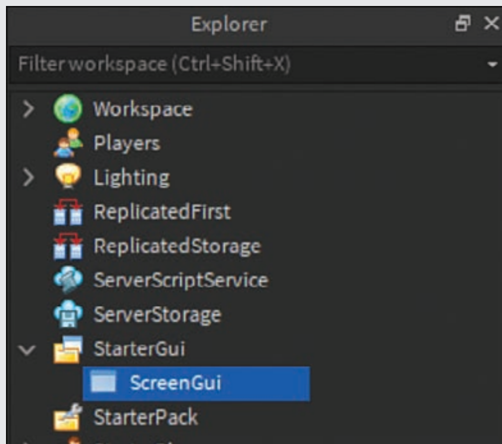


FIGURE 11.2
Insert a ScreenGui object into StarterGui.

3. Inside of ScreenGui, add a TextLabel. Rename the TextLabel **`PlayerName`**, as shown in Figure 11.3.

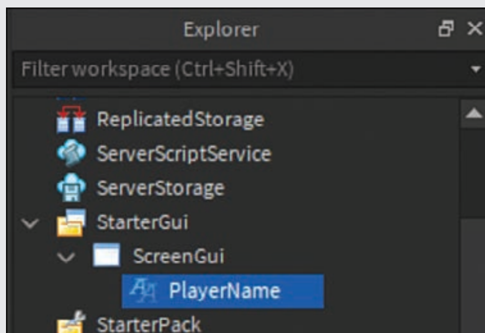


FIGURE 11.3
Insert a TextLabel into the ScreenGui just created.

TIP

Customizing GUIs

To learn about customizing the appearance and placements of ScreenGuis, check out the companion book *Roblox Game Development in 24 Hours* or look up Intro to ScreenGuis on the Developer Hub.

Script

You use a LocalScript instead of the normal Script object. The Script object is for server-side code:

1. With the ScreenGui selected, insert a new LocalScript object (see Figure 11.4).

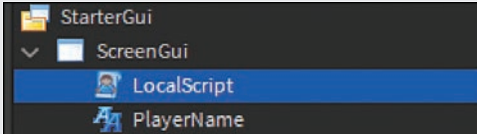


FIGURE 11.4
LocalScripts are for client-side code.

TIP

GUI Script Placement

GUI scripts must be inside of StarterGUI. ServerScriptService can only access server Script objects.

2. Within the LocalScript, create variables for the Players service and the ScreenGui.
3. Create a new variable for the TextLabel:

```
local Players = game:GetService("Players")

local screenGui = script.Parent
local textLabel = screenGui.PlayerName
```

4. Get the local player. In LocalScripts, this can easily be done with `Players.LocalPlayer`:

```
local Players = game:GetService("Players")

local screenGui = script.Parent
local textLabel = screenGui.PlayerName
local localPlayer = Players.LocalPlayer
```

5. Set TextLabel's Text property to the name of the local player:

```
local Players = game:GetService("Players")

local screenGui = script.Parent
local textLabel = screenGui.PlayerName
local localPlayer = Players.LocalPlayer

textLabel.Text = localPlayer.Name
```

6. Use the Network Simulator to test your code. You'll see that each person's name is displayed on screen.

Understanding RemoteFunctions

One thing to keep in mind is that the server and the client don't have access to the same information. There are certain folders that the client can't access, and vice versa. Here are a few examples:

Object	Server	Client
Workspace	yes	yes
ServerScriptService	yes	no
ServerStorage	yes	no
ReplicatedStorage	yes	yes

Also, the server and the client don't share information. Some people call this the server/client divide, but you can just imagine it as if there was a wall between the two environments keeping them separate.

To get information from one side to the other, special objects are used to toss information over the wall. This can be done through RemoteEvent and RemoteFunction objects that both Scripts and LocalScripts can use to communicate with each other. In this hour, RemoteFunctions are covered, and the next hour gets into the different types of RemoteEvents.

Using RemoteFunctions

As stated earlier, RemoteFunctions are designed to send a request across the server-client boundary.

What makes RemoteFunctions special is that they can also wait for a response from the other side acting as a messenger between the client and the server. Usually this is a request from the local client for the server to do something, and then the server sends the results back.

RemoteFunctions must be created where both clients and the server can access it—for instance, ReplicatedStorage (see Figure 11.5).

Meanwhile, you have a normal server Script in ServerScriptService and a LocalScript in StarterPlayerScripts, as shown in Figure 11.6.

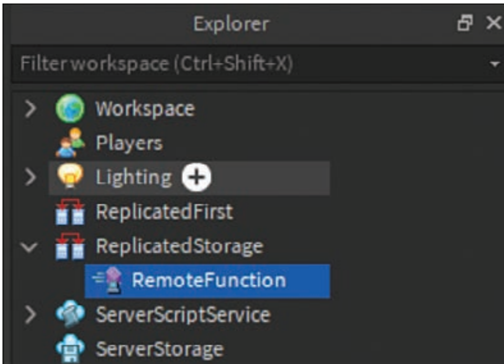


FIGURE 11.5
RemoteFunctions must be placed someplace like ReplicatedStorage, which both the client and server can access.

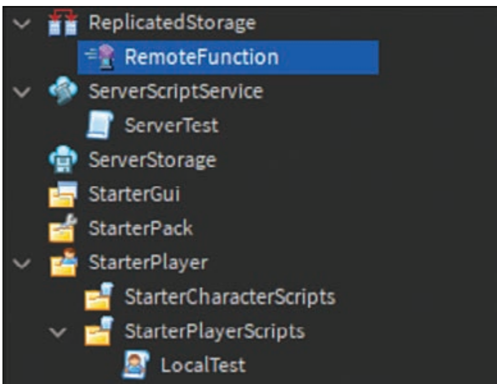


FIGURE 11.6
LocalScript in StarterPlayerScripts and server Script in ServerScriptService.

Get a message from the server, and print it locally: On the server side, set up a function that returns a simple string to print. Bind the function to the RemoteFunction object, as highlighted here:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local remoteFunction = ReplicatedStorage:WaitForChild("RemoteFunction")

local function sayHello()
    local serverMessage = "Hello from the server"
    return serverMessage
end

remoteFunction.OnServerInvoke = sayHello
```

RemoteFunctions can only have one function bound to them at a time. On the local side, the code to invoke (indirectly call) the server would look like this:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local remoteFunction = ReplicatedStorage:WaitForChild("RemoteFunction")

local messageFromServer = remoteFunction:InvokeServer()

print(messageFromServer)
```

Server to Client

It is possible to go the other direction—from server to client to server. However, it's quite risky and won't be covered within this book for the following reasons:

- If the client throws an error, the server will throw the error, too.
- If the client disconnects while it's being invoked, the `InvokeClient()` call will error.
- If the client never returns a value, the server will hang forever.

TRY IT YOURSELF ▼

Make a Store

A good example of when you might need to double-check with the server and wait for a response is if someone wants to buy something. A client clicks a button to buy something, and then the server checks whether the client actually has enough money and confirms the purchase.

For the purposes of this Try It Yourself, you take the leaderboard system you've worked with before and modify it to allow players to spend gold to buy more logs to burn for the fires (see Figure 11.7).

FIGURE 11.7
The end result will allow people to buy logs for the fire.

Set Up

For the sake of speed, use the leaderboard system for fuel and fire you previously set up. If you don't have it, you can use the code in the Hour 11 section of the appendix to quickly set it up.

1. In ServerScriptService, PlayerStats, give people a starting gold amount of 10 to make testing easier:

```
local gold = Instance.new("IntValue")
gold.Name = "Gold"
gold.Value = 10
gold.Parent = leaderstats
```

2. In ReplicatedStorage, add a new RemoteFunction instance named **`CheckPurchase`** (see Figure 11.8).

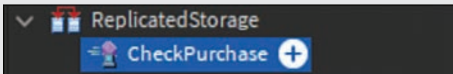


FIGURE 11.8
Add a RemoteFunction named CheckPurchase.

3. In ServerStorage, add a new folder named **`ShopItems`** (see Figure 11.9).

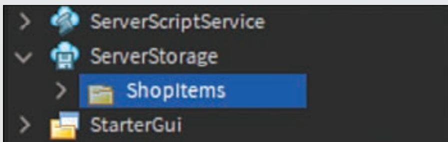


FIGURE 11.9
Add a folder named ShopItems.

4. In ShopItems, add a Folder object named **`3Logs`** and add the three attributes shown on the right in Figure 11.10. You'll use these names and values in the script.

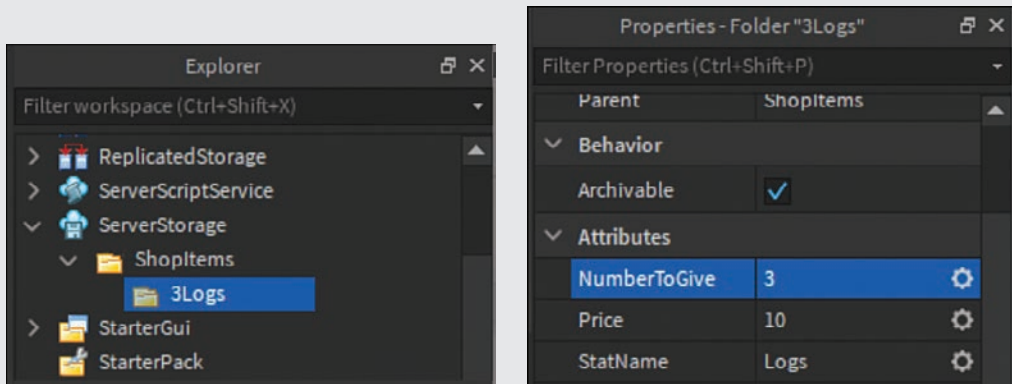


FIGURE 11.10
A new folder with custom attributes for NumberToGive, Price, and StatName.

TIP

Future Proofing the Shop

In a more advanced shop, the folder can also be used to hold mesh models, image icons, and more.

5. In StarterGUI, add
   - A new ScreenGui named **`ShopGui`**.
   - In ShopGui, add a new TextButton named **`Buy3Logs`** (see Figure 11.11).

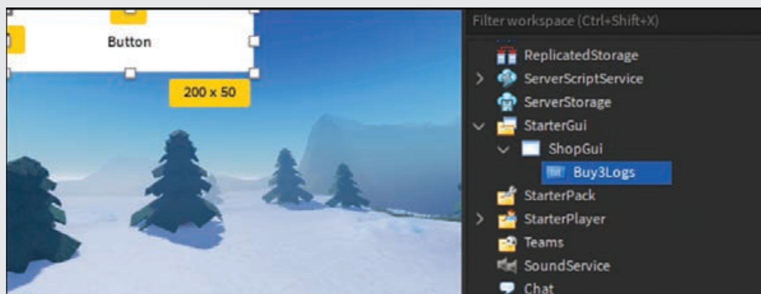


FIGURE 11.11
Set up for the GUIs.

TIP

Moving GUIs

To move the GUIs around, you can select the GUI objects in Explorer and then move and scale them.

6. Select Buy3Logs and add an attribute, as shown in Figure 11.12:
   - **Name:** PurchaseType
   - **Value:** 3Logs
   - **Type:** String

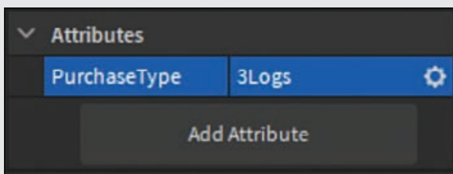


FIGURE 11.12
An attribute for Buy3Logs.

LocalScript

LocalScripts for GUI buttons need to be a direct child of the button they affect. In the LocalScript, you set up the code to invoke the server and tell the person if their purchase was successful or if they need more gold:

1. In the Buy3Logs button, add a LocalScript.
2. Get the information you need for the RemoteFunction, CheckPurchase:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local checkPurchase= ReplicatedStorage:WaitForChild("CheckPurchase")
```

3. For the button, you need to get the `PurchaseType` attribute:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local checkPurchase= ReplicatedStorage:WaitForChild("CheckPurchase")

local button = script.Parent
local purchaseType = button:GetAttribute("PurchaseType")
```

4. Use the `PurchaseType` to create and assign default text for the button and then set up a cooldown for how long the button will be deactivated between purchases:

```
local defaultText = "Buy " .. purchaseType
button.Text = defaultText

local COOLDOWN = 2.0
```

TIP

Make Sure to Assign Default Property Values

You'll be changing the Text property several times, so you want to make sure you have a default message assigned in the beginning of the script.

5. Create a new function to be called when the button is activated:

```
local function onButtonActivated()


end

button.Activated:Connect(onButtonActivated)
```

6. Create a variable to invoke the server to send the `purchaseType` and hold the returned purchase confirmation:

```
local function onButtonActivated()
    local confirmationText = checkPurchase:InvokeServer(purchaseType)
end
```

7. Display the confirmation text while disabling the button and then return the button to normal:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local checkPurchase = ReplicatedStorage:WaitForChild("CheckPurchase")

local button = script.Parent
local purchaseType = button:GetAttribute("PurchaseType")
local defaultText = "Buy ".. purchaseType
button.Text = defaultText

local COOLDOWN = 2.0

local function onButtonActivated()

    local confirmationText = checkPurchase:InvokeServer(purchaseType)
    button.Text = confirmationText
    button.Selectable = false
    wait(COOLDOWN)
    button.Text = defaultText
    button.Selectable = true
end

button.Activated:Connect(onButtonActivated)
```

Server Script

The server side is where you want to do all the heavy lifting of checking and updating stats. Once the client sends over what the user wants to purchase, the server checks whether the user has enough gold. If they do, the purchase will be made, and the button text will say `Purchase Successful!` If they don't have enough gold, then the server will send back a message saying `Not enough gold.`

1. In ServerScriptService, add a new script.
2. Think about what your script needs to do and make the references you think it will need. Compare your work to the following snippet:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local Players = game:GetService("Players")
local ServerStorage = game:GetService("ServerStorage")

local checkPurchase = ReplicatedStorage:WaitForChild("CheckPurchase")
local shopItems = ServerStorage.ShopItems
```

3. Create a function named **`confirmPurchase`** with parameters to pass in `player` and `purchaseType`. Bind `confirmPurchase` to the RemoteFunction:

```
local function confirmPurchase(player, purchaseType)

end

checkPurchase.OnServerInvoke = confirmPurchase
```

4. Inside of `confirmPurchase`, get how much gold the person has:

```
local function confirmPurchase(player, purchaseType)
    local leaderstats = player.leaderstats
    local currentGold = leaderstats:FindFirstChild("Gold")

end
```

5. Use the passed in `purchaseType` to find the item they want to buy. Get the resource stat that will be updated on the leaderboard, the item's price, and how many of the resource will be received:

```
local function confirmPurchase(player, purchaseType)
    local leaderstats = player.leaderstats
    local currentGold = leaderstats:FindFirstChild("Gold")

    local purchaseType = shopItems:FindFirstChild(purchaseType)
    local resourceStat =
        leaderstats:FindFirstChild(purchaseType:GetAttribute("StatName"))
    local price = purchaseType:GetAttribute("Price")
    local numberToGive = purchaseType:GetAttribute("NumberToGive")
end
```

TIP

Check Your Work

You should have four variables here. Notice how `purchaseType:GetAttribute("StatName")` is passed into `shopItems:FindFirstChild()`.

6. Set up a variable for the server message that will be sent back once everything is checked:

```
local function confirmPurchase(player, purchaseType)
    local leaderstats = player.leaderstats
    local currentGold = leaderstats:FindFirstChild("Gold")

    local purchaseType = shopItems:FindFirstChild(purchaseType)
    local resourceStat =
        leaderstats:FindFirstChild(purchaseType:GetAttribute("StatName"))
    local price = purchaseType:GetAttribute("Price")
    local numberToGive = purchaseType:GetAttribute("NumberToGive")

    local serverMessage = nil


    return serverMessage
end
```

TIP

Set Undetermined Values to nil

In this code, the value of `serverMessage` will be determined in the next step. Rather than just leaving the variable without a value for now, set it to nil so it's clear that a value was not mistakenly left out.

7. Set up conditionals to check how much gold the person has and whether they can buy the item. Depending on the results, send back an appropriate message to the client and update the leaderboard. Here's the completed script:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local Players = game:GetService("Players")
local ServerStorage = game:GetService("ServerStorage")

local checkPurchase = ReplicatedStorage:WaitForChild("CheckPurchase")
local shopItems = ServerStorage.ShopItems

local function confirmPurchase(player, purchaseType)

    local leaderstats = player.leaderstats
    local currentGold = leaderstats:FindFirstChild("Gold")
```

```
        local purchaseType = shopItems:FindFirstChild(purchaseType)
        local resourceStat =
            leaderstats:FindFirstChild(purchaseType:GetAttribute("StatName"))
        local price = purchaseType:GetAttribute("Price")
        local numberToGive = purchaseType:GetAttribute("NumberToGive")

        local serverMessage = nil

        if currentGold.Value >= price then

            currentGold.Value = currentGold.Value - price
            resourceStat.Value += numberToGive

            serverMessage = ("Purchase Successful!")

        elseif currentGold.Value < price then
            serverMessage = ("Not enough Gold")

        else
            serverMessage = ("Didn't find necessary info")

        end

        return serverMessage
    end

    checkPurchase.OnServerInvoke = confirmPurchase
```

8. Test everything out! You can make the store prettier by adding images for the items and styling the look and font of the text and buttons.

Summary

Every Roblox experience has two sides that come together to make the world users see. The first side is the local client, which is the device such as a computer or phone where people are interacting with Roblox. The second side is the server, which is making sure that everyone is going through the experience in mostly the same way.

Generally, you want to make sure as much of the code that you create stays on the server side of things where it's more secure. The last thing you want is people taking advantage of the local client to make unauthorized purchases or updates to their stats.

Some things, however, are specific to a client. If one person opens a shop window or makes a purchase, you don't want everyone in the experience to have to look at the shop window until they're done. So, shops are an example of something you would want done locally.

Q&A

Q. Can anything other than a function be bound to a RemoteFunction?

A. RemoteFunctions expect a function; trying to bind something like a variable will cause an error.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. The device somebody uses to join a Roblox experience is the____.
2. The super powerful hardware running most of a Roblox experience is the ____.
3. Code that only makes changes to what a client sees is typed into a _____ object.
4. A RemoteFunction is
    - **A.** A type of event
    - **B.** An object
    - **C.** A data type
5. RemoteFunctions can be used for ____-way communication between the server and the client.
6. What does it mean to invoke something?

Answers

1. Local client
2. Server
3. LocalScript
4. An object
5. Two
6. Invoke means to call something indirectly.

Exercises

One problem with the current shop is that the price of the items aren't listed. However, remember that only one function can be bound to a RemoteFunction at a time. Create a second RemoteFunction and use it to display not only the name of items people can purchase, but also the cost. (See Figure 11.13.) Be sure to test your code with multiple items for sale.

FIGURE 11.13
Extend the shop code so that it can retrieve the price for each item in the shop.

HOUR 12

Remote Events: One-Way Communication

What You'll Learn in This Hour:

- ▶ How to use remote events
- ▶ How to send a message to all clients
- ▶ How to create player-specific messages
- ▶ How to send a message from the client to the server
- ▶ How to create a GUI countdown

Hour 11 covers the differences between the local client and the server living on Roblox hardware. It also covers one way of communicating across the divide. This hour covers a second way to send messages.

Remote Events: A One-Way Street

Sometimes you just need to send a message from the client to the server or vice versa without needing a response. If that's the case, instead of using a RemoteFunction, you should instead use a RemoteEvent.

As a reminder, a RemoteEvent is an object that you can insert instances of into Workspace, typically within ReplicatedStorage where both the client and server can access it (see Figure 12.1).

There are three major ways in which RemoteEvents can be used to send a signal:

- ▶ From the server to a specific client
- ▶ From the server to all of the joined clients
- ▶ From a client to the server

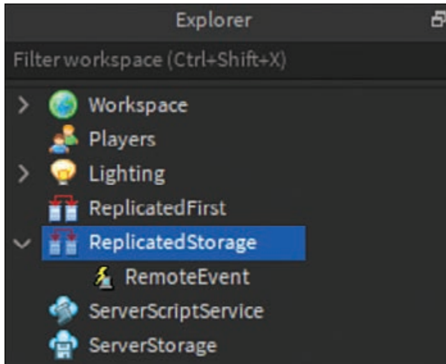


FIGURE 12.1
RemoteEvent should be in ReplicatedStorage to be accessed by both the client and server.

Communicating from the Server to All Clients

The basic formula for sending a message from the server to all clients is as follows:

```
remoteEventName:FireAllClients(variableName)
```

Information needing to be sent to the client is passed into `FireAllClients(informationHere)`.

On the client side, you set up one or more functions to call when the event gets fired:

```
local function firstFunction(incomingInfo)
    -- Do stuff
end

local function secondFunction(incomingInfo)
    -- Do different stuff
end

-- Connect both functions to onClientEvent
remoteEventName.OnClientEvent:Connect(firstFunction)
remoteEventName.OnClientEvent:Connect(secondFunction)
```

▼ TRY IT YOURSELF

Quick Countdown

Let's start off with something familiar to demonstrate: a countdown. This is a good example because the server needs to get information to everyone in the server, but it doesn't need any information back.

So far, we've demonstrated a countdown in two ways. The first was simply in Output where the client can't see it. The second was displayed on a 3D GUI in the game space. The problem with that is that people can walk away from it and not see it. If you want to be sure everyone in the experience sees the countdown, as shown in Figure 12.2, using a RemoteEvent is the way to do it.

FIGURE 12.2
A TextLabel displays a countdown until the next round.

1. In ReplicatedStorage, add a RemoteEvent named `CountdownEvent`. (See Figure 12.3.)

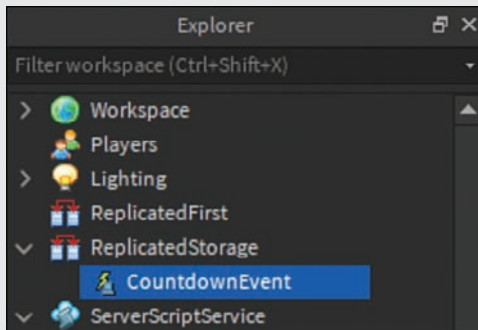


FIGURE 12.3
Insert a RemoteEvent into ReplicatedStorage.

2. In ServerScriptService, add a script. Create references for ReplicatedStorage and the RemoteEvent:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local countdownEvent = ReplicatedStorage:WaitForChild("CountdownEvent")
```

3. Create a countdown using a `for` loop. On every iteration, fire the event and pass back the current countdown:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local countdownEvent = ReplicatedStorage:WaitForChild("CountdownEvent")

local secondsRemaining = 20
for count = secondsRemaining, 1, -1 do
    countdownEvent:FireAllClients(count)
    wait(1.0)
end
```

4. Get the client side going; in StarterGui, add a new ScreenGui (see Figure 12.4) and a TextLabel. This is where you'll display the countdown.

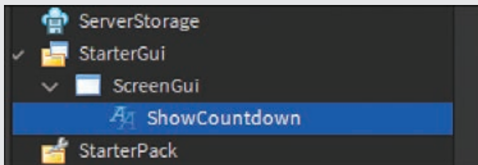


FIGURE 12.4
Set up the GUI to be seen by everyone.

5. In ScreenGui, add a LocalScript. This is where you'll put the code that you want to run whenever the event is fired. In the example shown in Figure 12.5, it's been named `DisplayManager`.

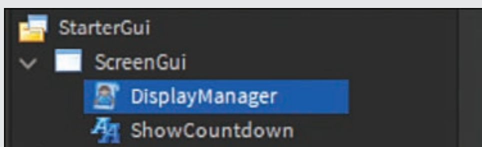


FIGURE 12.5
Add a new LocalScript to StarterPlayerScripts.

6. Set up your references; then create a new function to be called when the event is fired:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local countdownEvent = ReplicatedStorage:WaitForChild("CountdownEvent")

-- Get the ScreenGui and the TextLabel
local screenGui = script.Parent
local countDisplay = screenGui.ShowCountdown

local function onTimerUpdate(count)
    -- Set the TextLabel to match the incoming count
    countDisplay.Text = count
end

-- Call "onTimerUpdate()" when the server fires the remote event
countdownEvent.OnClientEvent:Connect(onTimerUpdate)
```

TIP

Checking Your Code

The first thing to check is your references. Make sure that the name of your events, instances, and GUI elements matches what's being referenced in your code. It's okay if they're different from the example code as long as you're paying attention. Secondly, it's always a good idea to check all of your code using the network simulator and on a live published game.

Communicating from the Client to the Server

Now it's time to find out how to go the opposite direction—sending information from the client to the server. Remember, in this case, you don't need a response from the server. If a client needs to make a change that affects the entire server, the client can send a remote request to the server.

In this case, the client side would fire the RemoteEvent:

```
remoteEvent:FireServer(infoToPass)
```

And functions would be connected on the server side:

```
local function functionName(player, passedInfo)
    print(player.Name)
    -- Stuff to do
end
remoteEvent.OnServerEvent:Connect(functionName)
```

Take a look at the example function in the preceding code. The player that triggered the event was passed in automatically. The first parameter will always have to take that into account.

▼ TRY IT YOURSELF

Pick a Map, Any Map

One way to create variety within your experience is to allow for different map choices. The world in Figure 12.6 has a map picker where a person can choose between three different locations. Depending on your experience, people might be teleported to that location or that map might then be loaded up. For this example, instead of teleporting people, you'll create the chosen map by cloning it out of ServerStorage.

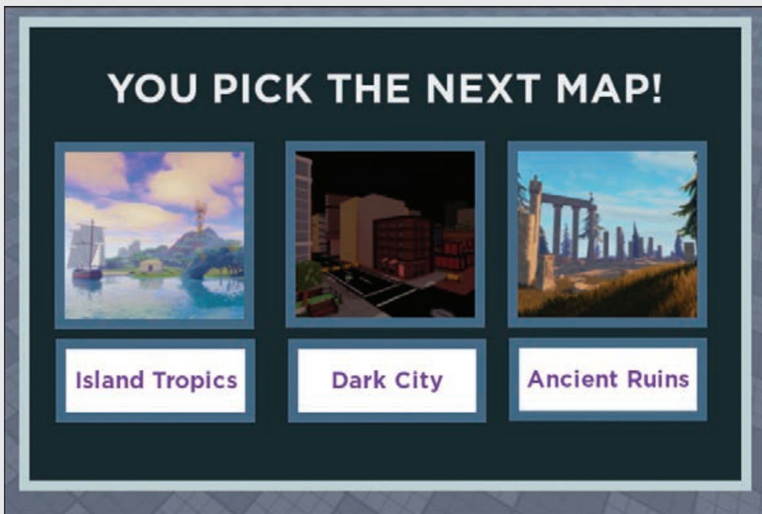


FIGURE 12.6
This GUI has three different map choices.

Set Up

All of the buildings and props of a map can be grouped into a single model. In this section, you set up a few simple models to practice with, and create the GUI buttons:

1. In ServerStorage, add a new folder named `Maps`.
2. Place three different models into the Maps folder (see Figure 12.7). Make sure each model has a unique name. To group parts into a model, select all desired parts; then right-click and select Group (Cmd+G or Ctrl+G).

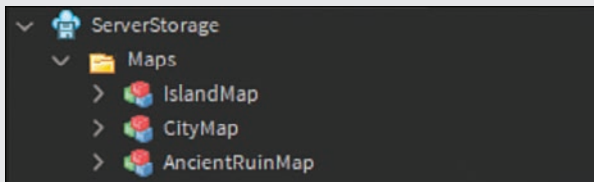


FIGURE 12.7
Three models holding all the parts for three different maps.

TIP

Practice with Simple Models

For the sake of this Try It Yourself, it's OK if the models consist of just a part or two.

3. In ReplicatedStorage, add a RemoteEvent named **`MapPicked`**. (See Figure 12.8.)

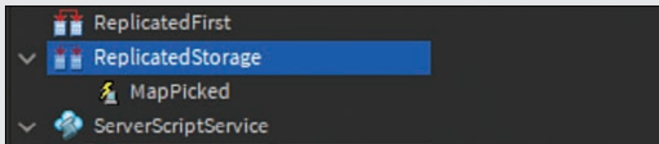


FIGURE 12.8
Add a new RemoteEvent within ReplicatedStorage.

4. In StarterGui, add a newScreenGui, and within that, add a frame named **`MapSelection`**. (See Figure 12.9.) Frames allow you to group different GUI elements together.

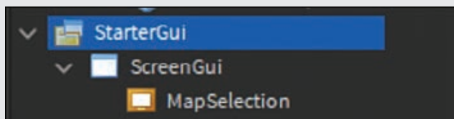


FIGURE 12.9
A new GUI frame holds the buttons.

5. Select the frame, and insert three TextButtons (see Figure 12.10). Make sure each button's name matches the maps in ServerStorage (refer to Figure 12.8). You'll use the name to clone the correct map.

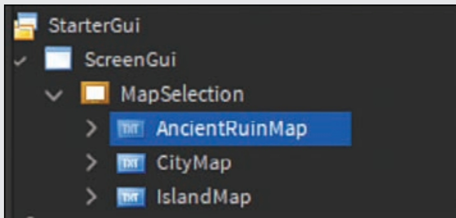


FIGURE 12.10
Three buttons have names matching the map models.

Client Side

The buttons appear on the client, which makes it possible for the map to be chosen and a message sent to the server:

1. Select one of the buttons and insert a LocalScript.
2. Set up your references for the RemoteEvent, the button, and the frame:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local mapPicked = ReplicatedStorage:WaitForChild("MapPicked")

local button = script.Parent
local frame = button.Parent
```

3. Create a new function connected to the button's `Activated` event:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local mapPicked = ReplicatedStorage:WaitForChild("MapPicked")

local button = script.Parent
local frame = button.Parent

local function onButtonActivated()

end

button.Activated:Connect(onButtonActivated)
```

4. Inside, use `FireServer()` to send the name of the button that was picked and then make the frame invisible:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local mapPicked = ReplicatedStorage:WaitForChild("MapPicked")

local button = script.Parent
local frame = button.Parent

local function onButtonActivated()
```

```
        mapPicked:FireServer(button.Name)
        frame.Visible = false
    end

    button.Activated:Connect(onButtonActivated)
```

ServerSide

On the server side, the selected map will be cloned from ServerStorage:

1. In ServerScriptService, add a new script.
2. Set up the references for the RemoteEvent, ServerStorage, and the Maps folder:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local mapPicked = ReplicatedStorage:WaitForChild("MapPicked")

local ServerStorage = game:GetService("ServerStorage")
local mapsFolder = ServerStorage:WaitForChild("Maps")
```

3. Add one more reference that will be used for the current map. Set it to `nil` for now. You'll use this variable to delete the old map when a new one is made so they aren't stacked on top of each other:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local mapPicked = ReplicatedStorage:WaitForChild("MapPicked")

local ServerStorage = game:GetService("ServerStorage")
local mapsFolder = ServerStorage:WaitForChild("Maps")

local currentMap = nil
```

4. Create a new function and connect it to the RemoteEvent's event named `OnServerEvent`:

```
local function onMapPicked(player, chosenMap)
end

mapPicked.OnServerEvent:Connect(onMapPicked)
```

5. Find the chosen map within the Maps folder:

```
local function onMapPicked(player, chosenMap)
    local mapChoice = mapsFolder:FindFirstChild(chosenMap)
end
```

6. Check to make sure the chosen map was found; then destroy the old map and make a copy of the new one:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local mapPicked = ReplicatedStorage:WaitForChild("MapPicked")

local ServerStorage = game:GetService("ServerStorage")
```

```
local mapsFolder = ServerStorage:WaitForChild("Maps")

local currentMap = nil

local function onMapPicked(player, chosenMap)
    local mapChoice = mapsFolder:FindFirstChild(chosenMap)

    if mapChoice then
        -- Check for the old map and destroy it
        if currentMap then
            currentMap:Destroy()
        end
        -- Make a copy of the new map
        currentMap = mapChoice:Clone()
        currentMap.Parent = workspace
    else
        print("Map choice not found")
    end
end

mapPicked.OnServerEvent:Connect(onMapPicked)
```

7. Test it out! If it works, place copies of the LocalScript within the other two buttons.

TIP

Troubleshooting Tip

If the scripts don't work as expected, make sure you're testing the correct button. If the buttons are on top of each other, or you try the wrong button, nothing will happen.

Communicating from the Server to One Client

If you need to pass information to a specific player, like maybe if they were randomly chosen to be the hunter or "it" in a game, things are a little different. You need to make sure that you send along the player when the server fires the RemoteEvent, before you pass along any additional information.

Server side:

```
remoteEvent:FireClient(player, additionalInfo)
```

Client side:

```
local function onServerEvent(player, additionalInfo)
    -- Whatever you want to happen
end
remoteEvent.OnClientEvent:Connect(onServerEvent)
```

Even if you don't plan to use the player argument, it still needs to be sent and accounted for on the local side. It's just a quirk of the RemoteEvent needing to know specifically who to send a message to.

Communicating from Client to Client

The fourth and final way to use RemoteEvents is from client to client. Clients can't communicate directly with each other; they have to go through the server, so you actually use a combination of the three previous methods.

The client would use `FireServer(infoToPass)`, and from there, the server would pass the information back to one or all of the clients.

Summary

RemoteEvents are a really versatile way to send information between the server and the client because you can connect several functions to the same event. Since the information goes only one way, the RemoveEvents don't have to wait for a response, and are generally faster and easier to use than RemoteFunctions whenever a response isn't required.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. True or false: RemoteEvents can send information to the server, and return a response to the client.
2. True or false: RemoteEvents can only have one function bound to them at a time.
3. To send a message from a client to a server, you would use the function ____.
4. True or false: The server automatically receives the name of the client that fired the RemoteEvent.

Answers

1. False. RemoteEvents can only send information in one direction. They can't wait for a response.
2. False. Instead of binding a single function to a RemoteEvent, as many functions as you want can be connected. This is one of their benefits over RemoteFunctions.
3. To send a message from a client to a server, use `FireServer()`—for example, `mapPicked:FireServer(button.Name)`.
4. True, which is why you need to account for the incoming player argument when connecting functions to the RemoteEvent.

Exercise

Once the map is picked, be sure to announce the map choice to everyone in the server, as shown in Figure 12.11.

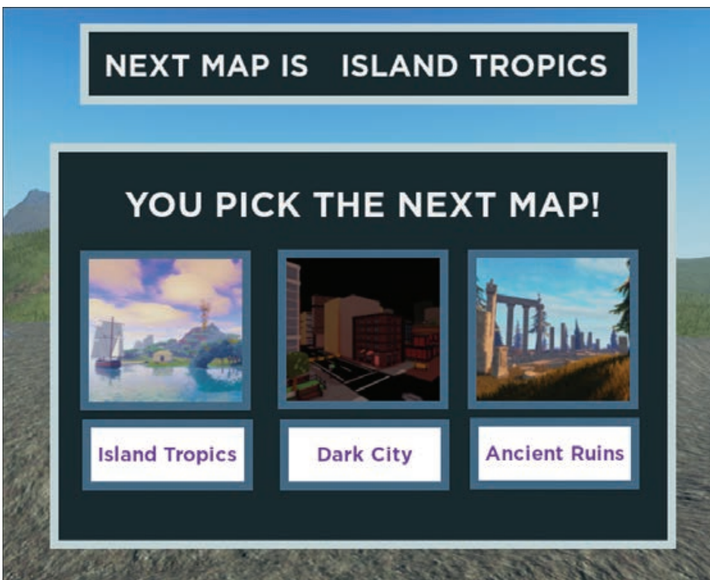


FIGURE 12.11
The map choice is announced.

Tips

- Remember that clients can't talk directly to other clients.
- You don't need another RemoteEvent.

HOUR 13
Using ModuleScripts

What You'll Learn in This Hour:

- What ModuleScripts are
- How to create with ModuleScripts
- How to create a jump pad
- What abstractions are
- How to avoid repeating yourself

There's a good chance your experience will have a lot of things in it: a lot of buttons, a lot of things to touch, a lot of things to pick up. As much as possible, you want to make sure you're not ending up with lots of duplicate code in your world for handling all of these objects. Having multiple copies of scripts everywhere is hard to manage and hard to update. Imagine making the same change to dozens of item pickup scripts or going trap by individual trap to change how much damage they do.

This hour explains ModuleScripts, another tool in your belt for keeping your code centralized and easy to update. It also covers a little bit more about the general principle of Don't Repeat Yourself—also known as DRY—coding and how to organize your code to make it easier to tweak and update.

Coding Things Just Once

ModuleScripts are a unique script object that allows you to store functions and variables that can then be used by both scripts and LocalScripts. This way, you can create a main source of information for things like gold pickups, monster stats, and even button behavior. Instead of having a dozen or even a hundred scripts to update if a change needs to be made, you only have to update the ModuleScript.

Scripts and LocalScripts will still be needed for accessing the module script, but the code within them can be kept to the bare minimum.

Placing ModuleScripts

Where you place ModuleScripts depends on how you plan on using them. If they will only be used by server scripts, you should put them in ServerStorage, where they're better protected. If client-side LocalScripts need to use the ModuleScripts, you can put them in ReplicatedStorage. (See Figure 13.1.)

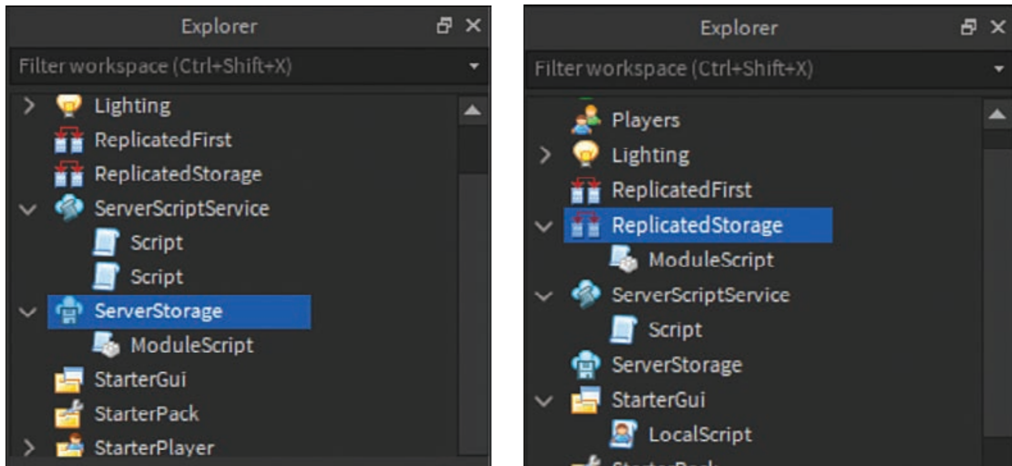


FIGURE 13.1
ModuleScripts in ServerStorage (on left) can only be accessed by scripts. ReplicatedStorage (on right) can be used by both scripts and LocalScripts.

Understanding How ModuleScripts Work

By default, every ModuleScript starts out with this code:

```
local module = {}
return module
```

These should always be the first and last lines of code for the ModuleScript. Notice the curly brackets? All of the code within the ModuleScript is placed into a table and then returned on the last line. Within the table, all of the module's shared functions and variables are stored.

Naming ModuleScripts

The first thing you'll need to do with the table is update the name to match that of the script, as shown in the following code and Figure 13.2. The name should match the purpose of all the shared functions, such as ShopManager, TrapManager, or PetManager.

TIP

Script Naming

The word Manager is pretty commonly used to mark a script that tells something else what to do. So ButtonManager can be interpreted as "Tells the button what to do."

```
local TrapManager = {}

function TrapManager.modifyHealth(player, amount)
    -- Code
end

return TrapManager
```

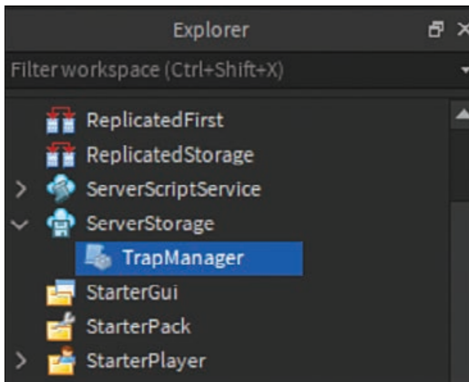


FIGURE 13.2
The name of the ModuleScript and the table name match exactly.

Notice that the names of both the ModuleScript and the internal table shown in Figure 13.2 are in PascalCase—the first letter of each word is capitalized. If you want to learn more about common naming conventions in Roblox, some style guide rules can be found in the appendix.

Adding Functions and Variables

To add a new function or variable to the module table, use dot notation similar to how you have worked with dictionaries before, as shown here:

```
local ModuleName = {}

-- Add a variable
ModuleName.variableName = 100

-- Adds a function
```

```
function ModuleName.functionName(parameter)
    -- Code goes here
end
return ModuleName
```

Remember, anything added to the module table *must* be typed between `local ModuleName = {}` and `return ModuleName`.

Understanding Scope in ModuleScripts

If you look back at the last code snippet, you'll notice the keyword `local` isn't used for the added function and variable:

```
-- The keyword 'local' isn't used
function ModuleName.functionName(parameter)
    -- Code goes here
end
```

Typing `local` in front of variables and functions means they are usable only by that code chunk. Usually that's what we want, but ModuleScripts are different. The whole point is to make the code shareable:

```
local ScoreManager= {}

-- The shareable function is not local
function ScoreManager.scoreCalculator(originalScore, newPoints)
    local newScore = originalScore + newPoints
    return newScore
end
return ScoreManager
```

However, variables only used by the ModuleScript, like variables *within* a function, should still be `local`:

```
local ScoreManager= {}

function ScoreManager.scoreCalculator(originalScore, newPoints)
    -- This variable doesn't need to be shared outside the function
    local newScore = originalScore + newPoints
    return newScore
end
return ScoreManager
```

Using Modules in Other Scripts

ModuleScripts don't run code on their own. Instead, the variables and functions are accessed by other scripts and run from there.

Within a script or LocalScript, use `require()` and pass in the ModuleScript's location:

Script example:

```
local ServerStorage = game:GetService("ServerStorage")
local ModuleName = require(ServerStorage.ModuleName)
```

The script will load in the module table, making the module's variables and functions available for use.

To use a variable or a function from the module, use the ModuleScript's name, followed by the name of what you need. The following code sample has a practice variable with the value of 7 within a ModuleScript. The snippet after that demonstrates that variable being accessed from a normal Script object. Figure 13.3 shows the results in Output.

ModuleScript code in ServerStorage

```
local PracticeModuleScript = {}

PracticeModuleScript.practiceVariable = 7

function PracticeModuleScript.practiceFunction ()
    print("This came from the practice ModuleScript")
end

return PracticeModuleScript
```

Script code in ServerScriptService

```
local Serverstorage = game:GetService("ServerStorage")
local PracticeModuleScript = require(Serverstorage.PracticeModuleScript)

-- This should print '7'
print(PracticeModuleScript.practiceVariable)

-- This should print the message within practiceFunction()
PracticeModuleScript.practiceFunction()
```

The resulting output looks like Figure 13.3. Notice that the source for the printed statement on the first line is the script, whereas the source for the second line is the ModuleScript.

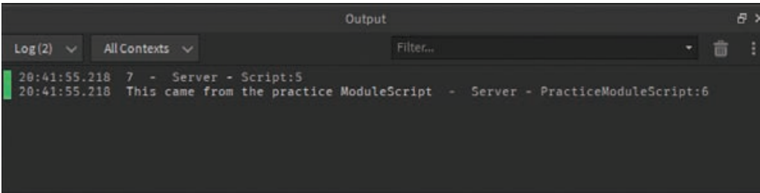


FIGURE 13.3
The printed results are shown in Output.

Make sure you match the names of the ModuleScript, functions, and variables *exactly*; otherwise, they won't work. There's nothing wrong with copying and pasting to make sure things are the same.

Also, be sure not to make any changes to the ModuleScript while the experience is running. The table can't be refreshed. Once the module table is loaded, using `require()` again will only return the same table.

TRY IT YOURSELF

Create a Jump Pad

Jump pads (see Figure 13.4) are always a crowd pleaser because they let people in your experience reach areas they couldn't have otherwise. In this Try It Yourself, you're going to make a jump pad that will have a script handling only the most basic code that's not likely to need updating.

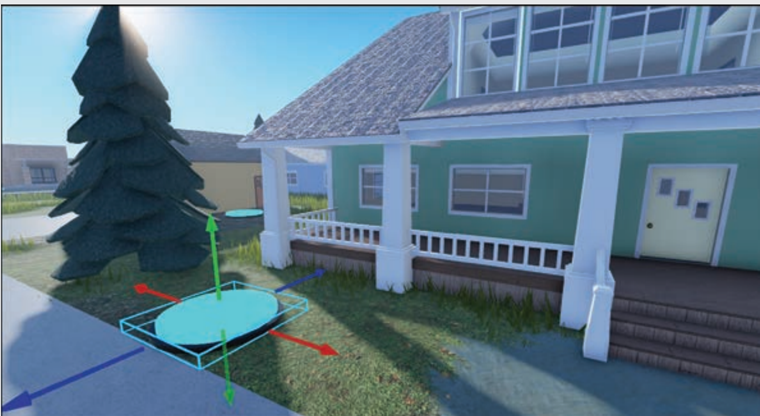


FIGURE 13.4
The blue pad allows people to reach places they couldn't otherwise, like the roof of this house.

Set Up

Set up the script objects and the jump pad; in the next section, you'll work on the code.

1. Add a part or a mesh. The jump pad in Figure 13.4 is just a neon blue base part.
2. Insert a script into the jump pad.
3. In ServerStorage, create a new ModuleScript named JumpPadManager.

ModuleScript

You'll start with setting up the ModuleScript. The ModuleScript will handle how high and how long the player will jump. To do so, you need to get the character's HumanoidRootPart, which handles the basic motion of a person. A VectorForce object will be added to the HumanoidRootPart, which will cause the person to shoot up for as long as the VectorForce exists. You may not be very familiar with some of the concepts used yet, but you'll use them more in upcoming hours.

All of the heavy lifting for the script will be done here, leaving only a few lines of code to create in the script:

1. Rename the table **`JumpPadManager`**:

```
local JumpPadManager = {}

return JumpPadManager
```

2. Create a `local` constant for how long the jump will last:

```
local JumpPadManager = {}

-- Local variable because they're not needed outside of this ModuleScript
local JUMP_DURATION = 1.0

return JumpPadManager
```

3. Create a second `local` constant for jump direction like the one shown next. VectorForce requires X, Y, Z coordinates to know which way to send things. The middle number, Y, makes things go upward:

```
local JumpPadManager = {}

local JUMP_DURATION = 1.0
local JUMP_DIRECTION = Vector3.new(0, 6000, 0)

return JumpPadManager
```

TIP

Moving and Animating Objects

You'll learn more about Vector3, X, Y, Z coordinates, and how to move and animate objects, in the next hour. If you're feeling brave, you can experiment with the X and Z values for front-and-back and side-to-side force.

4. Add a new function to the table:

```
local JumpPadManager = {}

local JUMP_DURATION = 1.0
local JUMP_DIRECTION = Vector3.new(0, 6000, 0)

-- Not local because the jump pads need these functions
function JumpPadManager.jump(part)
end

return JumpPadManager
```

5. This part is similar to making a trap. Find the part's parent and use that to search for a humanoid. If it finds Humanoid, then search for the HumanoidRootPart:

```
-- Top of ModuleScript

function JumpPadManager.jump(part)
    local character = part.Parent
    local humanoid = character:FindFirstChildWhichIsA("Humanoid")

    if humanoid then
        local humanoidRootPart = character:FindFirstChild("HumanoidRootPart")
    end
end

return JumpPadManager
```

6. Search for a VectorForce instance, even though it won't exist until you add one. You'll need this for the debounce in the next step:

```
-- Top of ModuleScript

function JumpPadManager.jump(part)
    local character = part.Parent
    local humanoid = character:FindFirstChildWhichIsA("Humanoid")

    if humanoid then
        local humanoidRootPart = character:FindFirstChild("HumanoidRootPart")
        local vectorForce = humanoidRootPart:FindFirstChild("VectorForce")
```

```
    end
end

return JumpPadManager
```

7. If there's not a VectorForce, add one. This makes sure there is only ever one VectorForce applied:

```
local JumpPadManager = {}

-- Top of ModuleScript

function JumpPadManager.jump(part)
    local character = part.Parent
    local humanoid = character:FindFirstChildWhichIsA("Humanoid")

    if humanoid then
        local humanoidRootPart = character:FindFirstChild("HumanoidRootPart")
        local vectorForce = humanoidRootPart:FindFirstChild("VectorForce")
        if not vectorForce then
            vectorForce = Instance.new("VectorForce")
        end
    end
end

return JumpPadManager
```

8. Set the Force property to `JUMP_DIRECTION` and then attach and parent it to the HumanoidRootPart, as shown here:

```
-- Top of ModuleScript

function JumpPadManager.jump(part)
    local character = part.Parent
    local humanoid = character:FindFirstChildWhichIsA("Humanoid")

    if humanoid then
        local humanoidRootPart = character:FindFirstChild("HumanoidRootPart")
        local vectorForce = humanoidRootPart:FindFirstChild("VectorForce")
        if not vectorForce then
            vectorForce = Instance.new("VectorForce")
            vectorForce.Force = JUMP_DIRECTION
            vectorForce.Attachment0 = humanoidRootPart.RootRigAttachment
            vectorForce.Parent = humanoidRootPart
        end
    end
end

return JumpPadManager
```

TIP

Keeping VectorForce and HumanoidRootPart Together

The attachment makes sure that the VectorForce instance doesn't become separated from the HumanoidRootPart.

9. Finally, wait for `JUMP_DURATION` before destroying the BodyVelocity:

```
local JumpPadManager = {}

-- Local because they're not needed outside of this ModuleScript
local JUMP_DURATION = 1.0
local JUMP_DIRECTION = Vector3.new(0, 6000, 0)

-- Not local because the jump pads need these functions
function JumpPadManager.jump(part)
    local character = part.Parent
    local humanoid = character:FindFirstChildWhichIsA("Humanoid")

    if humanoid then
        local humanoidRootPart = character:FindFirstChild("HumanoidRootPart")
        local vectorForce = humanoidRootPart:FindFirstChild("VectorForce")
        if not vectorForce then
            vectorForce = Instance.new("VectorForce")
            vectorForce.Force = JUMP_DIRECTION
            vectorForce.Attachment0 = humanoidRootPart.RootRigAttachment
            vectorForce.Parent = humanoidRootPart
            wait(JUMP_DURATION)
            vectorForce:Destroy()
        end
    end
end

return JumpPadManager
```

Script

On the script side, we want only the bare minimum of code. All it will do is load the ModuleScript and call `JumpPadManager.jump(otherPart)` whenever something touches the part:

1. Load JumpPadManager into the script:

```
local ServerStorage = game:GetService("ServerStorage")
local JumpPadManager = require(ServerStorage.JumpPadManager)
```

2. Connect a function to the `Touched` event, and inside, pass the touching part to the ModuleScript:

```
local ServerStorage = game:GetService("ServerStorage")
local JumpPadManager = require(ServerStorage.JumpPadManager)
```

```
local jumpPad = script.Parent

local function onTouch(otherPart)
    JumpPadManager.jump(otherPart)
end

jumpPad.Touched:Connect(onTouch)
```

Test everything out! If you get nil errors, there's a good chance it's due to a misspelling somewhere. Make sure all your naming and capitalization matches exactly.

Don't Repeat Yourself

As you've followed along in hours, more and more we said to centralize your code or to write your code in a way that can be reused.

Think of the resource items like gold and logs in Hour 9. Instead of having different scripts for gold and logs, they both use the same set of scripts, but the code allows for differing information to be processed. This is part of the general practice of DRY coding. DRY stands for *Don't Repeat Yourself*, and it's a concept that applies to all coding and coding languages—not just Lua and Roblox Studio.

The opposite, WET (Write Everything Twice), is generally considered to be a bad thing and is used to mean you have a lot of duplicate code in your scripts.

Dealing in Abstractions

A key part of DRY coding is abstractions. *Abstraction* is the process of pulling out the most important information, and hiding everything that you don't need to deal with right now.

A lot of things in Roblox Studio are abstracted for you. Think of all the functions or methods where you only have to call the function and pass information in. The functions are reusable abstractions. When called, users get the benefits of the function without having to rewrite or even look at the rest of the code.

A common example in coding languages is `print()`. Most of its code is hidden, so the coder can focus on what needs to be printed and not on how to get individual pixels to show up on the screen.

ModuleScripts also allow you to set up the abstractions necessary for good DRY coding practices. ModuleScripts can act as a Single Source of Truth (SSOT), meaning that information and functions needed by multiple scripts can all be kept in the one ModuleScript.

A good way to know if you need to set up an abstraction is if you think you'll need to use a variable or function in three or more places. So in the resource game example, there was a good chance that you would want people to collect not just logs and gold but potentially other resource types as well—maybe eventually things like iron, berries, or wool, too.

Summary

Abstractions provide a simplified representation of something larger by leaving out details. When deciding whether to create an abstraction, look for code that is often reused but with small changes each time. For example, a generic item like a backpack can be abstracted to a reusable function that looks up price and capacity.

Taking time to plan and structure code with abstractions helps coders focus on what's important. This investment in time keeps programs better organized and makes updating them easier.

Q&A

Q. Why not just always leave out `local` and make most variables and functions shareable?

A. Making variables local is usually best practice. It makes your code run a bit faster and eliminates the chances for errors and accidentally overwritten information. Nonlocal variables, such as in ModuleScripts, should always be the exception, not the rule.

Q. Can you go overboard with abstractions and DRY coding?

A. You can definitely go overboard with creating abstractions. Generally, though, if you can think of a future where you might want to use the same piece of code more than two or three times, abstractions are worth the effort.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. My ModuleScript is named `RoundManager`. What should the first and last lines of my ModuleScript look like?
2. Functions can be added to a ModuleScript using ____.
3. If a ModuleScript needs to be used by a LocalScript, where should it be placed?
4. If a ModuleScript only needs to be used by scripts, where should it be placed?

5. If a ModuleScript needs to be used by both LocalScripts and scripts, where should it be placed?

6. What does DRY stand for?

Answers

1. If your ModuleScript is named RoundManager, the first and last lines of code should be

```
local RoundManager = {}
    -- Code
return RoundManager
```

2. Dot notation, such as function MyModule.myfunction()

3. In ReplicatedStorage

4. In ServerStorage

5. In ReplicatedStorage

6. Don't Repeat Yourself

Exercise

Practice creating a module script by using code you're familiar with. Make a series of trap parts people need to avoid if they don't want to lose all their health. Figure 13.5 shows an example.

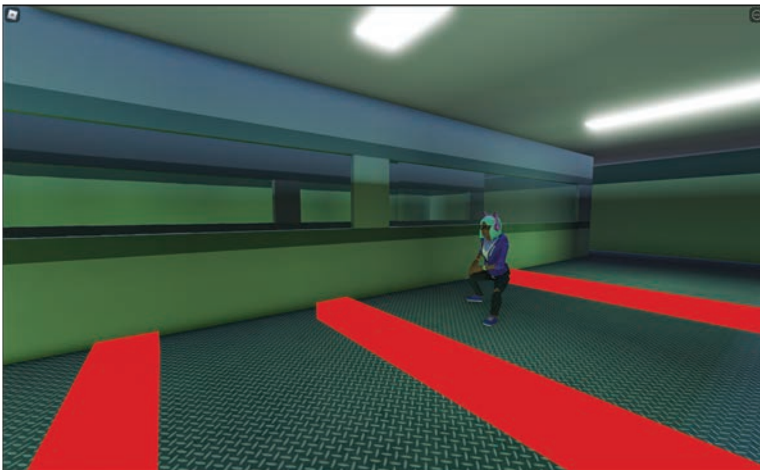


FIGURE 13.5
Red traps in a basement hallway maze

This page intentionally left blank

HOUR 14

Coding in 3D World Space

What You'll Learn in This Hour:

- ▶ How to work with XYZ coordinates
- ▶ How to place objects where you want them with CFrames
- ▶ What is the difference between world and local objects
- ▶ How to control where a character jumps with `RelativeTo`

In the last hour, you used a module script to create a jump pad that shot people straight up in the air. This hour covers how objects are placed in 3D space and how you can make parts spawn anywhere you want in the world.

Understanding X, Y, and Z Coordinates

Before you get coding, you need to understand how objects are placed and rotated within 3D space. Within the 3D world, every object can be positioned on a grid controlled by the three axes, X, Y, and Z. Up and down is controlled by the Y axis, whereas front-and-back and side-to-side are controlled by X and Z, respectively. In Figure 14.1, the green arrow represents the Y axis, the red arrows are the X axis, and the blue arrow is the Z axis.

If you don't already have it showing, turn on the View Selector (see Figure 14.2) using View > Actions > View Selector to allow you to see which way in the world your camera is facing.

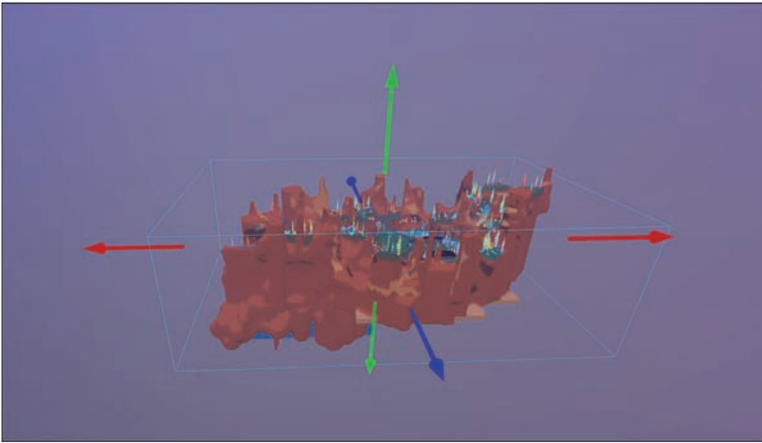


FIGURE 14.1
A zoomed-out view of an entire piece of terrain. The world axes are marked with red, green, and blue arrows.

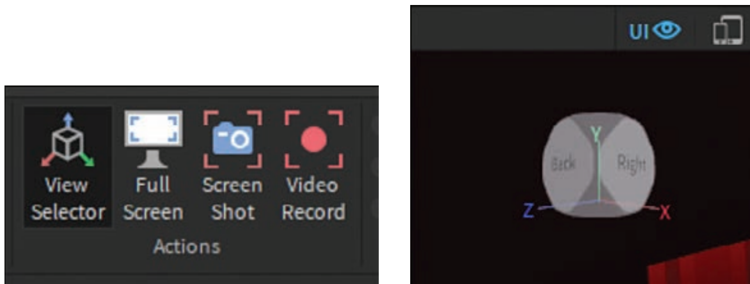
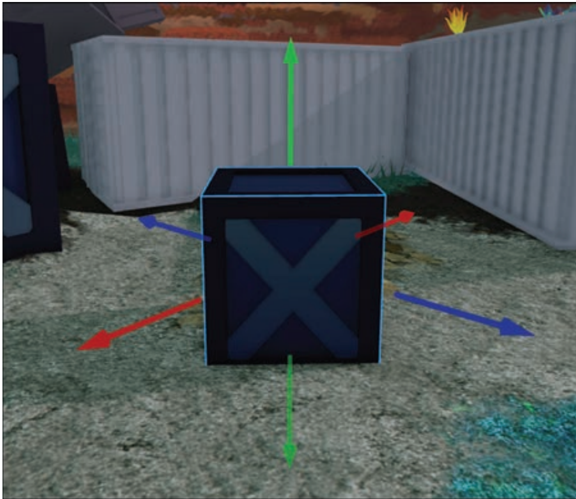


FIGURE 14.2
If the View Selector is enabled, you can see the axes represented there.

When you select an object with the Move tool, you can see the three axes represented with red (X), green (Y), and blue (Z) arrows. Dragging an object within the world space will update the X, Y, and Z position values in the Property window (see Figure 14.3). If you were to place an object at the center of the world, its X, Y, and Z position would be 0, 0, 0.



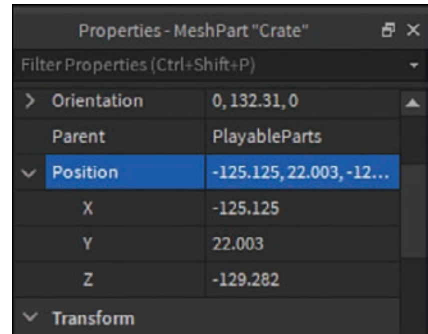


FIGURE 14.3
When you drag a part around, you can see the X, Y, and Z values updated in the Property window.

Refining Placement with CFrame Coordinates

If you want to place an object or a player in a specific place, you need to understand CFrames. With Cframes, you can place them exactly where you want them instead of only spawning objects at the center of the world.

CFrame stands for Coordinate Frame. Every object in the 3D space has one. The default value of a CFrame is 0, 0, 0—which is why new objects appear at the center of the world. To update an object's position, assign a new CFrame value using `CFrame.new()`:

Example:

```
local part = script.Parent
part.CFrame = CFrame.new(1, 4, 1)
```

You can set the X, Y, and Z positions individually, as shown in the preceding code, or you can pass in Vector3 data, as shown here:

```
local vector3 = Vector3.new( 1, 4, 1)
part.CFrame = CFrame.new(vector3)
```

▼ TRY IT YOURSELF

Place an Object Where Something Else Is

There are a number of ways to find the coordinates to use for a new CFrame position. One way is to find another part that's already where you want the part to go. Basic parts have a property named Position that uses Vector3 values. For this quick Try It Yourself, use the Position property of one part to set the CFrame of a brand-new part.

1. Create a part somewhere in your world. Name it something distinct like **`Marker`** and create a reference for it:

```
local marker = workspace.Marker
```

2. Create a new part instance. By default, new parts are unanchored, so make sure to anchor it in place:

```
local marker = workspace.Marker

local newPart = Instance.new("Part")
newPart.Anchored = true
```

3. Get the new part's CFrame and set it to `CFrame.new()`:

```
local marker = workspace.Marker

local newPart = Instance.new("Part")
newPart.Anchored = true
newPart.CFrame = CFrame.new()
```

4. Pass in the marker's position and then parent the new part to the workspace. Test your code; you should end up with two parts in the same place:

```
local marker = workspace.Marker

local newPart = Instance.new("Part")
newPart.Anchored = true
newPart.CFrame = CFrame.new(marker.Position)
newPart.Parent = workspace
```

You might be asking yourself why you wouldn't always use the Position property? The answer is because Position only works with parts and not with models. We cover that more in a bit.

Offsetting CFrames

Quite often, you don't want to place something exactly in the same place; instead, you want to place it above or a little to the side. You can combine CFrame and Vector3 values to make that happen. In the following example, a Vector3 adds four studs to the Y value that was passed into `CFrame.new`:

```
local marker = workspace.Marker

local newPart = Instance.new("Part")
newPart.Anchored = true

-- Will place the new part 4 studs above Marker
newPart.CFrame = CFrame.new(marker.Position) + Vector3.new(0, 4, 0)
newPart.Parent = workspace
```

Adding Rotations to CFrames

You can add rotation values to CFrames. To rotate an existing object, take the current CFrame and multiply it by the number of degrees you want it to rotate using `CFrame.Angles()`:

```
local spinner = script.Parent
local ROTATION_AMOUNT = CFrame.Angles(0, math.rad(45), 0)


while wait(0.5) do
    -- Take spinner's current CFrame and rotate it.
    spinner.CFrame = spinner.CFrame * ROTATION_AMOUNT
end
```

`CFrame.Angles` also takes in three values for X, Y, and Z. The preceding snippet rotates `spinner` on the Y axis. One thing to note is that it does not operate using degrees. It uses radians. Radians are a math concept used when working with the arc of a circle. Luckily, you don't need to know how to use radians. Instead, `math.rad()` can convert from degrees to radians for you.

So, if you want a part to rotate 20 degrees on the X axis, it would look like this:

```
local ROTATION_AMOUNT = CFrame.Angles(math.rad(20), 0, 0)
part.CFrame  = part.CFrame * ROTATION_AMOUNT
```

Working with Models

As mentioned earlier, individual base parts have a property called Position. However, models do not. To move the position of a model, you need to get the PrimaryPart of the model. To demonstrate, we've taken a very simple cloud model made from spheres that have been grouped together, as shown in Figure 14.4.

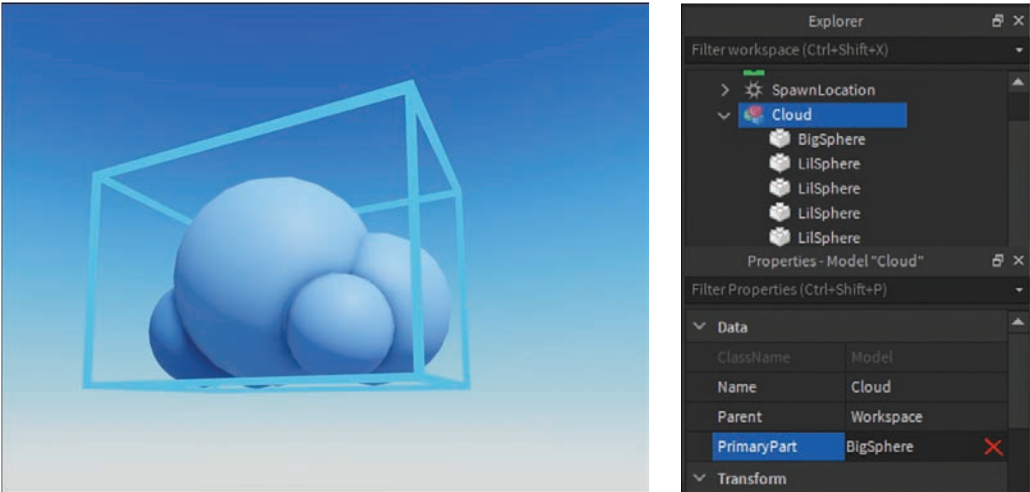


FIGURE 14.4
The cloud model is made from several parts. In Properties, you can see PrimaryPart is set to BigSphere.

TIP

Grouping Parts into a Model

To group parts into a model, right-click a selection of parts and select Group.

This cloud model cannot be moved using the earlier method. You need to use `SetPrimaryPartCFrame()` and then pass in the new CFrame:

```
local cloud = workspace.Cloud
cloud:SetPrimaryPartCFrame(CFrame.new(0, 20, 0))
```

TIP

Setting the Primary Part

You can set the PrimaryPart of a model in Properties. Click PrimaryPart; then in Explorer, click the part you want to designate as the main part of the model.

Understanding World Coordinates and Local Object Coordinates

In a 3D experience, there's actually *two* sets of coordinates you need to think about. The first is the world coordinate, as we've been talking about so far—how something is placed and rotated according to the X, Y, and Z axis of the entire 3D space.

The other is the local object axis—how an object is positioned and rotated *relative to* itself. The X, Y, and Z of an individual object might not line up with the world. In Figure 14.5, you see the world axis on the left and the box's own local axis shown on the right.

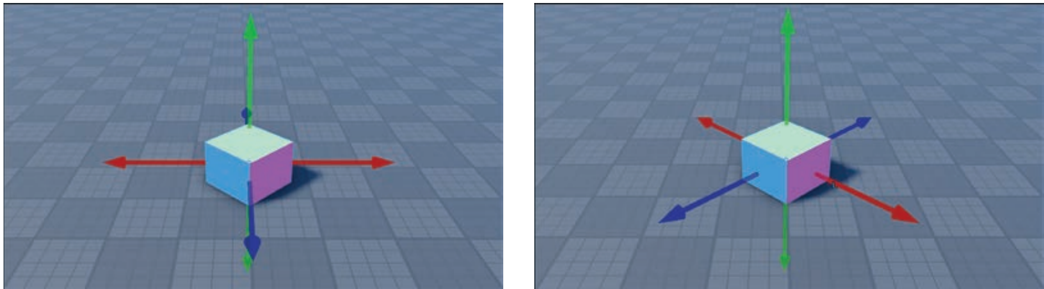


FIGURE 14.5
The image on the left shows the object in relation to the world axis. Each object has its own separate X, Y, and Z that might not line up with the world, as shown on the right.

Think of it this way. The world you walk around in has global compass directions (north, south, east, west) that don't change no matter which way you face. But your personal left, right, forward, and back moves and rotates as you do.

You can change your scale and rotate tools to see world or local coordinates by pressing Cmd/Ctrl + L. You can tell you're in local mode if you see a little L in the corner along the red X axis, as shown in Figure 14.6.

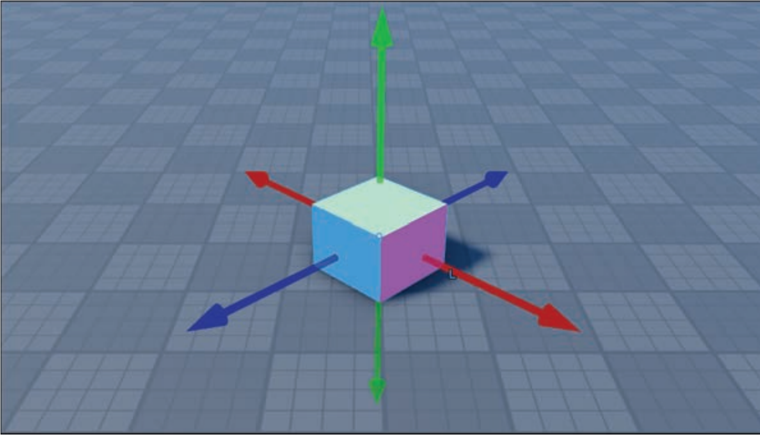


FIGURE 14.6
The Move tool and Rotation tools are in local mode.

▼ TRY IT YOURSELF

Super Jump Relative to the Player

The current jump pad uses global coordinates. Whenever somebody steps on the pad, they are always shot in the same direction, no matter which way they (or the jump pad) are facing. In this Try It Yourself, you tweak the code so that it uses which way the player is facing using `relativeto()`.

Use the code from Hour 13 to make slight changes:

1. In JumpPadManager, find the `JUMP_DIRECTION` constant. Change the Z value to **`-6000`**:

```
local JumpPadManager = {}
-- Local because they're not needed outside of this ModuleScript
local JUMP_DURATION = 0.5
local JUMP_DIRECTION = Vector3.new(0, 6000, -6000)
```

2. Set the `RelativeTo` property of the VectorForce to `Enum.ActuatorRelativeTo.Attachment0`:

```
local JumpPadManager = {}

-- Local because they're not needed outside of this ModuleScript
local JUMP_DURATION = 0.5
local JUMP_DIRECTION = Vector3.new(0, 6000, -6000)
```

```
-- Not local because the jump pads need these functions
function JumpPadManager.jump(part)
    local character = part.Parent
    local humanoid = character:FindFirstChildWhichIsA("Humanoid")
    if humanoid then
        local humanoidRootPart = character:FindFirstChild("HumanoidRootPart")
        local vectorForce = humanoidRootPart:FindFirstChild("VectorForce")
        if not vectorForce then
            vectorForce = Instance.new("VectorForce")
            vectorForce.Force = JUMP_DIRECTION
            vectorForce.Attachment0 = humanoidRootPart.RootRigAttachment
            vectorForce.RelativeTo = Enum.ActuatorRelativeTo.Attachment0
            vectorForce.Parent = humanoidRootPart
            wait(JUMP_DURATION)
            vectorForce:Destroy()
        end
     end
end
return JumpPadManager
```

TIP

VectorForce Is Relative to an Attachment

This will set it so the VectorForce is relative to Attachment0, which in this case is connected to the HumanoidRootPart.

3. Test it out. Your character should be boosted in the local direction they are facing instead of being boosted along the world axis.

TIP

Different Avatars Weigh Different Amounts

Just like people in real life, an avatar's weight depends on its size and type of accessories. How high an avatar is boosted will vary based on the weight.

Summary

Congratulations! You now have the power to place objects anywhere in the world. And not just parts, you can also teleport people around. Everything in the 3D world space, including people's characters, has coordinates that can be found along the X (red), Y (green), and Z (blue) of the world.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. What axis is up in Roblox Studio world space?
2. If you want to create a new Cframe from Vector3 information, use the function ______.
3. To rotate a CFrame, use the function _____.
4. To convert degrees to radians, use ________.
5. To rotate an object, you ________ the position CFrame and `CFrame.Angles()`.
6. To place an object to the side or above an object, you would ____ the position CFrame and a new Vector3.

Answers

1. The green Y axis is up.
2. `CFrame.New()`
3. `CFrame.Angles()`
4. `math.rad()`
5. Multiply
6. Add the desired CFrame position and how much you want to offset it by.

Exercise

People can be teleported from place to place within a world by updating their character's CFrame information. This might be so that players can cross a canyon like the one shown in Figure 14.7, or it could be teleporting participants from a lobby to an arena. For this exercise, create a part that teleports players to another part.

FIGURE 14.7
People can use the purple parts to teleport across the gap.

Tips

- For the sake of practice, you only need to worry about teleporting the player in one direction.
- You can find the PrimaryPart within a person's character.

This page intentionally left blank

HOUR 15
Smoothly Animating Objects

What You'll Learn in This Hour:

- What tweening properties are
- How to move parts smoothly over time
- How to use the completed event

CFrames allow you to suddenly move things from one place to another in the blip of an eye. But what if you don't want things to blip? Maybe you'd instead like them to transition smoothly between two points or change from one color to another. That's where tweens come in. In this hour, you use tweens to smoothly change the position and color of a block, but the same principles also apply to working with GUIs,

Understanding Tweens

Tweens take a starting point, like a position on a map, or a certain color, and smoothly change to an end point over time. To use tweens, you need to get TweenService, as shown here:

```
local TweenService = game:GetService("TweenService")
```

TRY IT YOURSELF ▼

Tween the Color of a Part

Tweens are one of those things that make more sense if you see them demonstrated. Follow these steps to set up a simple tween that changes the color of a part over time:

1. Create a new part and attach a script.
2. Within the script, get the tween service and create a variable that points to the target part:

```
local TweenService = game:GetService("TweenService")
local part = script.Parent
```

3. A `TweenInfo` controls how the transition is handled. Create a new `TweenInfo` and pass in `5.0` so that the transition to the new color will take 5 seconds:

```
local TweenService = game:GetService("TweenService")
local part = script.Parent
local tweenInfo = TweenInfo.new(5.0)
```

4. The `TweenService` needs a table to hold the goal values for each property to be changed—in this case, the final color of the part:

```
local TweenService = game:GetService("TweenService")
local part = script.Parent
local tweenInfo = TweenInfo.new(5.0)

local goal = {}
goal.Color = Color3.fromRGB(11, 141, 255)
```

TIP

Use Any RGB Value

This color happens to be bright blue.

5. Use `TweenService:Create()` to bring together the target part, `TweenInfo`, and the goal table:

```
local goal = {}
goal.Color = Color3.fromRGB(11, 141, 255)

-- Put together the target part, TweenInfo, and goal values
local tween = TweenService:Create(part, tweenInfo, goal)
```

6. Give a smidge of time to let the experience load, and then tell the tween to play:

```
local TweenService = game:GetService("TweenService")
local part = script.Parent

local goal = {}
goal.Color = Color3.fromRGB(11, 141, 255)
local tweenInfo = TweenInfo.new(5.0)
local tween = TweenService:Create(part, tweenInfo, goal)

-- Delay to give things time to load properly
wait(2.0)
-- Tell the tween to play
tween:Play()
```

TIP

Transition Time

If you don't add the `wait()`, you'll probably miss the beginning of the transition. If it's a short transition time or a long load time, you might miss it altogether. If this tween were to be played after an event is fired, `wait()` wouldn't be needed.

Setting `TweenInfo` Parameters

You can tween as many properties of an object as you like; you just have to add them to the table. Additionally, there's a lot more you can do to customize how the tween behaves as it interpolates, meaning transitions, to the goal values.

Table 15.1 lists all of `TweenInfo`'s parameters.

TABLE 15.1 `TweenInfo`'s Parameters

Parameter	What It Does
`Time [number, seconds]`	Determines how long it takes for the tween to reach its goals
`EasingStyle [Enum]`	Determines how the tween behaves toward its goal
`EasingDirection [Enum]`	Establishes the direction of the EasingStyle functions
`RepeatCount [number]`	Establishes the number of times the tween executes after its initial run
`Reverses [Bool]`	Designates whether the tween runs the reverse tween following its initial run
`DelayTime [number, seconds]`	Determines the elapsed time before the tween executes

The code looks like this:

```
local tweenInfo = TweenInfo.new(
    2.0, -- Time
    Enum.EasingStyle.Linear, -- EasingStyle
    Enum.EasingDirection.Out, -- EasingDirection
    -1, -- RepeatCount (when less than zero the tween will loop indefinitely)
    true, -- Reverses (tween will reverse once reaching its goal)
    0.0 -- DelayTime
)
```

This is a rare case where all the arguments are usually put on their own line just to make them easier to read. Not everything needs to be filled in, but you can't skip arguments. You can see a full list of EasingStyles and EasingDirections in the appendix.

Also notice that since `TweenInfo` is not a table, you *don't* put a comma after the last argument.

Create Elevator Doors

Practice using more of the `TweenInfo` parameters by setting up a sliding door that can be used by something like an elevator or a fancy office building, like the one shown in Figure 15.1.

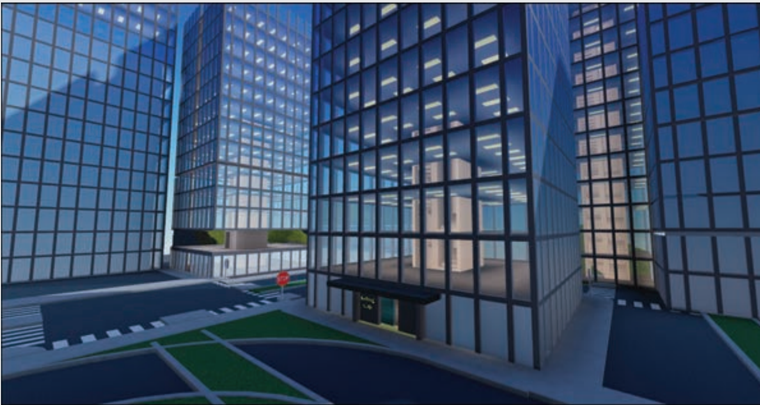


FIGURE 15.1
Fancy office buildings might have equally fancy sliding doors on an elevator.

After sliding the door open, the tween will pause and then reverse direction.

Set Up

You just need a part for this Try It Yourself. Keep in mind that if you use a model, you need to make sure to account for that in the code later.

1. Use a glass part somewhere to act as a sliding door.
2. Insert a ProximityPrompt named SlidingDoorPrompt.
3. Set HoldDuration to `0.5`.

Script

This script uses the first three parameters of `TweenInfo` to create a part that moves smoothly over a certain distance.

1. In ServerScriptService, create a new script.
2. Get the necessary service for the ProximityPrompt and the Tween Service:

```
local ProximityPromptService = game:GetService("ProximityPromptService")
local TweenService = game:GetService("TweenService")
```

3. Set up a new function and connect it to the ProximityPrompt's `PromptTriggered` event. Remember to add a check for which ProximityPrompt was triggered:

```
local ProximityPromptService = game:GetService("ProximityPromptService")
local TweenService = game:GetService("TweenService")

local function onPromptTriggered(prompt, player)

    if prompt.Name == "SlidingDoorPrompt" then
    end
end
ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)
```

4. Determine the target:

```
local ProximityPromptService = game:GetService("ProximityPromptService")
local TweenService = game:GetService("TweenService")

local function onPromptTriggered(prompt, player)

    if prompt.Name == "SlidingDoorPrompt" then
        local door = prompt.Parent
    end
end

ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)
```

5. Create a table with the goal CFrame values for when the door opens. Your values may differ:

```
if prompt.Name == "SlidingDoorPrompt" then
    local door = prompt.Parent
    local goal = {}
    goal.CFrame = door.CFrame + Vector3.new(0, 0, 5)
end
```

TIP

Your Vector3 Information May Be Different

For the sake of simplicity, this code is just moving the door along the Z axis. You may need to use a different axis. If you were to use this code with several doors rotated in different directions, you may even want to experiment with relative coordinates.

6. Create a new `TweenInfo` and set the duration to 1 second, the easing style to Linear, and the easing direction to In:

```
if prompt.Name == "SlidingDoorPrompt" then
    local door = prompt.Parent
    local goal = {}
    goal.CFrame = door.CFrame + Vector3.new(0, 0, 5)

    local tweenInfo = TweenInfo.new(
        1.0,
        Enum.EasingStyle.Linear,
        Enum.EasingDirection.In
    )

end
```

TIP

TweenInfo Isn't a Table

Keep in mind TweenInfo isn't a table, so it doesn't need a comma after the last argument.

TIP

Autocomplete

As you're typing the Enums, notice that the autocomplete (see Figure 15.2) and IDE hints (see Figure 15.3) can help you out quite a bit.

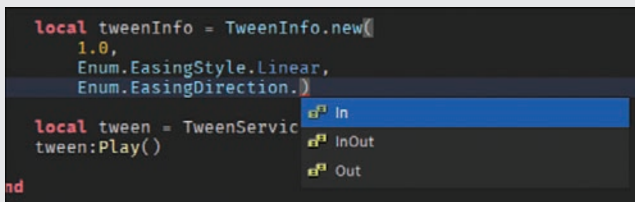


FIGURE 15.2
Use the autocomplete to make filling out the TweenInfo easier.

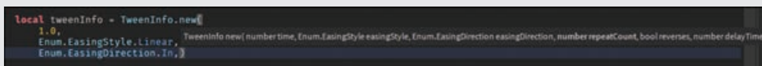


FIGURE 15.3
Also pay attention to the IDE hints to keep track of the parameter order.

7. Put it all together using `TweenService:Create()`:

```
if prompt.Name == "SlidingDoorPrompt" then
    local door = prompt.Parent
    local goal = {}
    goal.CFrame = door.CFrame + Vector3.new(0, 0, 5)

    local tweenInfo = TweenInfo.new(
        1.0,
        Enum.EasingStyle.Linear,
        Enum.EasingDirection.In
    )

    local openDoor = TweenService:Create(door, tweenInfo, goal)
end

ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)
```

8. Play the tween:

```
local ProximityPromptService = game:GetService("ProximityPromptService")
local TweenService = game:GetService("TweenService")

local function onPromptTriggered(prompt, player)

    if prompt.Name == "SlidingDoorPrompt" then
        local door = prompt.Parent
        local goal = {}
        goal.CFrame = door.CFrame + Vector3.new(0, 0, 5)

        local tweenInfo = TweenInfo.new(
            1.0,
            Enum.EasingStyle.Linear,
            Enum.EasingDirection.In
        )

        local openDoor = TweenService:Create(door, tweenInfo, goal)
        openDoor:Play()
    end
end

ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)
```

Chaining Tweens Together

Once one tween finishes, you may need a second tween to run—for example, if you want the door to remain open for a little while and then a second tween closes the door.

For cases like this, you can use the tween's event, `Completed`. First, wait for the starting tween's Completed event to fire, and then use a second `wait()` to control how long before the door closes, as shown here:

```
local ProximityPromptService = game:GetService("ProximityPromptService")
local TweenService = game:GetService("TweenService")

local DOOR_OPEN_DURATION = 2.0

local function onPromptTriggered(prompt, player)
    if prompt.Name == "SlidingDoorPrompt" then
        local door = prompt.Parent
        local openGoal = {}
        openGoal.CFrame = door.CFrame + Vector3.new(0, 0, 5)

        local closeGoal = {}
        closeGoal.CFrame = door.CFrame

        local tweenInfo = TweenInfo.new(
            1.0,
            Enum.EasingStyle.Linear,
            Enum.EasingDirection.In
        )

        local openDoor = TweenService:Create(door, tweenInfo, openGoal)
        local closeDoor = TweenService:Create(door, tweenInfo, closeGoal)

        -- Play the first tween
        openDoor:Play()
        -- Wait for the Completed event to fire
        openDoor.Completed:Wait()
        -- Pause, and then play the next tween
        wait(DOOR_OPEN_DURATION)
        closeDoor:Play()
    end
end

ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)
```

TIP

Don't Forget to Debounce

The script as it is doesn't include any sort of debounce. If someone were to keep opening the door, the door could move farther and farther to one side. Don't forget to add a debounce if you actually place this code in your experience.

Summary

Tweens allow you to smoothly transition almost any of a target's properties to a new value over time. Although the Try It Yourself exercises in this hour only showed one property being changed at a time, you can add as many properties to the goal table as you would like.

The pattern for creating tweens is

1. Get `TweenService`.
2. Set the target part.
3. Set up the `TweenInfo` information.
4. Create the goal dictionary.
5. Pass the target part, `TweenInfo`, and goal dictionary in:

   ```
   local tween = TweenService:Create(part, tweenInfo, goal)
   ```
6. Play the tween.

It's OK to depend on the IDE or to look up the order of the parameters if you can't remember them off the top of your head. All engineers do that. To revert a tween to the original state, you can enable reverse, or you can chain tweens together by listening for a tween's event, `Completed`.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. To use tweens, what service do you need?
2. Can you combine CFrames and tweens?
3. True or false: You can simply skip a parameter if you don't want to use it.
4. Whether a tween reverses at the end is controlled by a ____ value.
5. True or false: You can only tween one property at a time.
6. True or false: Tweens play automatically once they are set up.

Answers

1. `TweenService`
2. You can use tweens with many properties, including CFrame.
3. False: While you don't always have to pass an argument for every parameter, you can't skip over parameters.
4. Boolean
5. False: You can change multiple properties with the same tween.
6. False: Once a tween is set up, it needs to be told when to play.

Exercise

For this exercise, tween the color of a SpotLight (like the lights in Figure 15.4) so it changes over time and never stops looping.

FIGURE 15.4
Color-changing lights attract people to this fun party place.

Tip: You don't need multiple tweens.

HOUR 16
Solving Problems with Algorithms

What You'll Learn in This Hour:

- ▶ How an algorithm is defined
- ▶ What the three main parts of an algorithm are
- ▶ How to sort arrays
- ▶ How to sort dictionaries

This hour introduces a new computer science term to add to your vocabulary: *algorithms*. You'll get a better idea of how you've already been putting this concept into action and go a step further with Roblox Studio's built-in sorting algorithms, which you can use to do things like sort items in a shop from lowest to highest price or rank participants in an FPS according to how many kills they have.

Defining Algorithms

Algorithms are precise instructions for solving a problem. You've actually created a number of algorithms throughout this book. For example, you've created algorithms that figure out a player's health points after stepping on a trap and algorithms that determine a player's total points every time somebody touches a part.

For a function to be an algorithm, it has to have three very clear stages:

1. Information is taken in, typically through parameters.
2. That information is acted on using ordered steps.
3. A solution is given.

Let's take a very simple algorithm, one that solves the problem of dividing two numbers.

Problem: What's the dividend of two numbers?

```
local x = 20
local y = 9

-- Gives back the dividend of two numbers
local function divide(first, second)
    return first / second
end

local result = divide(x,y)
print(result)
```

In this little code sample, you can see the following:

1. Two numbers are passed through the parameters.
2. The step of dividing the two numbers takes place.
3. The dividend is returned.

Algorithms can be used over and over again with different inputs, much like a function can be used with any two numbers. Most algorithms will have more steps than this, but they'll never have infinite steps. To be a true algorithm, the code has to give you a solution to your problem.

Sorting an Array

A classic place where people need algorithms is when sorting things—taking a list of names, objects, or numbers and putting them in order. In Roblox experiences, this information will most likely be stored in a dictionary or array.

Let's start with arrays. `table.sort(arrayName)` uses a sorting algorithm to arrange the values within arrays numerically or alphabetically.

▼ TRY IT YOURSELF

Sort a List of Names

For this example, you'll sort an array of names in alphabetical order and then print it.

1. Create a list of three names:

   ```
   local nameArray = {"Cat", "Mei", "Ana"}
   ```

2. Pass the name of the array into `table.sort()`:

   ```
   local nameArray = {"Cat", "Mei", "Ana"}
   table.sort(nameArray)
   ```

3. Print the updated array:

```
local nameArray = {"Cat", "Mei", "Ana"}
table.sort(nameArray)
print(nameArray)
```

Tables in Output appear as little triangles, as shown in Figure 16.1.

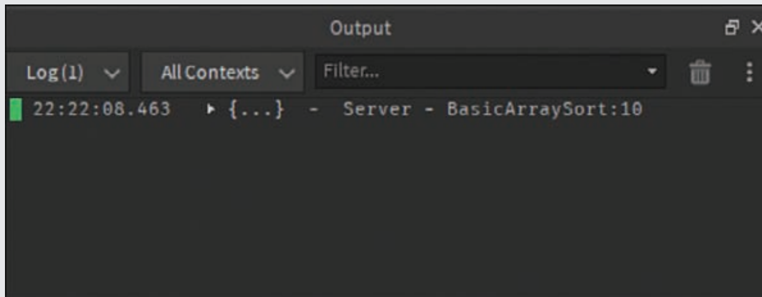


FIGURE 16.1
The triangle indicates a collapsed view of a table.

Click the triangle to expand it and see the whole table, as shown in Figure 16.2.

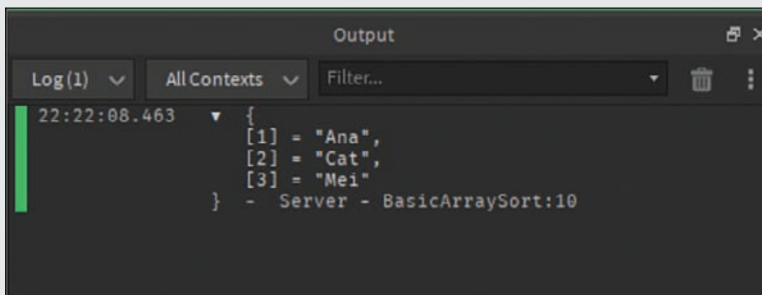


FIGURE 16.2
The now-sorted table is expanded; notice the index on the left.

The same method can be used to arrange numeric values. The following code snippet sorts the array into ascending order, as shown in Figure 16.3.

```
local testArray = {5, 2, 2, 10}
table.sort(testArray)
print(testArray)
```

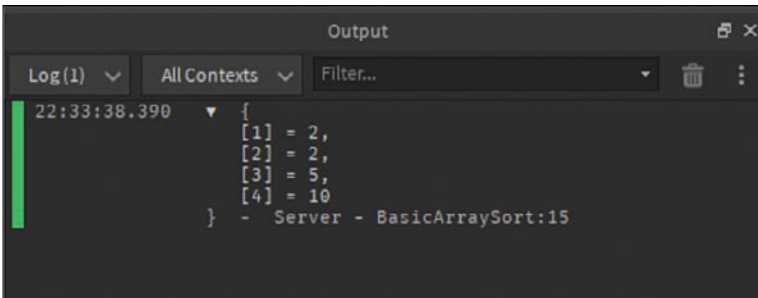


FIGURE 16.3
This result shows numeric values after they've been sorted.

WARNING

Be Careful with Numbers and Strings When Sorting

If you try to sort an array of mixed data types, such as numbers and strings, all you'll get is an error:

```
-- Strings and numbers cannot be compared
local mixedArray = {5, "Frog", 2, 10}
```

You could use `tostring()` to convert from number to string types, but be aware that will cause `table.sort` to put things in alphabetical order, like so:

```
--  Numbers converted to strings will sort alphabetically
local stringArray = {"10", "2", "5", "Frog"}
```

Sorting in Descending Order

Both the alphabetical and numerical data in the previous examples was sorted in ascending order. But what if the person who has the *most* points is actually the important factor? There's a second parameter that can be passed into `table.sort()`, and that parameter enables you to control how exactly the table is sorted.

`table.sort()` works by going through the whole array two values at a time and comparing the values against each other. By default, the function compares the two values using the less-than operator (<), making the lesser numbers come first.

To customize the sorting algorithm, a new function for comparing two values needs to be created and then passed in along with the array, as shown in the following snippet. Here, the greater-than operator is used, so greater values will appear first:

```
-- First, set up the array
local testArray = {5, 2, 2, 10}
```

```
-- Second, create a function that shows how two values should be compared
local function DescendingSort(a, b)
    return a > b
end

-- Third, pass the function into table.sort() along with the array.
table.sort(testArray, DescendingSort)
print(testArray)
```

The results of the code snippet will look like Figure 16.4.

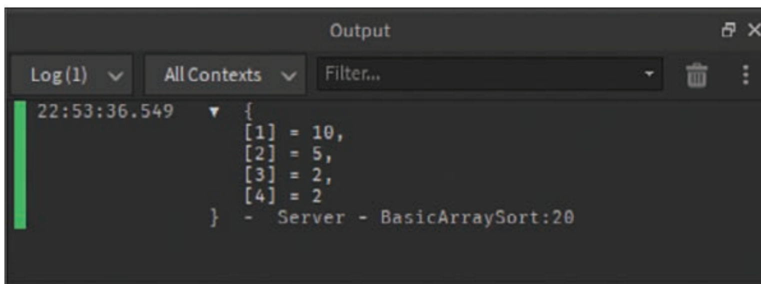


FIGURE 16.4
The array is now sorted in descending order.

Sorting a Dictionary

A very important thing to remember about dictionaries in Lua is that they do not have a guaranteed order. They might sometimes do things in order, but you can't depend on it. In other words, you can't actually sort a dictionary.

Instead, what you have to do is convert the dictionary to an array, the results of which might look like the following table, which has an unsorted dictionary on the left and that same dictionary after it's been converted to an array on the right. We show you the actual method for converting a dictionary to an array in a moment.

Unsorted Dictionary	Unsorted Array of Dictionaries
`local IngredientDictionary = {`	`local sortingArray = {`
`healthBerry = 10,`	`{name = "healthBerry", amount = 10},`
`staminaOnion = 5,`	`{name = "staminaOnion", amount = 5},`
`speedPepper = 1,`	`{name = "speedPepper", amount = 1},`
`}`	`}`

Notice that the right-hand column is truly an array, but each value is itself a dictionary. Arrays can hold any valid data type, including dictionaries, and this lets you tag the data to make it easier to sort.

Once sorted by name, the array might look like the following:

Array Sorted by Name

```
local sortingArray = {
     {name = "healthBerry", amount = 10},
     {name = "speedPepper", amount = 1},
     {name = "staminaOnion", amount= 5},
}
```

▼ TRY IT YOURSELF

Find Who Has the Most Points

Take a moment to create a dictionary with imaginary player scores and practice converting a dictionary to an array. Once the dictionary is converted to an array, a slightly more specific function for comparing values will be created and then passed in.

1. Set up a dictionary of four or five imaginary players and their scores, similar to the following:

```
local playerScores = {
    Ariel = 10,
    Billie = 5,
    Yichen = 4,
    Kevin = 14,
}
```

2. Create a new array to hold the sorted results:

```
local playerScores = {
    Ariel = 10,
    Billie = 5,
    Yichen = 4,
    Kevin = 14,
}

local sortedArray = {}
```

3. Use `pairs()` to go through the original dictionary and insert each key/value pair into the array as its own mini dictionary:

```
-- Previous code

local sortedArray = {}
```

```
-- Go through dictionary, and assign each key/value pair to an index
for key, value in pairs(playerScores) do
    table.insert(sortedArray, {playerName = key, points = value})
end
```

4. Set up the comparison function. This time, you want to compare the points and have the greater amounts be first:

```
-- Previous code

local function sortByMostPoints(a, b)
    return a.points > b.points
end
```

TIP

Use Dot Notation to Access Dictionary Keys

While sorting, the algorithm looks at two values and evaluates them with the comparison function. In this case, each value happens to be a table, so you can use dot notation to navigate to the correct key.

5. Pass the array and function into `table.sort()` and print the results:

```
local playerScores = {
    Ariel = 10,
    Billie = 5,
    Yichen = 4,
    Kevin = 14,
}

local sortedArray = {}
-- Go through dictionary, and assign each key/value pair to an index
for key, value in pairs(playerScores) do
    table.insert(sortedArray, {playerName = key, points = value})
end

-- Set up comparison function
local function sortByMostPoints(a, b)
    return a.points > b.points
end

-- Pass in array and function
table.sort(sortedArray, sortByMostPoints)
print(sortedArray)
```

Sorting by Multiple Pieces of Information

The final thing to cover for using the sorting algorithm is to sort by multiple pieces of information. Picture a fantasy world where you enter a store and can buy multiple types of weapons like those shown in Table 16.1.

TABLE 16.1 Unsorted Weapons

Weapon Name	Weapon Type	Price
Iron Sword	Sword	250
Light Bow	Bow	150
Training Sword	Sword	100
Dwarven Axe	Axe	300
The Galactic Slash	Sword	500

The unsorted information makes it difficult to find what you're looking for. To make it easier to shop, you might want to list the weapons by type and then by price, as shown in Table 16.2.

TABLE 16.2 Weapons Sorted by Type

Weapon Name	Weapon Type	Price
Dwarven Axe	Axe	300
Light Bow	Bow	150
Training Sword	Sword	100
Iron Sword	Sword	250
The Galactic Slash	Sword	500

In code form, the original array might look like this:

```
-- Original array
local inventory = {
    {name = "Iron Sword", weaponType = "Sword", price = 250},
    {name = "Light Bow", weaponType = "Bow", price = 150},
    {name = "Training Sword", weaponType = "Sword", price = 100},
    {name = "Dwarven Axe", weaponType = "Axe", price = 300},
    {name = "The Galactic Slash", weaponType = "Sword", price = 500},

}
```

Since this is already an array, you *wouldn't* need to convert it, which means the next thing would be to set up the comparison function. First you can compare types, or, if they're the same type, compare type and price:

```
-- Sort first by most weapon type, then by price
local function sortByTypeAndPrice(a, b)

    return (a.weaponType < b.weaponType)
    or (a.weaponType == b.weaponType and a.price < b.price)
end
```

TIP

Keywords Can't Be Key Names

On its own, `type` is a keyword. That's why it's best to use something like `weaponType` instead of `type`.

Finally, pass both the array and comparison function into `table.sort()` and print the result:

```
table.sort(inventory, sortByTypeAndPrice)
print(inventory)
```

Summary

You've used algorithms before, but now you know what they're called. There's lots of different sorting algorithms, each with their own strengths and weaknesses. Behind the scenes, Roblox Studio `table.sort()` uses what's called a quick sort. If you're interested in learning more about sorting algorithms or creating your own, there's a wealth of information about them on the Internet; just search "sorting algorithms."

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. What is an algorithm?
2. What are three components of an algorithm?
3. What's the first parameter of `table.sort()`?
4. What's the optional second parameter of `table.sort()`?

5. If you want to use `table.sort()` to list the players with the quickest times, use the ___ operator.

6. True or false: Dictionaries can be sorted using `table.sort()` as well.

Answers

1. An algorithm is a specific set of steps that can be used to solve a problem.

2. Takes in inputs, goes through a set of ordered steps, outputs a solution.

3. The name of the array to sort.

4. A function for a custom comparator can be passed into the second parameter.

5. The less-than operator (<).

6. False. Dictionaries have to be converted to arrays before being sorted.

Exercise

A common set of stats in competitive experiences are a person's number of kills, deaths, and assists (other people's kills they've contributed to). Take a sample dictionary such as the one shown here, and sort it according to who has the most kills. If people are tied for kills, prioritize who has assisted other members of the team the most.

Tips

- Include three to five different players and try to make sure some of them are tied for most kills. You can use the following example dictionary if you like:

```
local playerKDA = {
    Ana = {kills = 0, deaths = 2, assists = 20},
    Beth = {kills = 7, deaths = 5, assists = 0},
    Cat = {kills = 7, deaths = 0, assists = 5},
    Dani = {kills = 5, deaths = 20, assists = 8},
    Ed = {kills = 1, deaths = 1, assists = 8},
}
```

- You will have to convert the dictionary to an array.

- Printing the array right after it's created and before it's sorted can be a good troubleshooting step for making sure the array looks the way you expect.

You can find the solution in the appendix.

HOUR 17

Saving Data

What You'll Learn in This Hour:

- ▶ How to enable Data Stores
- ▶ How to save data between sessions
- ▶ How to protect data with protected calls
- ▶ How to create a player database
- ▶ How to get and update saved data

Without a special mechanism in place to save information, anything that people in your experiences earn or accomplish between play sessions is lost. Once a person leaves the experience, points, gold, and purchases are forgotten. This hour covers how data can be saved so that nothing is lost between sessions.

Data saved from one session to another is kept in special tables, most typically in Data Stores. Data Stores work like a dictionary in which keys and values can be stored in the cloud with Roblox. This hour starts out by creating a box that keeps track of how many times it's been clicked; then the topic changes to how to minimize the chances that a player's data is lost.

Enabling Data Stores

Data Stores are only available to experiences saved on the Roblox cloud. To use Data Stores, you have to update some security settings for your experience:

1. Make sure the experience is published to Roblox and not just saved locally on your computer.
2. On the Home tab, click Game Settings.
3. Select Security and turn on Enable Studio Access to API Services. You can then save and exit Game Settings.

Creating a Data Store

Once you've enabled Data Stores, you can get `DataStoreService` within a script. Individual stores can be created and accessed with `GetDataStore("DataStoreName")`:

```
local DataStoreService = game:GetService("DataStoreService")
local dataStoreName= DataStoreService:GetDataStore("DataStoreName")
```

`GetDataStore("DataStoreName")` gets the matching Data Store or creates one with that name if it doesn't already exist.

Using Data in the Store

Remember, Data Stores function like a dictionary. All of the data within is stored using key-value pairs. Key-value pairs can be created and updated using `dataStoreName.SetAsync("KeyName", value)`. Or you can retrieve information using `dataStoreName.GetAsync("KeyName")`:

```
local DataStoreService = game:GetService("DataStoreService")
local dataStoreName = DataStoreService:GetDataStore("DataStoreName")

-- Update info in the Data Store, or create a new key/value pair
local updateStat = dataStoreName:SetAsync("StatName", value)

-- Retrieve information from the store using the key name
local storedStat = dataStoreName:GetAsync("StatName")
```

Keep in mind that `SetAsync()` will override the value of a key if it already exists. Once it's overwritten, that information is gone. That's one reason to make sure you always use unique key names.

▼ TRY IT YOURSELF

Track the Number of Clicks

Data Stores can hold any type of information that you could normally store in a dictionary. The Data Store you create in this Try It Yourself keeps track of how many times this crate has been clicked (see Figure 17.1). You don't want to update Data Stores too frequently because frequent updating can cause your game to lag or not actually save data. So you'll use a `while` loop to update the Data Store every so often.

FIGURE 17.1
This crate displays how many times it has been clicked.

Set Up

You need a part with a TextLabel, so do the following:

1. Insert a part or mesh.
2. Insert a SurfaceGui.
3. Insert a TextLabel into the SurfaceGui named `ClickDisplay`.
4. Select the crate and insert a ProximityPrompt named `CratePrompt`. Don't worry about setting up a HoldDuration. Your hierarchy should look something like Figure 17.2.

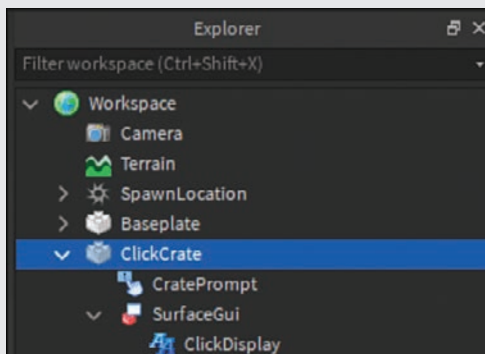


FIGURE 17.2
The finished hierarchy for the crate should look like this.

CrateManager

You'll use two scripts. The first one will manage the ProximityPrompt and update the Data Store. The second script will handle the default display:

1. In ServerScriptService, add a new script named **`CrateManager`**.
2. Get ProximityPromptService and DataStoreService.
3. Create a new Data Store named **`CrateData`**:

```
local ProximityPromptService = game:GetService("ProximityPromptService")
local DataStoreService = game:GetService("DataStoreService")
local crateData = DataStoreService:GetDataStore("CrateData")
```

4. Add two constants: one for how often players will be allowed to click the prompt and a second for how often the Data Store will be updated:

```
local DISABLED_DURATION = 0.1
local SAVE_FREQUENCY = 10.0
```

5. Get the total number of clicks so far from the Data Store; if there's no data yet, start at 0.

```
local DISABLED_DURATION = 0.1
local SAVE_FREQUENCY = 10.0

-- Get the current value of TotalClicks, or set to 0 if it doesn't exist
local totalClicks = crateData:GetAsync("TotalClicks") or 0
```

6. Set up everything you'll need for a new function connected to the prompt's `PromptTriggered` event:

```
local function onPromptTriggered(prompt, player)
    if prompt.Name == "CratePrompt" then
        prompt.Enabled = false
    end
end

ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)
```

7. Update `totalClicks` and the displayed text every time the player clicks:

```
-- Get the current value of TotalClicks, or set to 0 if it doesn't exist
local totalClicks = crateData:GetAsync("TotalClicks") or 0

local function onPromptTriggered(prompt, player)
    if prompt.Name == "CratePrompt" then
        prompt.Enabled = false

        local crate = prompt.parent
        local clickDisplay = crate:FindFirstChild("ClickDisplay", true)
```

```
            totalClicks = totalClicks + 1
            clickDisplay.Text = totalClicks

            wait(DISABLED_DURATION)
            prompt.Enabled = true
        end
    end

    ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)
```

TIP

Search the Children's Children

Adding true as a second parameter for `FindFirstChild()` is one way to go through all of an object's children and then go through the children's children to find what you're looking for.

8. Use a `while` loop to update the Data Store every so often:

```
local ProximityPromptService = game:GetService("ProximityPromptService")
local DataStoreService = game:GetService("DataStoreService")
local crateData = DataStoreService:GetDataStore("CrateData")

local DISABLED_DURATION = 0.1
local SAVE_FREQUENCY = 10.0

-- Get the current value of TotalClicks, or set to 0 if it doesn't exist
local totalClicks = crateData:GetAsync("TotalClicks") or 0

local function onPromptTriggered(prompt, player)
    if prompt.Name == "CratePrompt" then
        prompt.Enabled = false

        local crate = prompt.parent
        local clickDisplay = crate:FindFirstChild("ClickDisplay", true)

        totalClicks = totalClicks + 1
        clickDisplay.Text = totalClicks

        wait(DISABLED_DURATION)
        prompt.Enabled = true
    end
end

ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)

-- Update the Data Store every so often
while wait(SAVE_FREQUENCY) do
```

```
        crateData:SetAsync("TotalClicks", totalClicks)
    end
```

Crate Script

This second, shorter, script will be used to update the display text at the beginning of every session, before anyone has clicked on the crate.

1. Select the crate and insert a script.
2. Get `DataStoreService` and the Data Store you just created.
3. Create references for the crate and the TextLabel:

```
local DataStoreService = game:GetService("DataStoreService")
local crateData = DataStoreService:GetDataStore("CrateData")

local crate = script.Parent
local clickDisplay = crate:FindFirstChild("ClickDisplay", true)
```

4. Add a default value to be used in case the crate has never been clicked:

```
local DEFAULT_VALUE = 0
```

5. Retrieve the current number of clicks from the Data Store:

```
local DEFAULT_VALUE = 0
local totalClicks = crateData:GetAsync("TotalClicks")
```

6. Update the TextLabel to display the current count, or the default value if `TotalClicks` isn't found:

```
local DataStoreService = game:GetService("DataStoreService")
local crateData = DataStoreService:GetDataStore("CrateData")

local crate = script.Parent
local clickDisplay = crate:FindFirstChild("ClickDisplay", true)

local DEFAULT_VALUE = 0
local totalClicks = crateData:GetAsync("TotalClicks")

clickDisplay.Text = totalClicks or DEFAULT_VALUE
```

Test your place out. You should be able to stop the play test and then restart with the updated click amount displayed on the crate. Sometimes it might take a second to update values.

TIP

Make Sure To Use Unique Key Names

Keep in mind that keys within the Data Store need to be unique. If you duplicate the crate, any click on either crate will add to the total. If you want their totals to remain separate, each crate needs its own distinct key.

Limiting the Number of Calls

SetAsync() and GetAsync() are network calls, and using them often can be risky if you have a bad Internet connection or if you send more calls than the network can handle at a time. That's why a while loop was used to update the Data Store instead of updating every time a player clicked.

Each call request gets added to a queue, and there's only so many spots in line before the queue fills up and simply won't accept any more requests. Other good times to update the Data Store are when a player joins, leaves, or the server closes down.

Protecting Your Data

In addition to making sure you're not sending too many calls at once, another way to make sure network calls aren't missed or dropped is to *always* use a protected call—a pcall(). Protected calls track to make sure the network call went through. If the call wasn't successful, an error message is sent to help you figure out what went wrong.

A pcall() takes in a function and returns two values. The first value is a Boolean that states whether the call went through; the second value is for any returned error messages:

```
local setSuccess, errorMessage = pcall(functionName)
```

pcall() only accepts functions, so if you don't want to create a function beforehand, you can pass in the network call using an anonymous function:

```
local setSuccess, errorMessage = pcall(function()
   dataStoreName:SetAsync(key, value)
end)
```

You can then test the returned value to make sure it went through. Here, if setSuccess is false, the error message will be printed:

```
if not setSuccess then
   print(errorMessage)
end
```

If you were to update the while loop in the last section, it might look like this:

```
-- Update the Data Store every so often
while wait(SAVE_FREQUENCY) do
    local setSuccess, errorMessage = pcall(function()
        crateData:SetAsync("TotalClicks", totalClicks)
    end)

    if not setSuccess then
        print(errorMessage)
```

```
    else
        print("Current Count:")
        print(crateData:GetAsync("TotalClicks"))
    end
end
```

Saving Player Data

If you're saving player data, an important thing to remember is that a player's name can sometimes change. The safer alternative is to use the playerID to save the data. You can get the playerID from the player itself by using the following code :

```
local Players = game:GetService("Players")

local function onPlayerAdded(player)
    local playerKey = "Player_" .. player.UserId
end

Players.PlayerAdded:Connect(onPlayerAdded)
```

Using `UpdateAsync` to Update a Data Store

`UpdateAsync()`, which is similar to `SetAsync()`, should be used to update a Data Store if more than one server has a chance of accessing the same Data Store at the same time. If you're dealing with Robux or have an experience that is getting lots of people, you probably want to graduate to using `UpdateAsync()`. When called, `UpdateAsync()` returns the old value of a key and *then* updates it.

The blue highlight in the code snippet below is the `pcall()`:

```
local updateSuccess, errorMessage = pcall(function()
        pointsDataStore:UpdateAsync(playerKey, function(oldValue)
            local newValue = oldValue or 0
            newValue = newValue + GOLD_ON_JOIN
            return newValue
        end)
    end)
```

Within that, you get the Data Store as normal and use `UpdateAsync()` to pass in the key as the first parameter:

```
local updateSuccess, errorMessage = pcall(function()
        pointsDataStore:UpdateAsync(playerKey, function(oldValue)
            local newValue = oldValue or 0
            newValue = newValue + GOLD_ON_JOIN
            return newValue
        end)
    end)
```

Finally, the second parameter takes a function that accepts the old value and returns what the updated value should be. You can create the function beforehand or use an anonymous function as shown:

```
local updateSuccess, errorMessage = pcall(function()
        pointsDataStore:UpdateAsync(playerKey, function(oldValue)
            local newValue = oldValue or 0
            newValue = newValue + GOLD_ON_JOIN
            return newValue
        end)
end)
```

Summary

You can now save data, and that opens you up to opportunities like monetization that weren't available to you before. You can save just about any kind of data that you can think of. In an RPG experience, you can save people's skill level, weapon prowess, and inventory. In competitive games, you can save a player's rank or average KDA. You can also begin tracking whether people have purchased items in your experience, such as pets, power ups, and weapons.

Data Stores are really powerful; you just need to make sure that you are *always* using unique keys and verifying that you are both sending and receiving the correct data using `pcalls()`. The last thing you want is a community of people who are angry that their purchases have been lost to the ether.

Q&A

Q. What other ways can you save and update player data?

A. In addition to using `SetAsync()`, there are additional functions such as `UpdateAsync()` and `IncrementAsync()`. As you work on larger experiences, particularly if you're dealing with Robux, it's strongly recommended that you use `UpdateAsync()`. It's a little bit more work, but it adds a layer of protection to your data. Learn more about these additional methods on the Developer Hub.

Q. How do I know what a returned error means?

A. You can search the Roblox Developer hub to find a list of common errors, as well limits for how often requests can be made: https://developer.roblox.com/articles/Datastore-Errors.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. What's the pattern for retrieving a Data Store (not just a Data Store key)?
2. What does the p in `pcall()` stand for?
3. When should you use a `pcall()`?
4. Data Stores take and store two pieces of information; what are they?
5. If there's a chance more than one server might be updating a Data Store at a time, should you use `SetAsync()` or `UpdateAsync()`?

Answers

1. `local dataStoreName= DataStoreService:GetDataStore("DataStoreName")`
2. Protected
3. You should use a `pcall()` *every time* you are reading or updating information from a Data Store.
4. A key and a value
5. `UpdateAsync()`

Exercise

Using the information in this hour, you should be able to award people five pieces of gold every time they log in. You can display people's gold on a leaderboard, but for the sake of this exercise, you're only being asked to print out how much gold a person has after it's been updated.

Tips

- Make sure you use the player's ID rather than their name.
- Don't forget to check success while reading and updating the Data Store.

HOUR 18
Creating a Game Loop

What You'll Learn in This Hour:

- How to set up a simple game loop
- How to use bindable events
- Why you should practice organization of code and assets

This hour introduces the concept of a *game loop*, or the pattern of actions that take place in a game. You'll learn how to set up a simple round-based game where players are transported to an arena and back again after a certain amount of time. To do this, you need all of the skills you've learned so far, and add one new skill: using BindableEvents.

The real lesson of this hour is to think about how to organize your world and the scripts within it. The main project focuses on best practices for organizing the workspace as well as your code.

Setting Up Game Loops

Game loops are the patterns of actions that people do within your Roblox experience. There's an infinite variety of loops the people in your worlds might go through. Here are a few examples:

- In harvesting simulators, one pattern is to harvest items, sell items, buy a bigger backpack or a faster shovel, and then harvest even more items.
- In a competitive experience, contestants might be taken from a lobby and then teleported to an arena to fight it out in a 15-minute match. At the end of the match, everyone is returned to the lobby to prepare for the next match.
- In an exploration world, individuals might go through cycles of cooking, mining, and hunting to improve their equipment and skills.
- In an educational experience, students might perform a virtual dissection, record their findings, and then move on to another organism to compare the differences.

The code you create needs to be able to facilitate the loops that people naturally want to cycle through, giving them periods of both excitement and rest. The loop you'll go through in this hour is a simple round-based game. Participants will be teleported from the lobby to the arena. Once in the arena, they have a certain amount of time to complete a moon-themed obby before they're transported back to the lobby.

This game loop is one that can be expanded upon if you later want to introduce elements like competition between players, completing puzzles, or collecting items.

Working with BindableEvents

This project needs a number of things to happen at various phases of the game loop. To know when to make these things happen, you use BindableEvents, which are similar to RemoteEvents. What makes them different is that they have the ability to communicate server to server or client to client, whereas RemoteEvents send signals across the client-server divide.

BindableEvents for server use should be placed within ServerStorage. Within ServerStorage, it's best practice to create folders for different types of objects to keep them organized. Figure 18.1 shows two BindableEvents in a folder and a second folder for ModuleScripts.

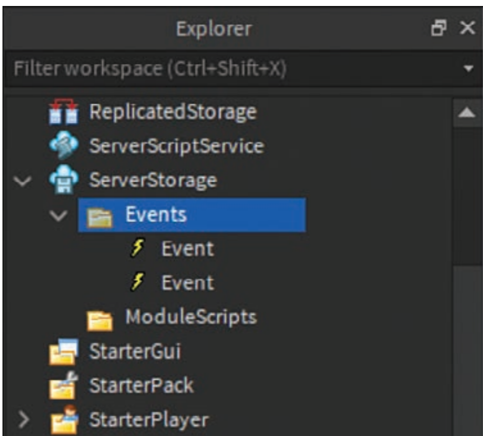


FIGURE 18.1
Within ServerStorage, use folders to organize different types of objects.

BindableEvents are fired using `EventName:Fire()`.

The event fired by BindableEvents is actually named Event. Functions can then be connected to Event like normal:

```
EventName.Event:Connect(functionName)
```

TRY IT YOURSELF ▼

Make a Simplified Game Loop

For the rest of this hour, you'll be working on a moon obby challenge while focusing on cleanly organizing your scripts and assets. The first step you'll take is to set up two distinct areas: a lobby and the arena. You can make them as detailed or functional as you would like. Figure 18.2 shows an elaborate lobby and arena; simpler versions are shown on the right.

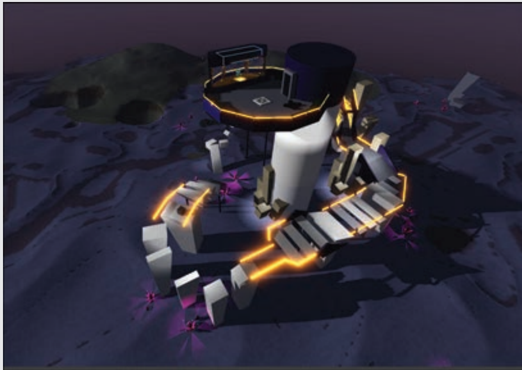
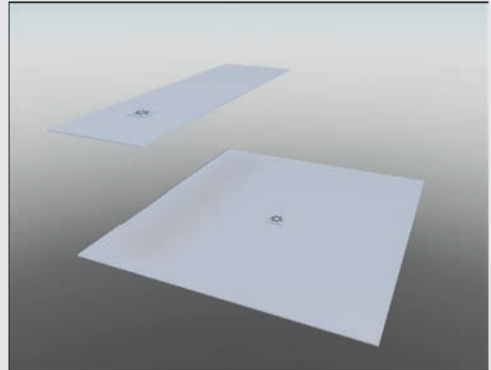



FIGURE 18.2
On the left, a lobby sits on a tower above an arena. On the right, simple parts are used to mark the lobby and arena.

After that, you'll set up the events to mark the beginning and end of each round, as well as set up the code for your simplified game loop.

Set Up

A big focus of this chapter is organization, which includes organizing the assets for your world. You'll keep all of the assets for the lobby and the arena in their own folders, and they'll have their own spawn locations.

1. Create the two areas you want to use for your world, such as those shown in Figure 18.2.
2. Separate all the elements for the two areas into unique folders (see Figure 18.3).

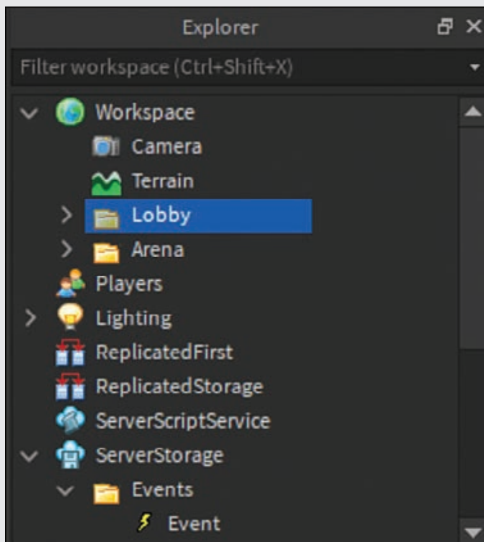


FIGURE 18.3
All of the assets for the lobby and arena should be separated into two folders.

3. In the Lobby folder, add a spawn location named `StartSpawn`. Then add a second spawn-Location in the Arena folder (see Figure 18.4).

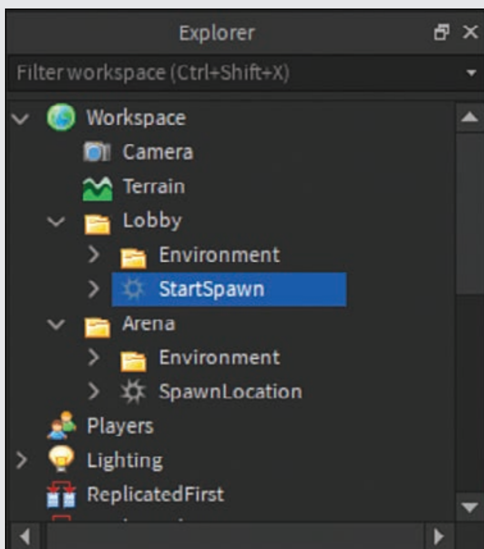


FIGURE 18.4
The SpawnLocations will be used for moving players back and forth.

TIP

Use Additional Folders as Needed

In Figure 18.4, you can see an additional folder named Environment; all of the environment parts have been placed inside. You don't need to do this, but it is helpful.

4. Within ServerStorage, add a folder named **`Events`**.

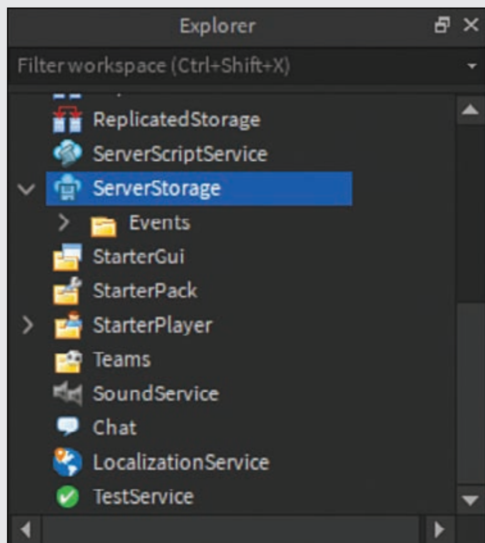


FIGURE 18.5
A new folder named Events is within ServerStorage.

5. Inside of the Events folder, add two new BindableEvents. Name one of the events **`RoundStart`** and the other **`RoundEnd`** (see Figure 18.6).

TIP

Theme Your World

If you would like to create a moonlike setting, you can change the gravity of the experience within Game Settings > World. Otherwise, you can stay focused on the code if you don't want to create an actual moon obby.

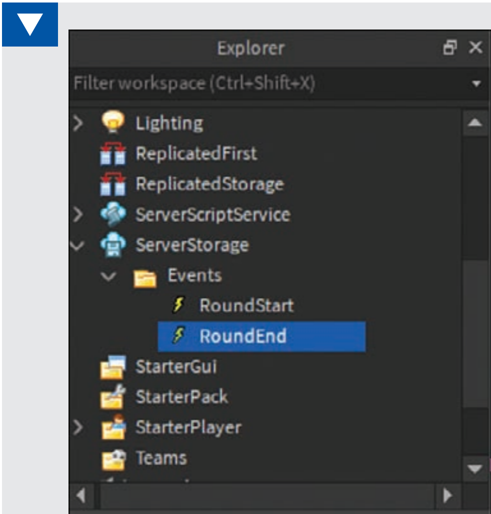


FIGURE 18.6
Within the Events folder, add two new BindableEvents.

RoundSettings ModuleScript

The basic settings that control how long each round lasts and how many people are required before starting will be pulled out into its own ModuleScript. This makes updating often-changed settings easier for you and easier for anyone who may end up working with you down the line.

1. In ServerStorage, add a new folder named **`ModuleScripts`**.
2. Within that folder, create a new ModuleScript named **`RoundSettings`** (see Figure 18.7).

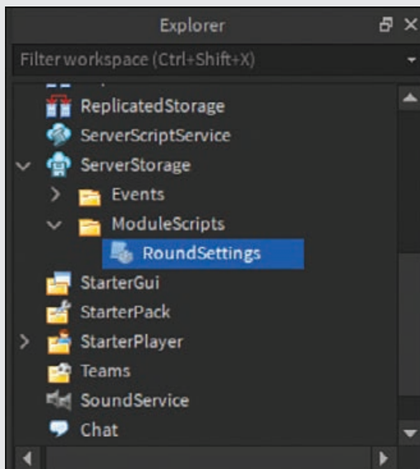


FIGURE 18.7
The ModuleScript folder and RoundSettings module have been created.

3. Insert values for how long each round should last, the amount of time the players spend in the obby, and the minimum number of people needed to start the round. Don't forget to rename the table:

```
local RoundSettings = {}

-- Game Variables
RoundSettings.intermissionDuration = 5
RoundSettings.roundDuration = 15
RoundSettings.minimumPeople = 1

return RoundSettings
```

RoundManager

The loop will run within a server-side script. As it runs, it'll fire the events at the appropriate time. A separate ModuleScript will then be listening for those events.

1. In the ModuleScript folder, add a new ModuleScript named **`PlayerManager`** (see Figure 18.8).

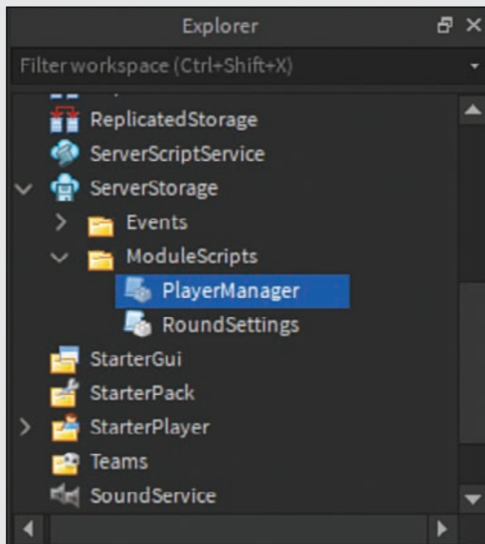


FIGURE 18.8
ModuleScripts includes a new folder named PlayerManager.

2. In ServerScriptService, add a new regular script named RoundManager (see Figure 18.9).

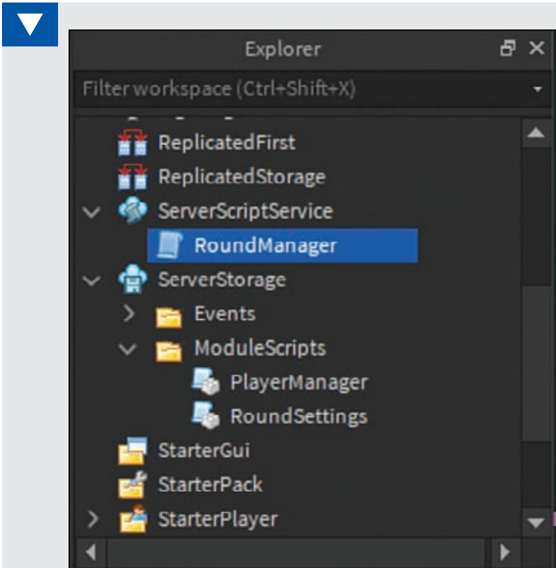


FIGURE 18.9
You should have three scripts altogether.

3. Get your services:

```
-- Services
local ServerStorage = game:GetService("ServerStorage")
local Players = game:GetService("Players")
```

4. Set up references for the ModuleScripts folder and the two ModuleScripts you'll be using:

```
-- Module Scripts
local moduleScripts = ServerStorage.ModuleScripts
local playerManager = require(moduleScripts.PlayerManager)
local roundSettings = require(moduleScripts.RoundSettings)
```

5. Get the two BindableEvents: `RoundStart` and `RoundEnd`:

```
-- Events
local events = ServerStorage.Events
local roundStart = events.RoundStart
local roundEnd = events.RoundEnd
```

6. Create a `while-true-do` loop. Within it, use the settings from the ModuleScript to wait for the right number of people to join the experience and then fire `RoundStart`:

```
-- Runs the game loop
while true do
    repeat
        wait(roundSettings.intermissionDuration)
```

```
        until Players.NumPlayers >= roundSettings.minimumPeople
        roundStart:Fire()
        wait(roundSettings.roundDuration)
        roundEnd:Fire()
    end
```

TIP

Repeat Until a Condition Is Met

Before now, we've often used a `while` loop, which runs until a condition becomes false. Here, we've used `repeat until`, which does the opposite; it runs until a condition becomes true. In this case, the condition is when the minimum number of players have joined.

7. Wait for the time specified in RoundSettings for the round duration and then fire RoundEnd. Here is the completed script:

```
-- Services
local ServerStorage = game:GetService("ServerStorage")
local Players = game:GetService("Players")

-- Module Scripts
local moduleScripts = ServerStorage.ModuleScripts
local playerManager = require(moduleScripts.PlayerManager)
local roundSettings = require(moduleScripts.RoundSettings)

-- Events
local events = ServerStorage.Events
local roundStart = events.RoundStart
local roundEnd = events.RoundEnd

while true do
    repeat
        wait(roundSettings.intermissionDuration)
    until Players.NumPlayers >= roundSettings.minimumPeople
    roundStart:Fire()
    wait(roundSettings.roundDuration)
    roundEnd:Fire()
end
```

This is the loop that will run over and over and over. You can now add functionality to the rounds by listening for the fired events.

PlayerManager

This is where the meat of your code will be. Everything that happens to the players as they enter and leave the round goes here. You can give people weapons, assign teams, record scores, or do pretty much anything else you can think of. For now, we're just going to transfer them to and from the obby.

1. Start with your services:

```
local PlayerManager = {}

-- Services
local Players = game:GetService("Players")
local ServerStorage = game:GetService("ServerStorage")

return PlayerManager
```

2. Set up variables for the lobby spawn, the arena map, and the arena spawn:

```
local PlayerManager = {}

-- Services
local Players = game:GetService("Players")
local ServerStorage = game:GetService("ServerStorage")

-- Variables
local lobbySpawn = workspace.Lobby.StartSpawn
local arenaMap = workspace.Arena
local arenaSpawn = arenaMap.SpawnLocation

return PlayerManager
```

3. Get the events like you did in the last script:

```
local PlayerManager = {}

-- Services
local Players = game:GetService("Players")
local ServerStorage = game:GetService("ServerStorage")

-- Variables
local lobbySpawn = workspace.Lobby.StartSpawn
local arenaMap = workspace.Arena
local arenaSpawn = arenaMap.SpawnLocation

local events = ServerStorage.Events
local roundEnd = events.RoundEnd
local roundStart = events.RoundStart


return PlayerManager
```

4. Start off with what should happen when the player first joins the experience, before they enter the arena. Here, we're just going to spawn them in the arena:

```
local PlayerManager = {}
```

```
-- Previous code not shown

local function onPlayerJoin(player)
    player.RespawnLocation = lobbySpawn
end

return PlayerManager
```

TIP

Get Any Saved Data

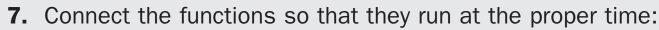
If you decide to add any saved data—such as skill level, appearance upgrades, or total points—this is where you can check for that and update a leaderboard.

5. Say what you want to happen at the beginning of the round. Here, we're going through the list of players and reloading their character at the arena spawn location:

```
local PlayerManager = {}

-- Previous code not shown

local function onRoundStart()
    for _, player in ipairs(Players:GetPlayers()) do
        player.RespawnLocation = arenaSpawn
        player:LoadCharacter()
    end
end

return PlayerManager
```

6. Create a function to run at the end of the round:

```
local PlayerManager = {}
-- Previous code not shown

local function onRoundEnd()
    for _, player in ipairs(Players:GetPlayers()) do
        player.RespawnLocation = lobbySpawn
        player:LoadCharacter()
    end
end

return PlayerManager
```



7. Connect the functions so that they run at the proper time:

```
local PlayerManager = {}

-- Services
local Players = game:GetService("Players")
local ServerStorage = game:GetService("ServerStorage")

-- Variables
local lobbySpawn = workspace.Lobby.StartSpawn
local arenaMap = workspace.Arena
local arenaSpawn = arenaMap.SpawnLocation

local events = ServerStorage.Events
local roundEnd = events.RoundEnd
local roundStart = events.RoundStart

local function onPlayerJoin(player)
    player.RespawnLocation = lobbySpawn
end

local function onRoundStart()
    for _, player in ipairs(Players:GetPlayers()) do
        player.RespawnLocation = arenaSpawn
        player:LoadCharacter()
    end
end

local function onRoundEnd()
    for _, player in ipairs(Players:GetPlayers()) do
        player.RespawnLocation = lobbySpawn
        player:LoadCharacter()
    end
end

Players.PlayerAdded:Connect(onPlayerJoin)
roundStart.Event:Connect(onRoundStart)
roundEnd.Event:Connect(onRoundEnd)

return PlayerManager
```

Summary

A game loop, or an action loop, is the pattern of activities that people take within your experience. The code you write needs to support those actions. In this hour, you created the loop using a literal coded `while` loop. In other experiences, the loop might still be driven by events such as harvesting, buying, and selling, but those may not be so literally circular. Once you have your basic loop designed, you can continue to add functionality, such as server announcements, team assignments, the ability to add random maps, and updating saved data.

BindableEvents are quite often used with game loops because they allow signals to be sent server to server or client to client.

This hour also talked a lot about keeping the objects in your projects organized, which is just as important as writing clean code. Use folders to organize the scripts, models, events, and everything else in your world.

Q&A

Q. Why did you use `LoadCharacter()` instead of updating the HumanoidRootPart's CFrame?

A. If all you want to do is simply move a player from one location to another, updating their CFrame position is a quick way to do it. However, there are some benefits that can be taken advantage of by forcing a character to reload. SpawnLocations are capable of providing temporary force fields as well as being used for team assignments and checkpoints.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. True or false: BindableEvents can send signals across the server-client divide.
2. A `repeat-until` loop does what?
3. Where should BindableEvents be stored?
4. If you were to add functionality such as giving the players superpowers, how would you do that?

Answers

1. False. If a signal needs to be sent between the server and client, you should use RemoteEvents.
2. A `repeat-until` loop repeats until a condition is true, which is the opposite of what a `while-do` loop does.
3. If BindableEvents will be used by the server, they should be stored within their own folder in ServerStorage. If BindableEvents are used by the client, then they are stored within ReplicatedStorage.
4. You can add functionality within `roundStart()` and remove it within `roundEnd()`. Or, if it's a larger chunk of code, you may want a separate ModuleScript to also be required within RoundManager.

Exercise

It's always good to keep players informed of what's about to happen to them. Create a new ModuleScript designed to announce the beginning and end of every match. For now, it's fine to just practice with BindableEvents and simply print "Round Starting" and "Round Over" at the beginning and end of each round. In an actual experience, you would want to set up the UI to make the announcement to the players.

Tips

- Create a separate module named Announcements.
- Within the module, create separate functions for the announcements at the beginning and end of the match.
- Add the module to those required in RoundManager.
- Use `print` statements to check everything, and then if you want to, move to creating more finalized UI code.

HOUR 19
Monetization: One-Time Purchases

What You'll Learn in This Hour:

- ▶ How to allow one-time purchases with Robux
- ▶ How to prompt the user for a purchase
- ▶ How to confirm whether they have made the purchase previously
- ▶ How to update avatar appearance on join

This hour deals with how to allow for users to buy items in your experiences using Robux. You can use your earned Robux in other games or to purchase catalog items, or you can eventually cash out for real-world money using the Developer Exchange Program.

To cash out, you must have an active Roblox Premium membership, be at least 13 years old, and have acquired at least 100,000 Earned Robux. To see the full set of guidelines, visit the Developer Exchange FAQs.

As you go through this hour, you may want to test the functionality of the items you're selling, which you have to have Robux to do. If you don't have any Robux at this time or don't want to spend them, you can safely continue to the next hour without missing any new coding concepts.

Adding Passes to Your Experience

There are several ways to monetize your experience on Roblox. One of those ways is *passes*. Passes let you create special items that can only be purchased once per person using Robux. Here are some examples of when you might want to use a pass:

- ▶ To give people the ability to visit new areas of your experience.
- ▶ To unlock avatar items people can wear.
- ▶ To offer people cosmetics such as sparkles, trails, and weapon skins.

TIP

Fun Comes First

The most profitable experiences are the ones in which people have the most fun. If people have been having fun in your experience for a while, they're more likely to spend Robux than if you make them pay for things upfront.

To make a new pass, you first need to set up some information on the Roblox site about the pass, and you need an image for the pass itself:

1. Make sure your experience is published and then go to https://www.roblox.com/.
2. Click Create (see Figure 19.1).

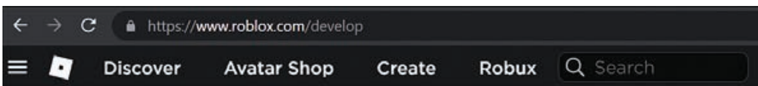


FIGURE 19.1
Click Create in the top navigation.

3. Find the experience you want to make a pass for.
4. Select Create Pass from the Settings drop-down menu on the right-hand side (see Figure 19.2).

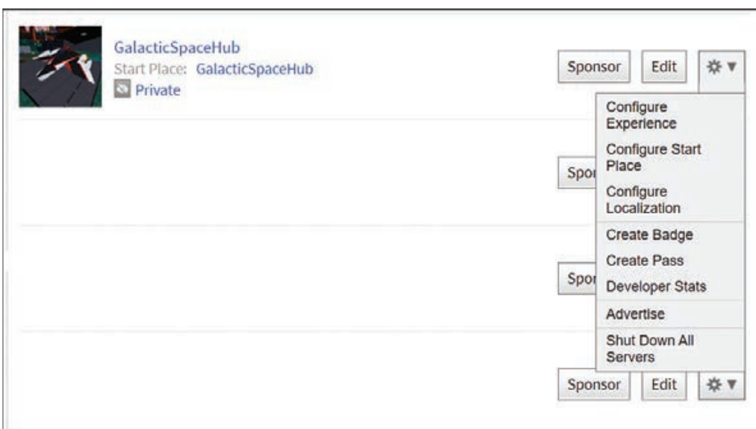


FIGURE 19.2
Select Create Pass.

5. All passes require an image, name, and description. Provide all three and then click the Preview button (see Figure 19.3).

Create a Pass

Target Experience: PassPractice

Find your image: Choose File Hat.png

Pass Name: Party Crown

Description: Buy this pass to wear a cool party crown

Preview

FIGURE 19.3
Upload an image for the pass, give the pass a name, and type a description.

6. On the next screen, click Verify Upload (see Figure 19.4) to create the pass and send it through moderation.

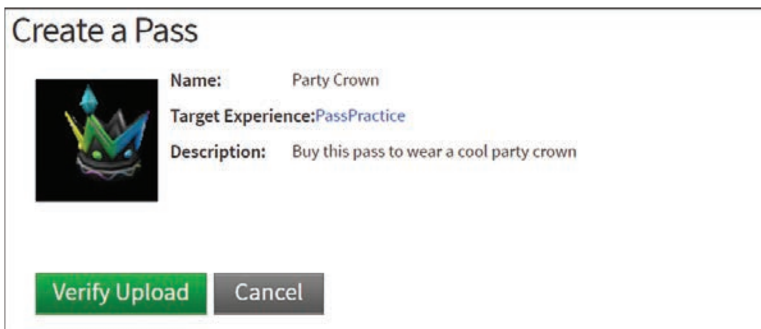


FIGURE 19.4
Make sure everything looks correct before you click Verify Upload.

TIP

You Can Update the Pass Later

Changes can be made to the pass by selecting the pass and then right-clicking and selecting Configure.

Configuring the Pass

Once you've created the game pass, it will appear slightly further down on the same page you're on beneath the create section. The final step is configuring the pass so players can buy it:

1. Select Configure from the Settings right-side drop-down menu for the new pass (see Figure 19.5).

FIGURE 19.5
Configure the new pass.

2. On the configuration page, click the Sales tab (see Figure 19.6).

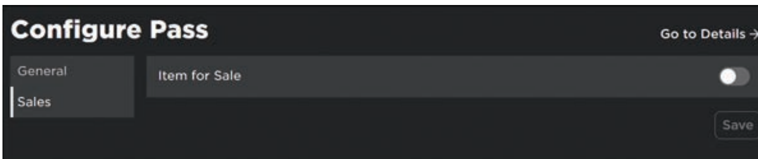


FIGURE 19.6
Click the Sales tab.

3. Click the Item for Sale toggle switch to make the pass available to players. Then enter the price (in Robux) players will pay for the item (see Figure 19.7).

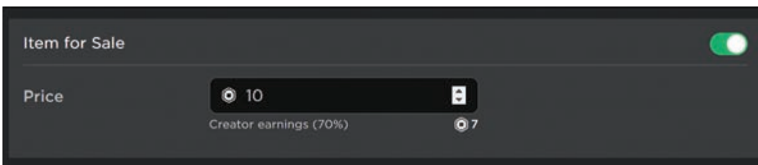


FIGURE 19.7
After you enter the price, the percentage of Robux you'll get from the sale is shown under the field.

TIP

Roblox Premium Members Get More

If you're a part of Roblox Premium, creator earnings are a greater percentage of the item cost. The percentage kept by Roblox keeps the lights on and the servers running.

4. Click the Save button to confirm your settings.
5. Take a moment to copy the ID number of the pass from the URL. You'll need this number in your code (see Figure 19.8).

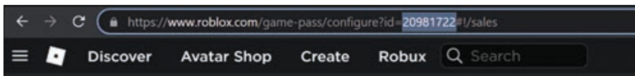


FIGURE 19.8
Make a note of the ID number for use in your code.

Prompting In-Game Purchases

Players can buy passes directly from your game's main page by clicking the Store tab and browsing the available items for purchase. Alternatively, you can call the MarketplaceService within an in-game shop using the following code:

```
MarketplaceService:PromptGamePassPurchase(player, gamePassID)
```

When you're checking to see if somebody already owns a pass, use this code:

```
UserOwnsGamePassAsync(player.UserId, gamePassID)
```

You should always wrap the check within a protected call like this:

```
local success, message = pcall(function()
    hasPass = MarketplaceService:UserOwnsGamePassAsync(player.UserId, gamePassID)
    end)
```

▼ TRY IT YOURSELF

Sell Crowns to the Crowd

Create a button and use a local script to create a pass that allows players in your experience to wear a cool neon party crown like the one shown in Figure 19.9.

FIGURE 19.9
Let people wear this cool party crown if they buy your pass.

Set Up the Pass

First, you need to set up the pass as shown earlier; then you delete the pass from your inventory so you can test purchasing it.

1. Follow the steps in the first part of the hour to create a pass. Remember to configure the pass so it's available to be purchased. You'll need to test purchasing the pass, so you may want to make the price something like 5 or 10 Robux.

2. On the pass page, delete the pass from your inventory (see Figure 19.10). If you closed the page, you can get back to the pass page by going back to the Create Pass drop-down menu and then clicking on the pass.

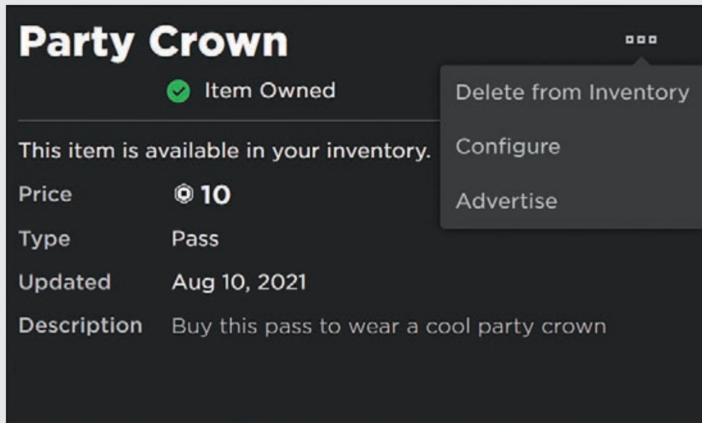


FIGURE 19.10
Delete the pass from your inventory so you can test what it's like to purchase it.

3. In StarterGui, add a new ScreenGui and Button named **BuyHat**. You can use either a TextButton or ImageButton.

Prompting the Purchase

This section is where you set up the script for the purchase:

1. In the ScreenGui, add a new LocalScript.
2. At the top of the script, get MarketplaceService, Players, and the local player:

```
local MarketplaceService = game:GetService("MarketplaceService")
local Players = game:GetService("Players")
local player = Players.LocalPlayer
```

3. Create a function named **promptPurchase** connected to the button:

```
-- Previous Code
local screenGui = script.Parent
local button = screenGui:FindFirstChild("BuyHat")

local function promptPurchase()

end
button.Activated:Connect(promptPurchase)
```

4. Set up a variable with your pass ID that you copied from the URL. Inside the function, set up a Boolean variable named **hasPass**. Set `hasPass` to false:

```
-- Previous code
local screenGui = script.Parent
local button = screenGui:FindFirstChild("BuyHat")
```

```
local gamePassID = 0000000  -- Change this to your game pass ID

local function promptPurchase()
    local hasPass = false
end
button.Activated:Connect(promptPurchase)
```

5. Inside `promptPurchase`, use protected calls to check whether the player has the pass:

```
local function promptPurchase()
    local hasPass = false
    local success, message = pcall(function()
        hasPass =
        MarketplaceService:UserOwnsGamePassAsync (player.UserId, gamePassID)
    end)

    if not success then
        warn("Error while checking if player has pass: " .. tostring(message))
        return
    end
end
```

6. If the player doesn't have a pass, trigger the purchase prompt. The following is the completed script:

```
local MarketplaceService = game:GetService("MarketplaceService")
local Players = game:GetService("Players")
local player = Players.LocalPlayer

local screenGui = script.Parent
local button = screenGui:FindFirstChild("BuyHat")

local gamePassID = 0000000  -- Change this to your game pass ID

local function promptPurchase()

    local hasPass = false

    local success, message = pcall(function()
        hasPass =
        MarketplaceService:UserOwnsGamePassAsync (player.UserId, gamePassID)
    end)

    if not success then
        warn("Error while checking if player has pass: " .. tostring(message))
        return
    end
    if hasPass then
            button.Text = "Already Owned"
```

```
        else
            -- Player does NOT own the game pass; prompt them to purchase
            MarketplaceService:PromptGamePassPurchase(player, gamePassID)
        end
    end
    button.Activated:Connect(promptPurchase)
```

Testing

Testing properly requires a live server, but first you can do some quick spot checks in Studio to see if you're on the right track:

1. Quickly test your code. You should see confirmation that you either own the pass or an error that purchases aren't allowed in this environment (see Figure 19.11). The error is a good sign because passes have to be bought in a live server. If you see this, go on to the next step. Otherwise, check your code and make sure you don't already have the pass.

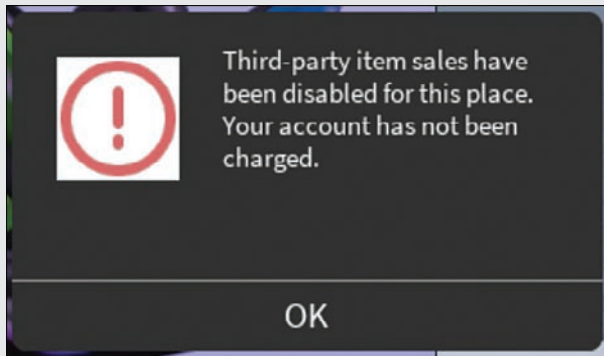


FIGURE 19.11
If your code is correct, this error should appear while testing in Studio.

2. Publish to Roblox. You'll need the most up-to-date version of your place.
3. Test the code from the live server. The easiest way is to go back to the Create tab and click on the starting place of the experience you want to test. That'll take you to the game page.
4. Within the place, test your button. You should see a prompt to purchase the pass like the one shown in Figure 19.12.

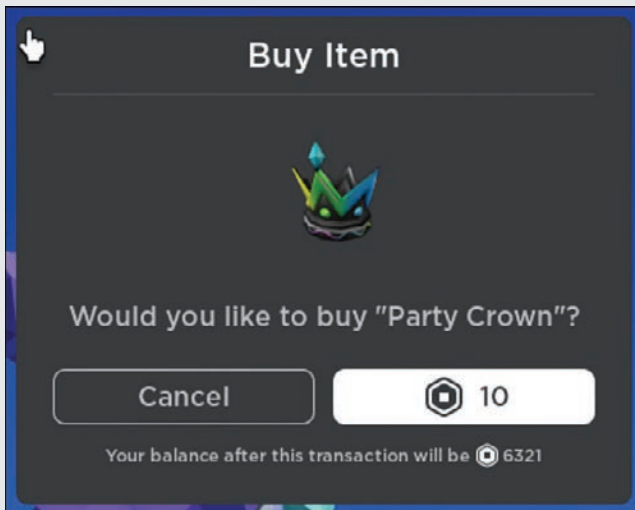


FIGURE 19.12
In the live environment, a prompt to purchase the pass displays.

TIP

Test the Live Server Often

It's always best to test your code on a live server, particularly if you have GUI elements. Different screen sizes and types can affect how people view your experience. If you can, test a variety of devices, such as tablets, phones, PCs, and Macs.

Give Them Their Hat

Once somebody buys the pass, they expect it to work each and every time they join the game. This section shows you how to check whether somebody has a pass. Then you'll override the user's current appearance once their character has been added to workspace.

1. In ServerScriptService, add a new script.
2. Set up the services and your pass ID. Then connect a function to `PlayerAdded`:

```
local MarketplaceService = game:GetService("MarketplaceService")
local Players = game:GetService("Players")

local gamePassID = 0000000  -- Change this to your game pass ID

local function onPlayerAdded(player)

end
Players.PlayerAdded:Connect(onPlayerAdded)
```

3. Inside the function, check for the pass as shown:

```
-- Previous code
local function onPlayerAdded(player)
    local hasPass = false

    local success, message = pcall(function()
        hasPass =
        MarketplaceService:UserOwnsGamePassAsync (player.UserId, gamePassID)
   end)

    if not success then
        warn("Error while checking if player has pass: " .. tostring(message))
        return
    end
end
```

4. If the player has the pass, you trigger whatever functionality the pass gives. Check for the pass as shown here:

```
-- Previous code
local function onPlayerAdded(player)
    local hasPass = false
    local success, message = pcall(function()
        hasPass = MarketplaceService:UserOwnsGamePassAsync
            (player.UserId, gamePassID)
    end)
    if not success then
        warn("Error while checking if player has pass: " .. tostring(message))
        return
    end

    if hasPass == true then
        print(player.Name .. " owns the game pass with ID " .. gamePassID)
       -- Code for pass functionality goes here.
    end
end
```

5. For this particular pass, we want to override the user's appearance. Create a new function named **`onCharacterAdded`** and call it if the user has the pass:

```
local MarketplaceService = game:GetService("MarketplaceService")
local Players = game:GetService("Players")

local gamePassID = 0000000  -- Change this to your game pass ID

local function onCharacterAdded(character)

end
```

```
local function onPlayerAdded(player)
    local hasPass = false
    local success, message = pcall(function()
        hasPass =
        MarketplaceService:UserOwnsGamePassAsync (player.UserId, gamePassID)
    end)
    if not success then
        warn("Error while checking if player has pass: " .. tostring(message))
        return
    end
    if hasPass == true then
        print(player.Name .. " owns the game pass with ID " .. gamePassID)
        player.CharacterAdded:Connect(onCharacterAdded)
    end
end
Players.PlayerAdded:Connect(onPlayerAdded)
```

6. Set up the hat ID you want to use:

```
local MarketplaceService = game:GetService("MarketplaceService")
local Players = game:GetService("Players")

local gamePassID = 0000000  -- Change this to your game pass ID
local HAT_ID = 156486131
```

7. Finally, inside `onCharacterAdded`, get the current humanoid description and wait for the character to be added to the workspace. Then use `ApplyDescription()` to add the hat, as shown here:

```
local MarketplaceService = game:GetService("MarketplaceService")
local Players = game:GetService("Players")

local gamePassID = 0000000  -- Change this to your game pass ID
local HAT_ID = 156486131
local function onCharacterAdded(character)
    local humanoid = character:WaitForChild("Humanoid")
    local description = humanoid:GetAppliedDescription()
    description.HatAccessory = HAT_ID

    while not character.Parent do
        character.AncestryChanged:Wait()
    end
        humanoid:ApplyDescription(description)
end

local function onPlayerAdded(player)
    local hasPass = false
```

```
        local success, message = pcall(function()
            hasPass =
            MarketplaceService:UserOwnsGamePassAsync (player.UserId, gamePassID)
        end)

        if not success then
            warn("Error while checking if player has pass: " .. tostring(message))
            return
        end

        if hasPass == true then
            print(player.Name .. " owns the game pass with ID " .. gamePassID)
            player.CharacterAdded:Connect(onCharacterAdded)
        end
    end
    Players.PlayerAdded:Connect(onPlayerAdded)
```

TIP

Using Descriptions

An alternative way to add a hat is to simply parent it to the user's head. However, descriptions are more reliable. Descriptions can also be used to update other accessories, such as hair. You can find a full list of description uses on the Developer Hub.

8. When you test in Studio, the print confirmation should appear in Output. Publish to Roblox and then check on the server to verify everything actually works as intended.

Summary

Above all else, always concentrate on making your experience somewhere people want to spend time before concentrating on getting your community to spend money.

To create one-time purchasable items, create a pass for the experience on www.roblox.com. Then, within the experience, set up the functionality for the pass.

Purchasing and checking for passes are handled with MarketplaceService. You can use MarketplaceService to prompt users to purchase a pass with `PromptGamePassPurchase()` and check whether they already have the pass with `UserOwnsGamePassAsync(player.UserId, gamePassID)`. Always wrap the check in a `pcall()`.

The pass demonstrated in this hour also took advantage of `humanoid:GetAppliedDescription()` to see what items someone is currently wearing and then updated their appearance using `humanoid:ApplyDescription(description)`.

Q&A

Q. What if you want to create something that can be purchased over and over again, like in-game currencies or power-ups?

A. To create items that can be purchased repeatedly, you want to create Developer Products. These work similarly to passes. If you search Developer Products on developer.roblox.com, you'll find sample code. With the knowledge you have from this hour, you should be able to figure out how to customize the code.

Q. How do I come up with ideas for pass and developer products?

A. The number one most important thing you can do is figure out what would make people want to stick around in your experience longer. If people are having fun, they'll spend more. One thing you can do is offer special skins, rare items, or consumables like health potions. You can find great workshops about good monetization practices and how to create more engaging experiences by following Roblox Developer Relations and their Level Up series on YouTube.

Q. Is there any other way to make money on Roblox?

A. In addition to creating developer products and passes, you can also receive engagement payouts. If users who are members of Roblox Premium spend a lot of time within your experience, you can earn Robux. The more time they spend, the more you earn. There's also opportunities to create and sell avatar items, t-shirts, and plug-ins.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. What service allows you to sell items in your experience?
2. True or false: If you want to create a pass to sell, you need to include an image.
3. You always wrap a check for passes in ________.
4. Should confirmation that the user owns a pass be done on the server or client side?
5. If you want your experience to do well and make Robux, what should you concentrate on before anything else?
6. How old do you need to be to take advantage of the Developer Exchange?

Answers

1. MarketplaceService
2. True
3. Protected calls, or `pcall()`s
4. On the server side. You never want something that important happening on the client side.
5. Fun. If your experience isn't fun, people won't spend Robux in it.
6. At least 13.

Exercise

Come up with an idea for a cosmetic change that can take place if a user has the correct pass. In this particular case, the code will be very unique to you, but you should be able to make it work if you follow the basic pattern already shown. The cosmetic change can be anything from giving the player sparkles to giving them a permanent speed boost.

Whatever pass you decide to create, it should never give purchasers an unfair advantage within your experience. That's not fun for anyone. Your community will complain, and in the long run, you will most likely make fewer Robux than if you keep the playing field even.

This page intentionally left blank

HOUR 20

Object-Oriented Programming

What You'll Learn in This Hour:

- What object-oriented programming means
- How to define a custom class
- How to add properties and methods to a class
- How to create an instance of that class

This hour explains how to create custom instances using a concept called *object-oriented programming*, which is more commonly referred to as just *OOP*. As you practice object-oriented programming, you'll begin to think about what objects in your experience have in common and how you can begin to group them and create new categories of objects.

What Is OOP?

The core of OOP are the concepts of objects and classes. An object represents an individual thing in your world, such as a house, a car, a tree. Indeed, everything in the Explorer is an object. Parts, models, particle emitters, proximity prompts—all of these are objects.

A *class* describes what an object is and does. For example, a Part and SurfaceLight are very different things even though they are both objects. Their classes are what make them different. A Part is one type of class, whereas a Light is another.

Organizing Code and Projects

OOP allows you to break problems down into smaller parts. As you plan your experience, you can think about the different types of things your world needs. For example, you might have cars, NPCs, playable classes, and different weapon types.

Once you know all the different types of things you need, you can begin thinking about how to code them in a way that allows you to reuse code to create different versions of the same things. For example, one car might be red with a racing stripe, whereas another is yellow. What you

don't want is to have to create two different scripts just because the cars need to be two different colors. With OOP, you just have to make one class, car, and then have color be one of the modifiable properties.

Making a New Class

Some programming languages use special keywords to create classes. In Lua, you simply use a table with a couple of modifications. The following steps walk you through the general process of making a new class:

1. To create a class, type the following:

```
local NameOfClass = {}
NameOfClass.__index = NameOfClass
```

TIP

Use Two Underscores Before `index`

There are actually two underscores before `index`. It can easily be mistaken for just one. `__index` should always be set to the name of the class.

2. A class on its own doesn't do much, though. You need to add a function to the class that describes how to create a new object of that class. These functions are called *constructors* because they make things. Get it?

```
function NameOfClass.new()

end
```

3. Inside the constructor, create a new table and return it at the end of the function. This table is the object that gets made whenever a new class instance is asked for:

```
function NameOfClass.new()
    local self = {}

    return self
end
```

TIP

`self` as a Naming Convention

`self` is a commonly recognized way of referring to the object being created inside of the class. It's such a common convention that Studio bolds `self` to make it easier to see within a class, even though it's not normally a keyword in Lua.

4. Use `setmetatable()` to insert the object `self` into the original table:

```
function NameOfClass.new()
    local self = {}
    setmetatable(self, NameOfClass)

    return self
end
```

5. Finally, you can call the function to create a new class instance:

```
local NameOfClass = {}
NameOfClass.__index = NameOfClass

function NameOfClass.new()
    local self = {}
    setmetatable(self, NameOfClass)

    return self
end

local newObject = NameOfClass.new()
```

Adding Class Properties

Like other Roblox Instances, your custom classes can also have properties like color, size, and scale. You can add a new property into the constructor function of the class. As for the value of the property, you can add that yourself by directly assigning a value, or you can pass the value in as a parameter of the constructor:

```
local NameOfClass = {}
NameOfClass.__index = NameOfClass

function NameOfClass.new(parameterProperty)
    local self = {}
    setmetatable(self, NameOfClass)

    self.defaultProperty = "Default Value"
    self.parameterProperty = parameterProperty

    return self
end

local newObject = NameOfClass.new()
```

▼ TRY IT YOURSELF

Create a Car Class

Create a car class that has one property for its color and another property for the number of wheels on the car. The color should be passed in through the constructor, but the number of wheels should be hard coded to 4.

1. Create a class named `Car` and its constructor:

```
local Car = {}
Car.__index = Car

function Car.new()
    local self = {}
    setmetatable(self, Car)

    return self
end
```

2. Hard code the number of wheels:

```
local Car = {}
Car.__index = Car

function Car.new()
    local self = {}
    setmetatable(self, Car)

    self.numberOfWheels = 4

    return self
end
```

3. Add the parameter for the color of the car. Remember to add the parameter to the constructor declaration as well:

```
local Car = {}
Car.__index = Car

function Car.new(color)
    local self = {}
    setmetatable(self, Car)

    self.numberOfWheels = 4
    self.color = color

    return self
end
```

4. Test your code by creating a new instance of the car and try printing out its properties:

```
local Car = {}
Car.__index = Car

function Car.new(color)
    local self = {}
    setmetatable(self, Car)

    self.numberOfWheels = 4
    self.color = color

    return self
end

local redCar = Car.new("red")
print(redCar.numberOfWheels) -- Prints "4"
print(redCar.color) -- Prints "red"
```

Using Class Functions

Custom classes can also have functions. Unlike properties, functions are declared outside the constructor:

```
local NameOfClass= {}
NameOfClass.__index = NameOfClass

function NameOfClass.new()
    local self = {}
    setmetatable(self, NameOfClass)
    return self
end

-- Class function is declared outside of constructor
function NameOfClass:nameOfFunction()

end
```

You'll often find that you will want to reference the object in class functions, particularly to access that object's properties. You can use `self` in functions that represent the object:

```
function NameOfClass:nameOfFunction()
    local variable = self.nameOfProperty
end
```

TRY IT YOURSELF

Create Your Own Pet

What's better than a pet that follows you around when you pat it on the head? For this exercise, create your own custom pet class that follows players for a short time after they interact with it.

Set Up

First, you need to have a model to use as the pet. You could use a fancy pet model if you have one. Otherwise, just follow these steps to create a square stand-in:

1. Insert a model into the workspace; rename the model something like `Dog`. If using just a square part, use the key combination Ctrl+G/Cmd+G to turn it into a model and then rename it.
2. Inside the model, insert a new part. Rename the part **`HumanoidRootPart`**.

TIP

`MoveTo()` Needs a `HumanoidRootPart`

Later in this script, the function `MoveTo()` will be used to make the pet move. In order for it to work, the main part must be named exactly `HumanoidRootPart`. If it's named anything else or capitalized wrong, the part won't be able to animate properly.

3. Select the model, and set PrimaryPart to `HumanoidRootPart`.
4. With the model still selected, insert a humanoid.
5. Move the pet to ServerStorage.

Create a Pet Class

Now, set up the class to create the pet, and then in the next section, you'll add a function to make the pet follow anyone who pets it:

1. In ServerScriptService, add a new script.
2. Reference ServerStorage and create a constant for how long the pet will follow a player after someone has patted its head:

```
local ServerStorage = game:GetService("ServerStorage")

local FOLLOW_DURATION = 5
```

3. Create a new pet class and its constructor. Include a parameter to allow you to pass in what model to use for the pet:

```
local Pet = {}
Pet.__index = Pet

function Pet.new(model)
    local self = {}
```

```
    setmetatable(self, Pet)

    return self
end
```

4. Assign the passed-in parameter to the class's model and parent it to the workspace, as follows:

```
local Pet = {}
Pet.__index = Pet

function Pet.new(model)
    local self = {}
    setmetatable(self, Pet)

    self._model = model
    self._model.Parent = workspace


    return self
end
```

TIP

Pay Attention to the Naming Convention

Notice that inside of the constructor, names are preceded by a single underscore.

5. Inside the constructor, create a new ProximityPrompt. Update the prompt's `ObjectText` and `ActionText` as shown in the following code, and then parent the prompt to the pet:

```
local Pet = {}
Pet.__index = Pet

function Pet.new(model)
    local self = {}
    setmetatable(self, Pet)

    self._model = model
    self._model.Parent = workspace

    self._petPrompt = Instance.new("ProximityPrompt")
    self._petPrompt.ObjectText = "Pet"
    self._petPrompt.ActionText = "Give pets!"
    self._petPrompt.Parent = model.PrimaryPart

    return self
end
```

Add a Function to the Pet

This is where you'll create the following function. `MoveTo()` updates the location of the pet every 0.25 seconds to where the player who patted it is:

1. Add a new function to the pet class named **`getPets()`** with a parameter for the player:

```
-- Previous Code

function Pet:getPets(player)

end

Disable the prompt when it's been triggered.
function Pet:getPets(player)
    self._petPrompt.Enabled = false
end
```

2. Use a `for` loop to update the location of the pet every 0.25 seconds and then re-enable the prompt:

```
function Pet:getPets(player)
    self._petPrompt.Enabled = false

    for i = 0, FOLLOW_DURATION, 0.25 do
        local character = player.Character
        if character and character.PrimaryPart then
            self._model.Humanoid:MoveTo(character.PrimaryPart.Position)
        end
        wait(0.25)
    end
    self._petPrompt.Enabled = true

end
```

3. Return the `pet` class:

```
function Pet:getPets(player)
    self._petPrompt.Enabled = false
    for i = 0, FOLLOW_DURATION, 0.25 do
        local character = player.Character
        if character and character.PrimaryPart then
            self._model.Humanoid:MoveTo(character.PrimaryPart.Position)
        end
        wait(0.25)
    end
    self._petPrompt.Enabled = true

    return Pet
end
```

4. Go back up to the `Pet` constructor and use an anonymous function to call `getPets` when the prompt has been triggered:

```
local Pet = {}
Pet.__index = Pet

function Pet.new(model)
    local self = {}
    setmetatable(self, Pet)

    self._model = model
    self._model.Parent = workspace

    self._petPrompt = Instance.new("ProximityPrompt")
    self._petPrompt.ObjectText = "Pet"
    self._petPrompt.ActionText = "Give pets!"
    self._petPrompt.Parent = model.PrimaryPart
    self._petPrompt.Triggered:Connect(function (player)
        self:getPets(player)
    end)

    return self
end
```

5. Now, all that's left is to make a new instance of the `Pet` class and pass in the model of the pet you want it to use. Here is the completed script:

```
local ServerStorage = game:GetService("ServerStorage")

local FOLLOW_DURATION = 5

local Pet = {}
Pet.__index = Pet

function Pet.new(model)
    local self = {}
    setmetatable(self, Pet)

    self._model = model
    self._model.Parent = workspace

    self._petPrompt = Instance.new("ProximityPrompt")
    self._petPrompt.ObjectText = "Pet"
    self._petPrompt.ActionText = "Give pets!"
    self._petPrompt.Parent = model.PrimaryPart
    self._petPrompt.Triggered:Connect(function (player)
        self:getPets(player)
    end)
```

```
    return self
end

function Pet:getPets(player)
    self._petPrompt.Enabled = false
    for i = 0, FOLLOW_DURATION, 0.25 do
        local character = player.Character
        if character and character.PrimaryPart then
            self._model.Humanoid:MoveTo(character.PrimaryPart.Position)
        end
        wait(0.25)
    end
    self._petPrompt.Enabled = true

    return Pet
end

-- Create a new Pet object and pass in the desired model
local rufus = Pet.new(ServerStorage.Dog:Clone())
local whiskers = Pet.new(ServerStorage.Cat:Clone())
```

TIP

Modify the ProximityPrompt if You Can't See It

If you have trouble finding the ProximityPrompt while testing, try setting `RequiresLineOfSight` to False. It may be buried within the model and therefore blocked from view. You can also customize other ProximityPrompt properties, such as `UIOffset` and `Exclusivity`, as needed.

Summary

An important part of DRY coding is not repeating yourself. Creating classes saves you time and work by creating reusable code. In the next hour, you find out how classes can be further customized. So maybe instead of just having unique models for pets, you can have different pets, each with its own model, texture, and sound.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. What does OOP stand for?
2. What is a class?
3. What is at least one benefit of OOP?
4. What is a constructor?
5. What's wrong with the following code?

```
local NameOfClass = {}
NameOfClass.__index = NameOfClass

function NameOfClass.new(parameterProperty)
    local self = {}

    self.defaultProperty = "Default Value"
    self.parameterProperty = parameterProperty

    return self
end

local newObject = NameOfClass.new()
```

Answers

1. Object-oriented programming
2. A class describes what an object is and does.
3. Benefits of OOP are
    a. Allows you to break an experience into smaller chunks
    b. Keeps your code organized
    c. Cuts down on the amount of repetitive code in your project
4. A constructor is a function that describes how to build a new object of a class. All of the properties of a class go inside the constructor.

5. The metatable wasn't set.

```
local NameOfClass = {}
NameOfClass.__index = NameOfClass

function NameOfClass.new(parameterProperty)
    local self = {}
    setmetatable(self, NameOfClass) -- Was missing

    self.defaultProperty = "Default Value"
    self.parameterProperty = parameterProperty

    return self
end

local newObject = NameOfClass.new()
```

Exercise

Create an NPC person class where each NPC person has a parameter to set their name, and has a function to print out their name when called.

Tips

- Create the class and constructor first.
- Include a parameter that allows the name to be passed in.
- You don't need to include a model or other information unless you want to.
- Create the function outside of the constructor and call it separately.

You can find the code solution in the appendix.

HOUR 21

Inheritance

What You'll Learn in This Hour:

- What the relationship between parent and child classes (inheritance) is
- How to create child classes that inherit properties and functions from a parent class
- How to overload functions for polymorphism
- How to call parent functions

As your projects get larger and more complex, you might notice some of your classes will overlap and share certain characteristics. In such cases, you can structure your code so that classes pass on their properties and functions to other classes. We call this practice *inheritance*. The classes that pass on their behaviors are called *parent* classes, and the classes that inherit these behaviors are called *child* classes.

Consider the built-in Roblox classes PointLight and SpotLight. Although they illuminate a scene in slightly different ways, they also have a lot in common, as you can see in the properties shown in Figure 20.1. You can turn either of them on and off, and you can adjust the color and brightness for both. Both of these classes inherit these behaviors from their parent class Light.

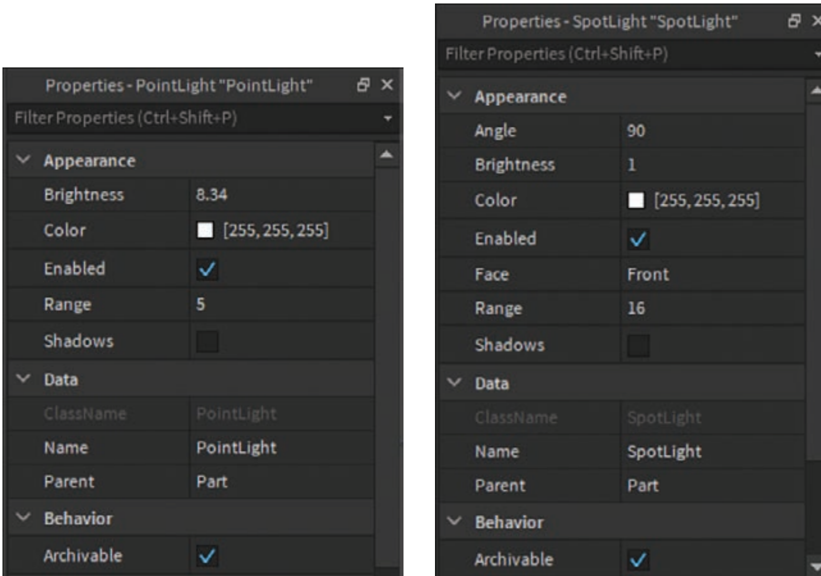


FIGURE 21.1
PointLight (left) and SpotLight (right) share a number of properties from their parent class, Light.

Setting Up Inheritance

In Lua, inheritance is built on the same class and metatable structure that we used for classes in the previous chapter. The following steps walk you through the basic pattern:

1. Create the parent class and its constructor:

```
local ParentClass = {}
ParentClass.__index = ParentClass

function ParentClass.new()
    local self = {}
    setmetatable(self, ParentClass)
    return self
end
```

2. Create the child class:

```
local ParentClass = {}
ParentClass.__index = ParentClass

function ParentClass.new()
    local self = {}
    setmetatable(self, ParentClass)
```

```
    return self
end

local ChildClass = {}
ChildClass.__index = ChildClass
```

3. To make a child class, use the `setmetatable` function. This time, pass in the name of the child class, and then the name of the parent class:

```
local ParentClass = {}
ParentClass.__index = ParentClass

function ParentClass.new()
    local self = {}
    setmetatable(self, ParentClass)
    return self
end

local ChildClass = {}
ChildClass.__index = ChildClass
setmetatable(ChildClass, ParentClass)
```

4. Create the constructor for the new class:

```
local ParentClass = {}
ParentClass.__index = ParentClass

function ParentClass.new()
    local self = {}
    setmetatable(self, ParentClass)
    return self
end

local ChildClass = {}
ChildClass.__index = ChildClass
setmetatable(ChildClass, ParentClass)

function ChildClass.new()

end
```

5. Inside the child class constructor, create a variable called **`self`**. However, instead of setting the value to { } like we have been in constructors, have it set to a new instance of the parent class:

```
local ParentClass = {}
ParentClass.__index = ParentClass

function ParentClass.new()
    local self = {}
```

```
    setmetatable(self, ParentClass)
    return self
end

local ChildClass = {}
ChildClass.__index = ChildClass
setmetatable(ChildClass, ParentClass)

function ChildClass.new()
    local self = ParentClass.new()
end
```

6. Set the metatable for the `self` variable to the child class and then return `self`:

```
local ParentClass = {}
ParentClass.__index = ParentClass

function ParentClass.new()
    local self = {}
    setmetatable(self, ParentClass)
    return self
end

local ChildClass = {}
ChildClass.__index = ChildClass
setmetatable(ChildClass, ParentClass)

function ChildClass.new()
    local self = ParentClass.new()
    setmetatable(self, ChildClass)
    return self
end
```

Inheriting Properties

Any properties that a parent class defines will also become properties of children classes. If you remember talking about DRY coding a few hours ago, this is ideal because that means you don't have to repeat yourself by creating the same properties for every class:

```
local ParentClass = {}
ParentClass.__index = ParentClass

function ParentClass.new()
    local self = {}
    setmetatable(self, ParentClass)
    self.inheritedProperty = "Inherited property"
    return self
end
```

TRY IT YOURSELF ▼

Create a Vehicle Class and a Car Class

The pattern just shown will be used to create a `Vehicle` parent class that has a property for the number of engines it has. Then, create a `Car` child class that inherits the engine number property from the `Vehicle` class but then uses its own property to determine the number of wheels the car has.

1. Create the `Vehicle` class and its constructor:

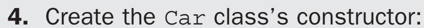
```
local Vehicle = {}
Vehicle.__index = Vehicle

function Vehicle.new()
    local self = {}
    setmetatable(self, Vehicle)
    return self
end
```

2. Add a `numberOfEngines` property to `self`:

```
local Vehicle = {}
Vehicle.__index = Vehicle

function Vehicle.new()
    local self = {}
    setmetatable(self, Vehicle)
    self.numberOfEngines = 1
    return self
end
```

3. Create a `Car` class and have it inherit from `Vehicle`:

```
local Vehicle = {}
Vehicle.__index = Vehicle

function Vehicle.new()
    local self = {}
    setmetatable(self, Vehicle)
    self.numberOfEngines = 1
    return self
end

local Car = {}
Car.__index = Car
setmetatable(Car, Vehicle)
```



4. Create the `Car` class's constructor:

```
local Vehicle = {}
Vehicle.__index = Vehicle

function Vehicle.new()
    local self = {}
    setmetatable(self, Vehicle)
    self.numberOfEngines = 1
    return self
end

local Car = {}
Car.__index = Car
setmetatable(Car, Vehicle)

function Car.new()
    local self = Vehicle.new()
    setmetatable(self, Car)
    return self
end
```

5. Add a `numberOfWheels` property to the `Car` class:

```
local Vehicle = {}
Vehicle.__index = Vehicle

function Vehicle.new()
    local self = {}
    setmetatable(self, Vehicle)
    self.numberOfEngines = 1
    return self
end

local Car = {}
Car.__index = Car
setmetatable(Car, Vehicle)

function Car.new()
    local self = Vehicle.new()
    setmetatable(self, Car)
    self.numberOfWheels = 4
    return self
end
```

6. Create a new instance of the car and print the number of engines and wheels it has:

```
local Vehicle = {}
Vehicle.__index = Vehicle

function Vehicle.new()
    local self = {}
    setmetatable(self, Vehicle)
    self.numberOfEngines = 1
    return self
end

local Car = {}
Car.__index = Car
setmetatable(Car, Vehicle)

function Car.new()
    local self = Vehicle.new()
    setmetatable(self, Car)
    self.numberOfWheels = 4
    return self
end

local car = Car.new()
print("Engines:", car.numberOfEngines)
print("Wheels:", car.numberOfWheels)
```

Working with Multiple Child Classes

You can have as many classes as you want to inherit from a parent class. In the preceding example, you created a `Car` class that inherits from a `Vehicle` class. You can also make a `Motorcycle` class that also inherits from the `Vehicle`. Although the car and motorcycle will have distinct properties, in this case the number of wheels, they will share the number of engines:

```
local Motorcycle = {}
Motorcycle.__index = Motorcycle
setmetatable(Motorcycle, Vehicle)

function Motorcycle.new()
    local self = Vehicle.new()
    setmetatable(self, Motorcycle)
    self.numberOfWheels = 2
    return self
end
```

```
local motorcycle = Motorcycle.new()
print("Engines:", motorcycle.numberOfEngines)
print("Wheels:", motorcycle.numberOfWheels)
```

Inheriting Functions

Just like they do with properties, child classes will also inherit all of the functions of their parent class:

```
local ParentClass = {}
ParentClass.__index = ParentClass

function ParentClass.new()
    local self = {}
    setmetatable(self, ParentClass)
    return self
end

function ParentClass:inheritedFunction()

end
```

Understanding Polymorphism

Sometimes child classes need to perform similar actions, but each class executes those functions in specific ways. In these cases, the parent class can define a function that says what the default behavior of child classes that use that function will be. But any child classes that want to break from this and do something special can have a function with the same name that will override the function they inherited from the parent:

```
local ParentClass = {}
ParentClass.__index = ParentClass

function ParentClass:doSomething()
    -- Default behavior if child classes do not define their own doSomething
end

local ChildClassOne = {}
setmetatable(ChildClassOne, ParentClass)

function ChildClassOne:doSomething()
    -- Does something specific to ChildClassOne
end
```

```
local ChildClassTwo = {}
setmetatable(ChildClassTwo, ParentClass)

function ChildClassTwo:doSomething()
    -- Does something specific to ChildClassTwo
end

local ChildClassThree = {}
setmetatable(ChildClassThree, ParentClass)
-- ChildClassThree did not define a doSomething function.
-- It will call the Parent class doSomething function instead.
```

TRY IT YOURSELF ▼

Create Different Sounding Animals

Let's say you have an `Animal` parent class and two child classes, `Dog` and `Cat`. You want dogs and cats able to make the sounds "woof" and "meow," respectively. You could make separately named functions for these in each class, such as `Dog:woof()` and `Cat:meow()`, but a more common practice would be to create a shared function name. In this case, make a function `Animal:speak()`:

1. Create the `Animal` parent class and its constructor:

```
local Animal = {}
Animal.__index = Animal

function Animal.new()
    local self = {}
    setmetatable(self, Animal)
    return self
end
```

2. Create the `Dog` and `Cat` child classes and their constructors:

```
local Animal = {}
Animal.__index = Animal

function Animal.new()
    local self = {}
    setmetatable(self, Animal)
    return self
end

local Dog = {}
Dog.__index = Dog
setmetatable(Dog, Animal)
```

```
function Dog.new()
    local self = Animal.new()
    setmetatable(self, Dog)
    return self
end

local Cat = {}
Cat.__index = Cat
setmetatable(Cat, Animal)

function Cat.new()
    local self = Animal.new()
    setmetatable(self, Cat)
    return self
end
```

3. Add a `speak` function to the `Animal` class:

```
local Animal = {}
Animal.__index = Animal

function Animal.new()
    local self = {}
    setmetatable(self, Animal)
    return self
end

function Animal:speak()
    print("The animal makes a noise")
end

local Dog = {}
Dog.__index = Dog
setmetatable(Dog, Animal)

function Dog.new()
    local self = Animal.new()
    setmetatable(self, Dog)
    return self
end

local Cat = {}
Cat.__index = Cat
setmetatable(Cat, Animal)

function Cat.new()
    local self = Animal.new()
```

```
        setmetatable(self, Cat)
        return self
    end
```

TIP

Use Colons for Functions

Notice that we are using a colon instead of dot notation like you would with properties.

4. Add `speak` functions to the `Dog` and `Cat` classes:

```
local Animal = {}
Animal.__index = Animal

function Animal.new()
    local self = {}
    setmetatable(self, Animal)
    return self
end

function Animal:speak()
    print("The animal makes a noise")
end

local Dog = {}
Dog.__index = Dog
setmetatable(Dog, Animal)

function Dog.new()
    local self = Animal.new()
    setmetatable(self, Dog)
    return self
end

function Dog:speak()
    print("Woof")
end

local Cat = {}
Cat.__index = Cat
setmetatable(Cat, Animal)

function Cat.new()
    local self = Animal.new()
    setmetatable(self, Cat)
```

```
        return self
    end

    function Cat:speak()
        print("Meow")
    end
```

5. At the bottom of the script, call `speak()` for both the `Cat` and `Dog` classes:

```
Cat:speak()
Dog:speak()
```

Calling Parent Functions

With polymorphism, you may sometimes want to call the default function that the parent class defines, as well as custom behavior from the child class. If a parent class and child class have a function with the same name, you can call the parent function from the child class with the following pattern:

```
function ChildClass:sameFunctionName()
    ParentClass.sameFunctionName(self)
end
```

Notice that you are doing something slightly different from normal with this function call. You're not calling the function from a specific object; you're calling it with the class itself. You're also using the dot (.) operator instead of the colon (:) operator to call the function. Last, you pass in the variable `self`. This variable refers to the actual object you want to call the function with.

The following code sample shows two different jobs people can take on within an experience, warrior and mage. For both jobs, people must use energy to attack. The energy property and the attack function are both set within the parent job class:

```
local Job = {}
Job.__index = Job

function Job.new()
    local self = {}
    setmetatable(self, Job)
    self.energy = 1
    return self
end
```

```
function Job:attack()
    if self.energy > 0 then
        self.energy -= 1
        return true
    end
    return false
end

local Warrior = {}
Warrior.__index = Warrior
setmetatable(Warrior, Job)

function Warrior.new()
    local self = Job.new()
    setmetatable(self, Warrior)
    return self
end

function Warrior:attack()
    local couldAttack = Job.attack(self)
    if couldAttack then
        print("I swing my weapon!")
    else
        print("I'm too tired to attack!")
    end
end

local Mage = {}
Mage.__index = Mage
setmetatable(Mage, Job)

function Mage.new()
    local self = Job.new()
    setmetatable(self, Mage)
    return self
end

function Mage:attack()
    local couldAttack = Job.attack(self)
    if couldAttack then
        print("I cast a spell!")
    else
        print("I'm out of mana!")
    end
end
```

```
local warrior = Warrior.new()
local mage = Mage.new()

-- The first attack() is called, they'll attack.
-- The second time they'll be out of mana.
warrior:attack()
warrior:attack()
mage:attack()
mage:attack()
```

Summary

Inheritance and polymorphism are more tools in your belt when it comes to good object-oriented programming practices. Once you have a parent class, like a zoo animal class, you can then create as many child classes as you want. Default properties like the animal's number of heads, tails, and legs can be set within the parent class, and then passed on to the child class.

With polymorphism, the functions and properties of the parent class can be tweaked so that each zoo animal is unique. A zebra and chimpanzee might have the same wandering behaviors but different animations and sounds.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. True or false: A child class needs to include every property from the parent class.
2. What is the definition of inheritance?
3. What is the maximum number of child classes that can be created?
4. What is the definition of polymorphism?
5. True or false: Child classes inherit properties, but not functions.

Answers

1. False. If a property exists in the parent class and isn't defined in the child class, the child uses what's stated in the parent class.
2. Inheritance is when the functions and properties of one class are passed to another.
3. There is no maximum number of child classes. You can have as many as you want.

4. Sometimes child classes need to perform similar actions but execute them in different ways. The example in the hour was of animals each having a unique sound.

5. False. Child classes can inherit both functions and properties.

Exercise

Try making two jobs for a role-playing game. Make a class for each job. Each job should keep track of how much experience the player has earned and have a function to add to that experience. Each job should also have a special resource that they keep track of, such as energy, stamina, mana, etc. The jobs should have an attack function that depletes this resource. Once you are done, make objects of these classes.

Tips

- If you find two classes have the exact same variables or code, see if you can use a parent class to your advantage.
- Printing the values of object properties before and after function calls can be a good way to make sure your function works correctly.

This page intentionally left blank

HOUR 22
Raycasting

What You'll Learn in This Hour:

- How to raycast
- How to find an object on a ray
- How to find the ray's direction from the origin and destination
- How to ignore an object on a ray

Objects in a Roblox experience can move around a lot. Sometimes you'll need your code to examine the environment to understand where things are. One way to do this is with a technique called *raycasting*. When you raycast, you tell the engine to start at a given point and draw a line from that point in a certain direction for a certain distance. If that line hits anything as it is drawing, then the raycast function returns what is hit. This technique is used for everything from creating detailed reflections on virtual glass surfaces to tracing bullet paths in competitive games.

Setting Up the Function to Raycast

To raycast in Roblox, you use the `workspace:Raycast()` function. This function takes three parameters: the origin of the ray, the direction the ray should go in, and a third optional parameter that lets you specify certain behaviors of the ray. We will get to the third parameter later in the chapter. For now, let's focus on the first two:

1. Define parameters for the origin and direction of the ray using Vector3 coordinates:

```
local origin = Vector3.new(0, 5, 0)
local direction = Vector3.new(0, -10, 0)
```

2. Call `workspace:Raycast()` and store the result in a variable:

```
local origin = Vector3.new(0, 5, 0)
local direction = Vector3.new(0, -10, 0)
local result = game.Workspace:Raycast(origin, direction)
```

3. Check if the result exists:

```
local origin = Vector3.new(0, 5, 0)
local direction = Vector3.new(0, -10, 0)
local result = game.Workspace:Raycast(origin, direction)
if result then
    -- Do something
end
```

4. If the result exists, print the instance that it hit:

```
local origin = Vector3.new(0, 5, 0)
local direction = Vector3.new(0, -10, 0)
local result = game.Workspace:Raycast(origin, direction)
if result then
    print("Found object:", result.Instance)
end
```

▼ TRY IT YOURSELF

Chameleon Material

In this Try It Yourself, you're going to make a part that camouflages itself by copying the material of whatever part it is over, as shown in Figure 22.1.

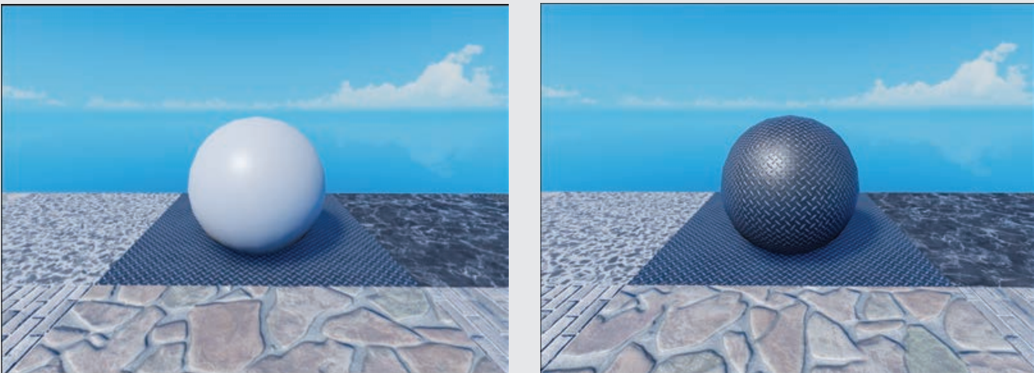


FIGURE 22.1
Before the ray is cast (left), the material will be plastic; then the part will update to match the part beneath it (right).

1. Create a few tiles of different materials, as shown in Figure 22.1, and one part to act as the chameleon.

2. Insert a script into the part; then reference the part:

```
local camouflagePart = script.Parent
```

3. Use the camo part to define the origin for the ray; then define the direction:

```
local camouflagePart = script.Parent
local origin = camouflagePart.Position

local direction = Vector3.new(0, -5, 0)
```

4. Create a ray and store the result:

```
local camouflagePart = script.Parent
local origin = camouflagePart.Position
local direction = Vector3.new(0, -5, 0)
local result = game.Workspace:Raycast(origin, direction)
```

5. If a part was found along the ray, update the material of the camo part to match:

```
local camouflagePart = script.Parent
local origin = camouflagePart.Position
local direction = Vector3.new(0, -5, 0)
local result = game.Workspace:Raycast(origin, direction)
if result then
    camouflagePart.Material = result.Material
end
```

TIP

Ray Length

If the part fails to change materials, it's possible the ray isn't long enough. Either update the direction or move the camo part.

3D Math Trick: Getting the Direction

The second parameter of the `Raycast` function, the direction, is a Vector3. In simple cases, you can know the direction you want the ray to go and just hard code those values. For example, when you want to check below a point, simply use negative values for the y axis. But what if you want to see if there is something between two points, and they aren't simply above and below each other, as in Figure 22.2?

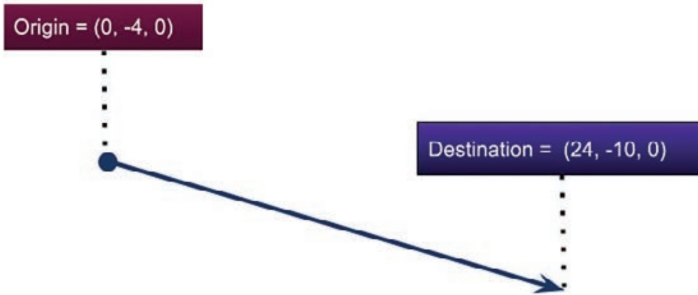


FIGURE 22.2
The origin and destination are offset from each other, and the direction is unknown.

Fortunately, there is a convenient way to get a direction between two points using vector math. Simply subtract the origin's position from the destination's position:

```
local pointA = Vector3.new(0, -4, 0)
local pointB = Vector3.new(24, -10, 0)
local fromAToB = pointB - pointA
```

Setting Raycast Parameters

While detecting parts on a ray is useful, there are several configurations we can use for more complicated situations. In particular, there are times when we care only about detecting certain parts in the environment but not others. The following steps show how to check for an object while ignoring a list of other objects:

1. Create variables for the origin and direction of the ray:

   ```
   local origin = Vector3.new(0, 5, 0)
   local direction = Vector3.new(0, -10, 0)
   ```

2. Create a new `RaycastParams` object and store it in a variable:

   ```
   local origin = Vector3.new(0, 5, 0)
   local direction = Vector3.new(0, -10, 0)
   local parameters = RaycastParams.new()
   ```

3. Set the value of `parameters.FilterDescendantsInstances` to a new table:

```
local origin = Vector3.new(0, 5, 0)
local direction = Vector3.new(0, -10, 0)
local parameters = RaycastParams.new()
parameters.FilterDescendantsInstances = {
  -- Filter information goes here
}
```

4. Put objects you want the raycast to ignore in the new table:

```
local origin = Vector3.new(0, 5, 0)
local direction = Vector3.new(0, -10, 0)
local parameters = RaycastParams.new()
parameters.FilterDescendantsInstances = {
    game.Workspace.IgnorePart1,
    game.Workspace.IgnorePart2,
}
```

5. Set the `FilterType` parameter of the parameters object to `Enum.RaycastFilterType.Blacklist`:

```
local origin = Vector3.new(0, 5, 0)
local direction = Vector3.new(0, -10, 0)
local parameters = RaycastParams.new()
parameters.FilterDescendantsInstances = {
    game.Workspace.IgnorePart1,
    game.Workspace.IgnorePart2,
}
parameters.FilterType = Enum.RaycastFilterType.Blacklist
```

6. Call the `Raycast` function:

```
local origin = Vector3.new(0, 5, 0)
local direction = Vector3.new(0, -10, 0)
local parameters = RaycastParams.new()
parameters.FilterDescendantsInstances = {
    game.Workspace.IgnorePart1,
    game.Workspace.IgnorePart2,
}
parameters.FilterType = Enum.RaycastFilterType.Blacklist
local result = game.Workspace:Raycast(origin, direction, parameters)
```

TRY IT YOURSELF

Raycast Through a Window

Create a translucent window with an object on either side of it. Cast a ray between the objects, ignoring the window.

1. Create a sphere and a cube on either side of a glass window as shown in Figure 22.3.

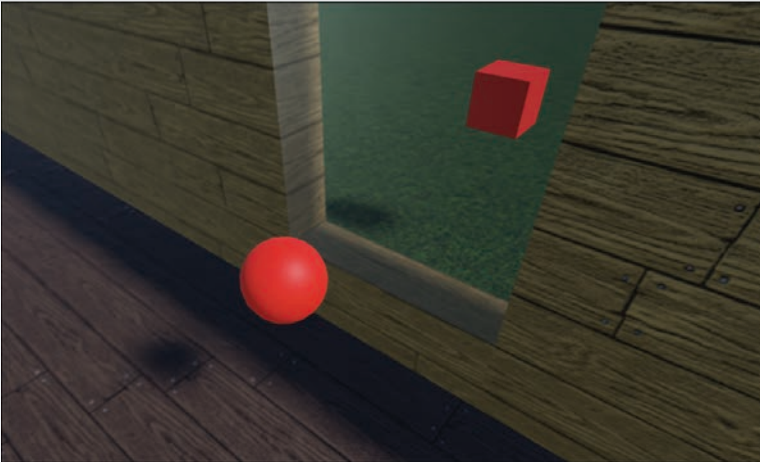


FIGURE 22.3
There are two red parts on either side of a clear glass window.

2. In ServerScriptService, add a script and create references for both of the parts and for the window.
3. Use the position of the sphere as the origin:

```
local sphere = game.Workspace.Sphere
local cube = game.Workspace.Cube
local window = game.Workspace.Window

local origin = sphere.Position
```

4. To find the direction, subtract the position of the sphere from the cube:

```
local origin = sphere.Position
local direction = cube.Position - sphere.Position
```

5. Create a new `RaycastParams` object:

```
local origin = sphere.Position
local direction = cube.Position - sphere.Position
local parameters = RaycastParams.new()
```

6. Set up the filter list and include the window:

```
local origin = sphere.Position
local direction = cube.Position - sphere.Position
local parameters = RaycastParams.new()
parameters.FilterDescendantsInstances = {
    window,
}
parameters.FilterType = Enum.RaycastFilterType.Blacklist
```

7. Cast the ray and confirm the window is not returned in the results:

```
local sphere = game.Workspace.Sphere
local cube = game.Workspace.Cube
local window = game.Workspace.Window

local origin = sphere.Position
local direction = cube.Position - sphere.Position
local parameters = RaycastParams.new()
parameters.FilterDescendantsInstances = {
    window,
}
parameters.FilterType = Enum.RaycastFilterType.Blacklist

local result = game.Workspace:Raycast(origin, direction, parameters)
if result then
    print("Found object:", result.Instance)
end
```

TIP

The Absence of Evidence Is Not Evidence of Absence

So far, you've only tested that the window is not being detected by the ray. When coding, try always to think of multiple ways you can test your scripts. In this case, you should also confirm that parts other than the window will be detected.

3D Math Trick: Limit Direction

The second parameter of the `Raycast` function not only determines the direction of the ray, it also determines how long the ray is. In some cases, you may want the ray to extend a certain distance.

For example, if you use raycasting to help enemies spot a player, you may not want them to be able to see all the way across the map. You'd want to put a cap on how far they could see. In

such cases, you can use the unit vector of the direction. The unit vector of a Vector3 is another vector that is in the same direction, but it's only 1 stud long:

```
local maximumDistance = 10
local pointA = Vector3.new(2.5, 10, 0)
local pointB = Vector3.new(16, 5, -9)
local fromAToB = pointB - pointA
local unit = fromAToB.Unit
local fromAToBCapped = unit * maximumDistance
```

Summary

Raycasting draws a line and returns the results of whatever is found on that line. You can use this technique to check for obstacles, draw reflections, create weapon's fire, or update an object based on its surroundings.

The ray requires an X, Y, Z origin point, and an X, Y, Z direction. If you don't know the direction but you do know the end point for the ray, you can find the direction by subtracting the origin from the destination. So, if you have an origin of (2, 2, 2) and a destination coordinate of (7, 7, 7), your direction would be (5, 5, 5).

Optionally, parameters can be provided for the ray so that it excludes certain objects along its path.

Q&A

Q. What if instead of ignoring objects along the ray, I only want to return certain objects?

A. If you want to search for only certain types of objects instead of blacklisting objects, you can filter using `Enum.RaycastFilterType.Whitelist`.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. What are the two required parameters for casting a ray?
2. How can you find the direction of a ray when you know the origin and destination?
3. True or false: The ray will keep searching for things along the path that's been drawn.
4. True or false: You can set a maximum length for the ray to cast.

Answers

1. Origin and direction.
2. Subtract the origin from the destination to find the direction for a ray.
3. False. The ray returns results only once unless you tell it otherwise.
4. True. You can use the unit vector of the direction to set a maximum length a ray is allowed to draw.

Exercise

Build a detector in your game that senses when a player is nearby. The detector should change colors if a player is within a certain distance of the detector. Remember that you can get all of the players in a game using `Players:GetPlayers()`, and you can get a player character with `player.Character`.

Tip: Until now, you've been raycasting only once per script. Keep in mind that the `Raycast` function only draws its ray at the exact moment when you call it. To build a proper detector, you need to raycast every couple of milliseconds.

This page intentionally left blank

HOUR 23

Plopping Objects in an Experience: Part 1

What You'll Learn in This Hour:

- ▶ How to create a button that allows people to place objects
- ▶ What the render step is
- ▶ How to bind functions to the render step

Allowing your users to control what their environment looks like gives them a chance to express themselves within the world you've made, and that makes them more engaged and more likely to come back. For the next two hours, you'll work on your capstone project: creating a button that enables people in your experience to place or "plop" an item wherever they would like. They can use it to decorate their house as in Figure 23.1 or plant flowers in their garden.

Two new concepts will be covered as you work through the project: the ability to update an object or code whenever the game refreshes and to detect player clicks within the 3D environment.

FIGURE 23.1
MeepCity by AlexNewtron allows you to decorate your whole house, even the bathroom.

By the end of this first hour, you'll learn to track a user's mouse movement to allow them to drag a ghost image that only they can see. To do this, you'll learn about the render step. In the next hour, you'll learn how to listen for user input to finalize the placement of the object on the server.

Setting Up the Object

For this project, you need an object players can place and a part to plop the object onto. In this section, you also set up the events, buttons, and folders you need to keep everything organized:

1. In Workspace, add a new folder named **`Surfaces`** to hold all the parts that people can place items onto.
2. Within the folder, create a new part to act as a floor to place the item onto, as shown in Figure 23.2.

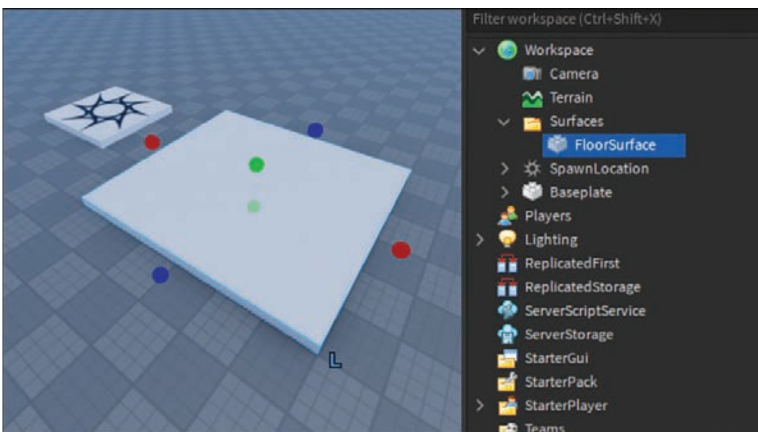


FIGURE 23.2
This large part acts as a floor.

3. In ReplicatedStorage, add a new folder named **`Events`** and a new RemoteEvent named **`PlopEvent`** (see Figure 23.3).
4. Also in ReplicatedStorage, add a folder named **`GhostObjects`**.
5. Decide on the object you want to place and create a slightly transparent ghost version the user will drag around before finalizing the placement. The model should have a base at the bottom of the model that will be used to line up the object with the floor. Figure 23.4 shows a ghost chair.

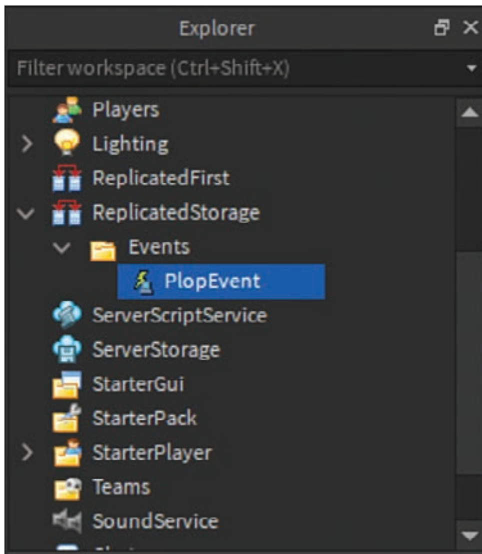


FIGURE 23.3
Add a new RemoteEvent in ReplicatedStorage.

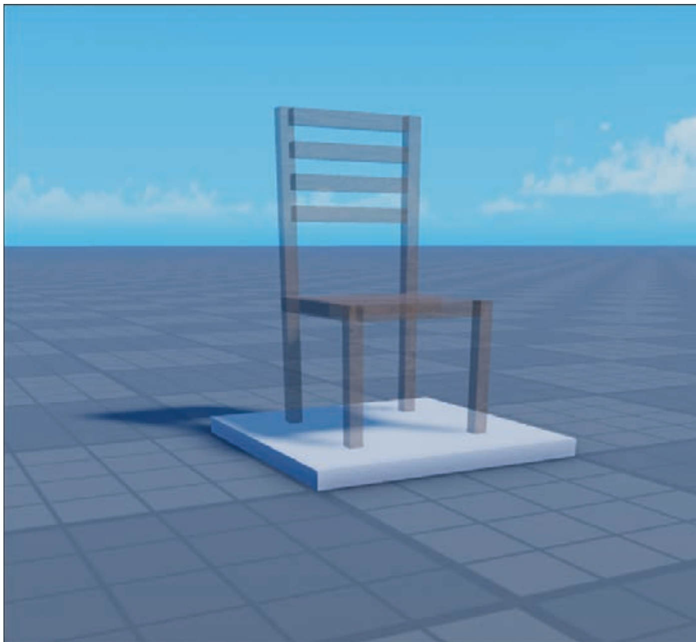


FIGURE 23.4
The slightly transparent ghost model has a base at the bottom.

6. Make sure the base is the PrimaryPart and insert the whole model into the GhostObjects folder (see Figure 23.5).

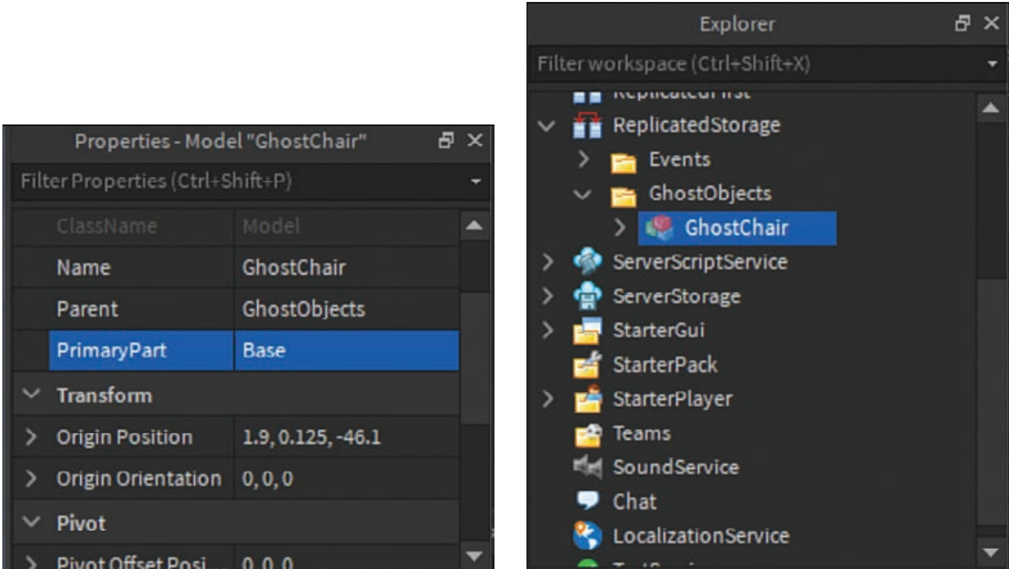


FIGURE 23.5
After setting the PrimaryPart (left), move the model to the GhostObjects folder (right).

7. In ServerStorage, add another folder named Ploppables and insert a copy of the same model at normal transparency (see Figure 23.6). It should also have a basepart as the PrimaryPart.

8. In StarterGui, add a ScreenGui and insert a TextButton named PlopButton (see Figure 23.7).

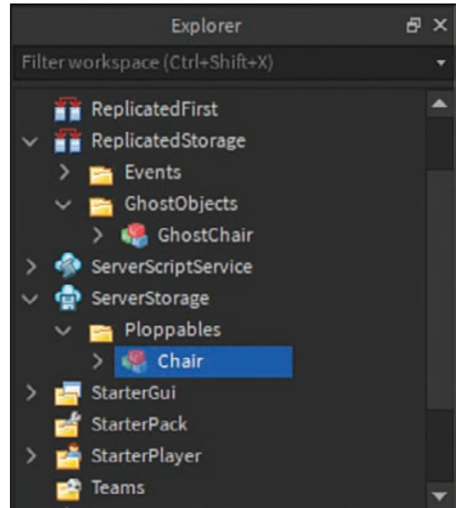


FIGURE 23.6
This folder will hold the objects that will actually be placed in the world.

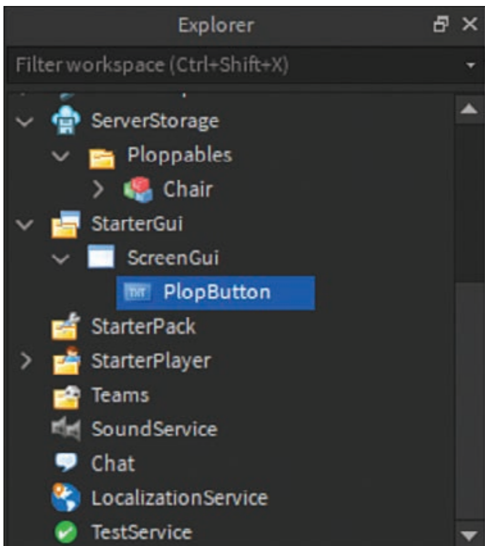


FIGURE 23.7
People will click the button to plop an object.

Creating a Plop Button

You start on the client side and create a LocalScript to create a ghost object people can move around as they decide where they want it to go. Since this is client side, only the user will be able to see the ghost. The first step is to hook up input to the plop button because that is the first part of the sequence of actions the player will take in the plopping logic:

1. In StarterPlayer > StarterPlayerScripts, add a new LocalScript.
2. In the LocalScript, create variables for Player service and the local player:

```
local Players = game:GetService("Players")

local player = Players.LocalPlayer
```

3. Create variables for the GUI components:

```
-- Previous code
local playerGui = player:WaitForChild("PlayerGui")
local plopScreen = playerGui:WaitForChild("ScreenGui")
local plopButton = plopScreen:WaitForChild("PlopButton")
```

4. Create a function named **`OnPlopButtonActivated`**:

```
-- Previous code
local function onPlopButtonActivated()

end
```

5. Inside the function, switch the `Visible` property of the button to false. This way, the button is hidden while the player is plopping:

```
-- Previous code
local function onPlopButtonActivated()
    plopButton.Visible = false
end
```

6. Link the function to the plopButton's `Activated` event:

```
local Players = game:GetService("Players")

local player = Players.LocalPlayer

local playerGui = player:WaitForChild("PlayerGui")
local plopScreen = playerGui:WaitForChild("ScreenGui")
local plopButton = plopScreen:WaitForChild("PlopButton")
```

```
local function onPlopButtonActivated()
    plopButton.Visible = false
end

plopButton.Activated:Connect(onPlopButtonActivated)
```

7. Test the code and make sure the button disappears after you click it.

Tracking Mouse Movements

Now that you have a button to start plopping, the next step is to track the mouse movements of the player because you need that information later to determine where to plop the object.

BindToRenderStep

Every time your screen refreshes, there's a whole host of code happening behind the scene that calculates what should appear on the screen. This is called the render step. If something needs to be smoothly animated, such as a camera, you can add functions to the render step using `BindToRenderStep()`. However, you want to be careful about adding too many. Giving the render step too much to do will slow down how often the screen refreshes, which makes your experience appear laggy and the motion seem jerky.

`BindToRenderStep` is part of `RunService`, and it has three parameters. It looks like this:

```
RunService:BindToRenderStep(bindingName, priority, functionName)
```

The three parts are as follows:

- **`bindingName`**: This is not the same as the name of the function; it's the name of the binding.
- **`priority`**: A numeric value stating how soon in the render step the bound function should be calculated.
- **`functionName`**: The name of the function to bind.

In the plopping project, you're going to add code to the renderstep so that a ghost object can be smoothly moved around before finalizing where the permanent object should be placed:

1. In the same LocalScript, create a variable for `RunService`, the service that owns renderstep:

```
local Players = game:GetService("Players")
local RunService = game:GetService("RunService")

local player = Players.LocalPlayer
-- Rest of code
```

2. Create a variable for the binding name:

```
-- Previous variables
local plopButton = plopScreen:WaitForChild("PlopButton")

local PLOP_MODE = "PLOP_MODE"

local function onPlopButtonActivated()
    plopButton.Visible = false
local

plopButton.Activated:Connect(onPlopButtonActivated)
```

3. Above `onPlopButtonActivated()`, create a new function named `onRenderStepped`:

```
-- Previous variables
local plopButton = plopScreen:WaitForChild("PlopButton")

local PLOP_MODE = "PLOP_MODE"

local function onRenderStepped()

end

local function onPlopButtonActivated()
    plopButton.Visible = false
end

plopButton.Activated:Connect(onPlopButtonActivated)
```

4. Inside the `onPlopButtonActivated` function, bind the `onRenderStepped` function:

```
-- Previous variables
local plopButton = plopScreen:WaitForChild("PlopButton")

local PLOP_MODE = "PLOP_MODE"

local function onRenderStepped()
end

local function onPlopButtonActivated()
    plopButton.Visible = false
    RunService:BindToRenderStep(PLOP_MODE,
        Enum.RenderPriority.Camera.Value + 1, onRenderStepped)
end
plopButton.Activated:Connect(onPlopButtonActivated)
```

TIP

Determining Priority Value to Use

Rather than trying to state a specific value, like 20, this code finds the value of the player camera and adds one, making the bound code happen right after.

Raycasting from the Mouse

Next, you use raycasting to draw a line from the user's cursor to any parts in the Surfaces folder.

1. Still in the same script, beneath the `player` variable, add variables for the camera and mouse:

```
-- Previous variables
local player = Players.LocalPlayer
local camera = game.Workspace.Camera
local mouse = player:GetMouse()

local playerGui = player:WaitForChild("PlayerGui")
-- Rest of code
```

TIP

Keep Like Variables Together

We're jumping around a fair amount, but the important thing is to keep like variables together. Services should be with services, objects with objects, and constants with constants.

2. Create `Raycast` parameters and store them in a variable named **`RaycastParameters`**:

```
-- Previous variables
local plopButton = plopScreen:WaitForChild("PlopButton")

local raycastParameters = RaycastParams.new()

local PLOP_MODE = "PLOP_MODE"
-- Rest of code
```

3. Set the filter type to `Whitelist` and add the surfaces folder you made earlier to the list of instances:

```
-- Previous variables
local plopButton = plopScreen:WaitForChild("PlopButton")

local raycastParameters = RaycastParams.new()
raycastParameters.FilterType = Enum.RaycastFilterType.Whitelist
raycastParameters.FilterDescendantsInstances = { game.Workspace.Surfaces }
```

```
local PLOP_MODE = "PLOP_MODE"
-- Rest of code
```

4. Make a variable named **RAYCAST_DISTANCE**. This will control how long a ray the `Raycast` function will use later:

```
-- Previous variables

local PLOP_MODE = "PLOP_MODE"
local RAYCAST_DISTANCE = 200

local function onRenderStepped()
-- Rest of code
```

5. In the `onRenderStepped` function, use the `ScreenPointToRay` function to get the ray starting at the camera and going toward the player's mouse:

```
-- Other variables

local function onRenderStepped()
    local mouseRay = camera:ScreenPointToRay(mouse.X, mouse.Y, 0)
end
-- Rest of code
```

6. Still within `onRenderStepped()`, use the `Raycast` function with the values you got from `ScreenPointToRay()` in the last step. Note that you have to multiply the direction by your `RAYCAST_DISTANCE` value because that value is a unit vector by default:

```
local function onRenderStepped()
    local mouseRay = camera:ScreenPointToRay(mouse.X, mouse.Y, 0)
    -- Casts the ray using the origin and direction from mouseRay
    local raycastResults = game.Workspace:Raycast(mouseRay.Origin,
        mouseRay.Direction * RAYCAST_DISTANCE, raycastParameters)
end
```

7. Test your code again. The game won't do anything different from the last test, but make sure there are no errors.

Here's the code up this point:

```
local Players = game:GetService("Players")
local RunService = game:GetService("RunService")

local player = Players.LocalPlayer
local camera = game.Workspace.Camera
local mouse = player:GetMouse()

local playerGui = player:WaitForChild("PlayerGui")
local plopScreen = playerGui:WaitForChild("ScreenGui")
local plopButton = plopScreen:WaitForChild("PlopButton")
```

```
local raycastParameters = RaycastParams.new()
raycastParameters.FilterType = Enum.RaycastFilterType.Whitelist
raycastParameters.FilterDescendantsInstances = { game.Workspace.Surfaces }

local PLOP_MODE = "PLOP_MODE"
local RAYCAST_DISTANCE = 200

local function onRenderStepped()
    local mouseRay = camera:ScreenPointToRay(mouse.X, mouse.Y, 0)
    local raycastResults = game.Workspace:Raycast(mouseRay.Origin,
        mouseRay.Direction * RAYCAST_DISTANCE, raycastParameters)
end

local function onPlopButtonActivated()
    plopButton.Visible = false
    RunService:BindToRenderStep(PLOP_MODE, Enum.RenderPriority.Camera.Value + 1,
        onRenderStepped)
end

plopButton.Activated:Connect(onPlopButtonActivated)
```

Previewing the Object

The next step is to show a phantom version of the plop object where the player's mouse is pointing. This gives the player a preview of their selection. This example involves placing an object named `GhostChair`. If your object is named differently, be sure to update that in the code:

1. Still in the same script, create variables for ReplicatedStorage and the ghost object stored within:

```
local Players = game:GetService("Players")
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local RunService = game:GetService("RunService")
...
...
raycastParameters.FilterDescendantsInstances = {game.Workspace.Surfaces}

local ghostObjects = ReplicatedStorage:WaitForChild("GhostObjects")
local ghostChair = ghostObjects:WaitForChild("GhostChair")

local PLOP_MODE = "PLOP_MODE"
...
```

TIP

Missing Code

If you see … in a code sample, assume there is code not being shown at the moment.

2. Create a variable named **plopCFrame**. This will store the position where the player's mouse is pointing:

```
...
local ghostChair = ReplicatedStorage:WaitForChild("GhostChair")

local plopCFrame = nil

local PLOP_MODE = "PLOP_MODE"
...
```

3. Inside the onRenderStepped function and after the Raycast, check if the Raycast returned any results:

```
...
local function onRenderStepped()
    local mouseRay = camera:ScreenPointToRay(mouse.X, mouse.Y, 0)
    local raycastResults = game.Workspace:Raycast(mouseRay.Origin,
        mouseRay.Direction * RAYCAST_DISTANCE, raycastParameters)
    if raycastResults then

    end
end
...
```

4. If the Raycast returned a result, that means the mouse is in a valid area, and you should show the object there. Set the value of plopCFrame to the position of the Raycast result:

```
...
local function onRenderStepped()
    local mouseRay = camera:ScreenPointToRay(mouse.X, mouse.Y, 0)
    local raycastResults = game.Workspace:Raycast(mouseRay.Origin,
        mouseRay.Direction * RAYCAST_DISTANCE, raycastParameters)
    if raycastResults then
        plopCFrame = CFrame.new(raycastResults.Position)
    end
end
...
```

5. Use the SetPrimaryPartCFrame function of the ghost object to move the object to plopCFrame:

```
...
local function onRenderStepped()
    local mouseRay = camera:ScreenPointToRay(mouse.X, mouse.Y, 0)
    local raycastResults = game.Workspace:Raycast(mouseRay.Origin,
        mouseRay.Direction * RAYCAST_DISTANCE, raycastParameters)
    if raycastResults then
        plopCFrame = CFrame.new(raycastResults.Position)
        ghostChair:SetPrimaryPartCFrame(plopCFrame)
```

```
        end
    end
    ...
```

6. Move the object into the workspace. In the `else` statement, move the object back into ReplicatedStorage:

```
...
local function onRenderStepped()
    local mouseRay = camera:ScreenPointToRay(mouse.X, mouse.Y, 0)
    local raycastResults = game.Workspace:Raycast(mouseRay.Origin, mouseRay.
Direction * RAYCAST_DISTANCE, raycastParameters)
    if raycastResults then
        plopCFrame = CFrame.new(raycastResults.Position)
        ghostChair:SetPrimaryPartCFrame(plopCFrame)
        ghostChair.Parent = game.Workspace
    else
           plopCFrame = nil
           ghostChair.Parent = ReplicatedStorage
    end
end
...
```

7. Test your game. Make sure that after you click the Plop Chair button that you see the ghost chair when you move your mouse over the build area, as shown in Figure 23.8.

FIGURE 23.8
After clicking the button, the ghost object should appear when hovering over objects in the Surfaces folder.

Summary

In this hour, you learned how to create ghost objects that only the user can see. You also learned about the render step, the short period of time during which all of the graphics are calculated.

To bind objects to the render step, you use `RunService:BindToRenderStep(bindingName, priority, functionName)`. The parameters work as follows:

- **`bindingName`**: This is not the same as the name of the function; it's the name of the binding. It allows you to connect and disconnect things to the render step.
- **`priority`**: A numeric value stating how soon in the render step the bound function should be calculated.
- **`functionName`**: The name of the function to bind.

In the next hour, you'll learn how to finalize the placement of the object by listening for the user to click.

Q&A

Q. Why do we use a whitelist for the raycast?

A. A whitelist allows you to look for only certain objects. You could in theory blacklist every other object in the experience, but that would be a much longer list and a lot more work. You would also need to update the blacklist every time you add a new object to the experience.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. What is the render step?
2. What does `BindToRenderStep` do?
3. What service do you need to add actions to the render step?
4. What is the first parameter of `BindToRenderStep` and what does it do?

Answers

1. The render step is when all the calculations needed to display an image on screen take place.
2. `BindToRenderStep` connects a function to the render step, making it happen during that process.
3. To add actions to the render step, you need the `RunService`.
4. The first parameter is for the name of the binding, and that allows you not only to connect functions to the render step but also disconnect things.

Exercise

Use the same pattern shown so far to add a second object for users to plop, as shown in Figure 23.9.

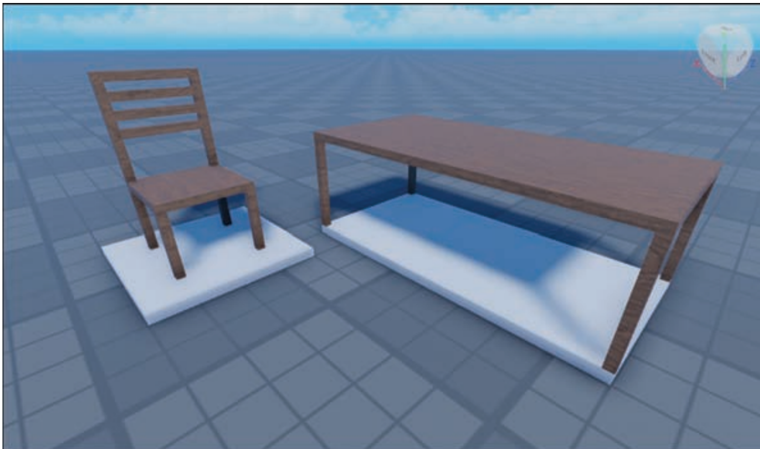


FIGURE 23.9
A table is another good option for people to decorate with.

Tip: Remember to make sure the model has a base set as the primary part.

You can find the code solution in the appendix.

This page intentionally left blank

HOUR 24
Plopping Objects in an Experience: Part 2

What You'll Learn in This Hour:

- How to use `ContextActionService` to check for player input
- How to allow players to place objects in the server
- How to track and raycast using the mouse

This is the second hour of your two-part capstone project. In the last hour, you learned how to track a player's mouse movement and update what they see using the render step. In this hour, you use `ContextActionService` to listen for a player to click and finalize the placement of the object, such as when a player is decorating a house in Figure 24.1.

FIGURE 24.1
In Welcome to Bloxburg, users click to decorate their houses and gardens.

Detecting Mouse Input

Now that you can see where the player is intending to place the object, it's time to listen for when the player clicks so you can finalize the placement of the object. For this, you use `ContextActionService`.

`ContextActionService` allows actions to take place only under certain conditions. A common use is to quickly bind and unbind actions to player input, such as mouse clicks or keyboard presses using `BindAction()`, as follows:

```
ContextActionService:BindAction(actionName, functionName, addMobileButton,
inputTypes)
```

- **`actionName`**: Name of the binding.
- **`functionName:`** The function to call when input is triggered.
- **`addMobileButton:`** Adds a button on the screen for this action on mobile devices.
- **`inputTypes:`** List of inputs to fire this binding.

The function that you bind with `BindAction` should have the following parameters:

```
onInput(actionName, inputState)
```

- **`actionName`**: Name of the binding.
- **`inputState`**: What state the input was in when this function was called.

If you no longer need to listen for particular input, you can remove a binding with the `UnbindAction` function:

```
ContextActionService:UnbindAction(actionName)
```

1. In same script as before, create a variable for `ContextActionService` and one for the binding name:

```
local ContextActionService = game:GetService("ContextActionService")
local Players = game:GetService("Players")
...
...
local plopCFrame = nil

local PLOP_CLICK = "PLOP_CLICK"
local PLOP_MODE = "PLOP_MODE"
...
```

2. Above `onPlopButtonActivated()`, add a function named **`onMouseInput`**:

```
...
local function onMouseInput(actionName, inputState)

end

local function onPlopButtonActivated()
...
```

3. Switch to the `onPlopButtonActivated` function and bind the `onMouseInput` function using the `BindAction` function:

```
...
local function onPlopButtonActivated()
    plopButton.Visible = false
    RunService:BindToRenderStep(PLOP_MODE,
        Enum.RenderPriority.Camera.Value + 1, onRenderStepped)
    ContextActionService:BindAction(PLOP_CLICK, onMouseInput, false,
        Enum.UserInputType.MouseButton1)
end

...
```

4. In the `onMouseInput` function, check if the input state was `End` because this is when the player has clicked the mouse:

```
...
local function onMouseInput(actionName, inputState)
    if inputState == Enum.UserInputState.End then

    end
end
...
```

5. In the `if` statement, make sure the ghost object is back in ReplicatedStorage:

```
...
local function onMouseInput(actionName, inputState)
    if inputState == Enum.UserInputState.End then
        ghostChair.Parent = ReplicatedStorage
    end
end
...
```

6. Also in the `if` statement, unbind the click action and the render actions using `UnbindAction` and `UnbindFromRenderStep`:

```
...
local function onMouseInput(actionName, inputState)
    if inputState == Enum.UserInputState.End then
```

```
            ghostChair.Parent = ReplicatedStorage
            RunService:UnbindFromRenderStep(PLOP_MODE)
            ContextActionService:UnbindAction(PLOP_CLICK)
        end
    end
    ...
```

7. Make the plop button visible again:

```
...
local function onMouseInput(actionName, inputState)
    if inputState == Enum.UserInputState.End then
        ghostChair.Parent = ReplicatedStorage
        RunService:UnbindFromRenderStep(PLOP_MODE)
        ContextActionService:UnbindAction(PLOP_CLICK)
        plopButton.Visible = true
    end
end
...
```

8. Test your game. Confirm that after you click that the object disappears and the plop button appears again.

Sending a Message to the Server

Now you have to let the server know that the player wants to plop an object and where they want to plop it. You use the RemoteEvent that we created in the set up for this:

1. In the same script, create a variable for the remote event:

```
...
local ghostChair = ReplicatedStorage:WaitForChild("GhostChair")

local events = ReplicatedStorage:WaitForChild("Events")
local plopEvent = events:WaitForChild("PlopEvent")

local raycastParameters = RaycastParams.new()
...
```

2. At the end of `onMouseInput` and in the `if` statement, fire the remote event passing the `plopCFrame` variable in as an argument if the `plopCFrame` exists:

```
...
local function onMouseInput(actionName, inputState)
    if inputState == Enum.UserInputState.End then
        ghostChair.Parent = ReplicatedStorage
        RunService:UnbindFromRenderStep(PLOP_MODE)
        ContextActionService:UnbindAction(PLOP_CLICK)
```

```
            plopButton.Visible = true
            if plopCFrame then
                plopEvent:FireServer(plopCFrame)
            end
        end
    end
    ...
```

TIP

Dealing with Multiple Objects

If you did the exercise in the last hour and have multiple objects, you need to send over the name of the ghost object as well.

Getting the Message

Now it's time to switch scripts! A new script in ServerScriptService listens for `PlopEvent` and places the object where everyone can finally see it on the server:

1. In ServerScriptService, add a new script.
2. Create variables for ReplicatedStorage and ServerStorage:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local ServerStorage = game:GetService("ServerStorage")
```

3. Create variables for the RemoteEvent and the chair object:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local ServerStorage = game:GetService("ServerStorage")

local events = ReplicatedStorage.Events
local plopEvent = events.PlopEvent
local ploppables = ServerStorage.Ploppables
local chair = ploppables.Chair
```

4. Create a function named **`onPlop`** that takes a player and CFrame as an argument:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local ServerStorage = game:GetService("ServerStorage")

local events = ReplicatedStorage.Events
local plopEvent = events.PlopEvent
local ploppables = ServerStorage.Ploppables
local chair = ploppables.Chair

local function onPlop(player, cframe)

end
```

TIP

Dealing with Multiple Objects

Remember to add another parameter here to take in an object name if you have more than one.

5. Connect the `onPlop` function to the `OnServerEvent` of `plopEvent`:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local ServerStorage = game:GetService("ServerStorage")

local events = ReplicatedStorage.Events
local plopEvent = events.PlopEvent
local ploppables = ServerStorage.Ploppables
local chair = ploppables.Chair

local function onPlop(player, cframe)
end

plopEvent.OnServerEvent:Connect(onPlop)
```

6. Inside the `onPlop` function, make a copy of the chair using the `Clone` function:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local ServerStorage = game:GetService("ServerStorage")

local events = ReplicatedStorage.Events
local plopEvent = events.PlopEvent
local ploppables = ServerStorage.Ploppables
local chair = ploppables.Chair

local function onPlop(player, cframe)
    local chairCopy = chair:Clone()
end

plopEvent.OnServerEvent:Connect(onPlop)
```

7. Move the copied chair to the CFrame and change its parent to the workspace:

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local ServerStorage = game:GetService("ServerStorage")

local events = ReplicatedStorage.Events
local plopEvent = events.PlopEvent
local ploppables = ServerStorage.Ploppables
local chair = ploppables.Chair
```

```
local function onPlop(player, cframe)
    local chairCopy = chair:Clone()
    chairCopy:SetPrimaryPartCFrame(cframe)
    chairCopy.Parent = game.Workspace
end

plopEvent.OnServerEvent:Connect(onPlop)
```

8. Test your game. Confirm that after clicking where you want the object to go that it does get added to the workspace. Also try placing several chairs.

Summary

Congratulations! You've completed a big project that used all of the knowledge you've acquired throughout this book. Here's a quick summation of the skills you used to create a plopping system that enables users to decorate and have more control of their environments:

1. You used raycasting to find the location of the mouse and a whitelist to ensure it only returned the names of certain objects.
2. Users could drag a ghost object around using their mouse and the location was updated by binding a function to the render step.
3. Users finalized the placement of the object by clicking. You used `ContextActionService` to bind a function to the click and place the object.

This is the last hour of this book, but there's still lots more to learn. All of the code you've worked on in this book should be seen as just a starting point. For every project, you can expand the code out and use what you know in new ways. There's a great community of Roblox developers who can help you out on the Roblox Developer Forums and tons of code samples for you to use on developer.roblox.com.

As you expand on your skills, keep in mind the importance of staying organized and testing your code with multiple scenarios. You should always be thinking about how different situations like differing screen sizes and having multiple people in a server affect how your code behaves. Ideally, you should even get other people to try out your experience and make sure everything works for them as expected.

If you're interested in learning more about how to build experiences and get better at things like lighting, sounds, environments, and animations, you can also check out the companion book, *Learn Roblox Game Development in 24 Hours*.

Q&A

Q. How can you make the plop buttons disappear when you don't need them?

A. You could place all of the GUI needed for the plop buttons within a larger frame named something like Decorations or Shop. By default, disable the frame. Use what you know to create another button people can click to enable and disable the frame and see or hide all of their decor choices.

Q. What other types of actions could you use with `ContextActionService`?

A. You can set up scenarios that enable new controls depending on what a player is doing. For example, if they're in a car, you might enable buttons to act as the breaks, gas, and horn. You can then disable these buttons when the player isn't in the car.

Workshop

Now that you have finished, let's review what you've learned. Take a moment to answer the following questions.

Quiz

1. What does `ContextActionService` allow you to do?
2. What is the function that allows you to enable certain keys under certain circumstances?
3. If you have multiple objects that can be plopped, what property do you need to pass over in addition to the base code?

Answers

1. `ContextActionService` allows you to make it so that an action can only take place in certain conditions.
2. Use `BindAction()` if you want to enable certain keys under certain circumstances.
3. You need to send the name of the object over as well if you have multiple objects that can be plopped.

Exercise

Try adding a command to rotate objects while you're placing them. Every time a player presses a key, the object should rotate a set amount of degrees.

Tips

- Set up a constant for how many degrees the object should rotate.
- Use `ContextActionService` to enable a key such as R to rotate the object.

You can find the code solution in the appendix.

APPENDIX A
Roblox Basics

The following tables list the keys and associated actions when editing in Roblox Studio

TABLE A.1 Studio Camera Movement

Key	Movement
W A S D	Move the camera: W: Forward A: Back S: Left D: Right
E	Raise camera
Q	Lower camera
Shift	Move camera slower
Right mouse button (hold and drag mouse)	Turn camera
Middle mouse button	Drag camera
Mouse scroll wheel	Zoom camera in or out
F	Focus on selected object

TABLE A.2 In-Game Camera Movement

Key	Movement
W A S D	Move the camera: W: Forward A: Back S: Left D: Right
E	Raise camera

Key	Movement
Q	Lower camera
Right mouse button (hold and drag mouse)	Turn camera
Mouse Scroll Wheel	Zoom camera in or out
Alt+P	Free camera

Keywords

Reserved Names in Lua

The following keywords are reserved by Lua and cannot be used as variable or function names:

`and`	`break`	`do`	`else`	`elseif`	`end`	`false`
`function`	`for`	`if`	`in`	`local`	`nil`	`not`
`or`	`repeat`	`return`	`then`	`true`	`until`	`while`

Here is a selection of a few additional keywords that perform important actions that are specific to the Roblox platform:

`script`	`game`	`self`	`workspace`

DataType Index

Lua Data Types

boolean	function	nil	number
string	thread	table	userdata

Roblox Lua Data Types

These data types have been added by Roblox to base Lua. Refer to the API pages on developer.roblox.com to learn more about a particular data type.

A

Axes

B

BrickColor

C

CatalogSearchParams
CFrame
Color3
ColorSequence
ColorSequenceKeypoint

D

DateTime
DockWidgetPluginGuiInfo

E

Enum
EnumItem
Enums

F

Faces

I

Instance

N

NumberRange
NumberSequence
NumberSequenceKeypoint

P

PathWaypoint
PhysicalProperties

R

Random
Ray
RaycastParams
RaycastResult
RBXScriptConnection
RBXScriptSignal
Rect
Region3
Region3int16

T

TweenInfo

U

UDim
UDim2

V

Vector2
Vector2int16ector2
Vector3
Vector3int16

Operators

An operator is a special set of symbols used to perform an operation or conditional evaluation.

Logical

The logical operators for conditional statements are `and`, `or`, and `not`. These operators consider both false and nil as "false" and anything else as "true."

Operator	Description
`and`	Evaluates as true only if both conditions are true.
`or`	Evaluates as true if either condition is true.
`not`	Evaluates as the opposite of the condition.

Relational

Relational operators compare two parameters and return a boolean true or false.

Operator	Description	Associated Metamethod
==	Equal to	__eq
~=	Not equal to	
>	Greater than	
<	Less than	__lt
>=	Greater than or equal to	
<=	Less than or equal to	__le

Arithmetic

Lua supports the usual binary operators along with exponentiation, modulus, and unary negation.

Operator	Description	Example	Associated Metamethod
+	Addition	1 + 1 = 2	__add
-	Subtraction	1 - 1 = 0	__sub
*	Multiplication	5 * 5 = 25	__mul
/	Division	10 / 5 = 2	__div
^	Exponentiation	2 ^ 4 = 16	__pow
%	Modulus	13 % 7 = 6	__mod
-	Unary negation	-2 = 0 - 2	__unm

Miscellaneous

Miscellaneous operators include concatenation and length.

Operator	Description	Associated Metamethod
..	Concatenates two strings	__concat
#	Length of table	__len

Naming Conventions

- Spell out words fully! Abbreviations generally make code easier to write but harder to read.
- Use PascalCase names for class and enum-like objects.
- Use PascalCase for all Roblox APIs. camelCase APIs are mostly deprecated, but still work for now.
- Use camelCase names for local variables, member values, and functions.
- For acronyms within names, don't capitalize the whole thing—for example, a JsonVariable or MakeHttpCall.
- The exception to this is when the abbreviation represents a set—for example, in an RGBValue or GetXYZ. In these cases, RGB should be treated as an abbreviation of RedGreenBlue and not as an acronym.
- Use LOUD_SNAKE_CASE names for local constants.
- Prefix private members with an underscore, such as _camelCase.
- Lua does not have visibility rules, but using a character like an underscore helps make private access stand out.
- A module's name should match the name of the object it exports.

Animation Easing

Animation easing defines a "direction" and style in which a tween will occur.

Style	Description
Linear	Moves at a constant speed.
Sine	Movement speed is determined by a sine wave.
Back	Tween movement backs into or out of place.

Style	Description
Quad	Eases in or out with quadratic interpolation.
Quart	Similar to Quad but with a more emphasized start and/or finish.
Quint	Similar to Quad but with an even more emphasized start and/or finish.
Bounce	Moves as if the start or end position of the tween is bouncy.
Elastic	Moves as if the object is attached to a rubber band.

Direction	Description
In	The tween will have less speed at its beginning and more speed toward its end.
Out	The tween will have more speed at its beginning and less speed toward its end.
InOut	In and Out on the same tween, with In at the beginning and Out taking effect halfway through.

Possible Solutions to Exercises

The following are just some of the possible solutions to the exercises presented in each hour. Your solution may end up being different.

Hour 1

Create a part that destroys whatever touches it.

Location and type: Part > Script

```
-- Destroys whatever touches the part
local lava = script.Parent
local function onTouch(partTouched)
    partTouched:Destroy()
    -- Makes it so players will fall through the lava
    lava.CanCollide = false
end
lava.Touched:Connect(onTouch)
```

Hour 2

Exercise 1

Use code to turn a regular part into an NPC with a greeting and face.

Location and Type: Part > Script

```
-- Turns the parent into an NPC

local guideNPC = script.Parent
local message = "Don't fall in!"
local decal = Instance.new("Decal")

guideNPC.Transparency = 0.25
guideNPC.Dialog.InitialPrompt = message
guideNPC.Color = Color3.fromRGB(40, 0, 160)
decal.Parent = guideNPC
```

Exercise 2

Use code to create a new part instance at the center of the world, and give it a face and dialog.

Location and Type: ServerScriptService > Script

```
-- Creates an NPC at the center of the world
local newNPC = Instance.new("Part")
local message = "Don't fall in!"
local dialog = Instance.new("Dialog")
local decal = Instance.new("Decal")

dialog.InitialPrompt = message
dialog.Parent = newNPC

decal.Texture = "rbxassetid://494291269"
decal.Parent = newNPC

newNPC.Transparency = 0.25
newNPC.Color = Color3.fromRGB(40, 0, 160)
newNPC.Anchored = true
newNPC.Parent = workspace
```

Hour 3

Use a button to activate and deactivate a bridge.

Location and Type: Part > Script

```
-- Button activates bridge when touched

local button = script.Parent
local bridge = workspace.BridgePiece01 -- Locate the bridge

local function deactivateBridge()
    bridge.Transparency = 1
    bridge.CanCollide = false
end

local function onTouch()
    bridge.Transparency = 0
    bridge.CanCollide = true
    -- Give just enough time to cross
    wait(3.0)
    deactivateBridge()
end

button.Touched:Connect(onTouch)
```

Hour 4

Create a part that sets whatever touches it on fire.

Location and type: Part > Script

```
-- Adds fire instance to bonfire
local bonfire = script.Parent
local function onTouch(otherPart)
    local fire = Instance.new("Fire")
    fire.Parent = otherPart
end

bonfire.Touched:Connect(onTouch)
```

Hour 5

Create a part that checks for a humanoid, and if found, increases their walk speed.

Location and type: Part > Script

```
-- Sets walk speed of anyone who touches to 50
local ServerStorage = game:GetService("ServerStorage")
```

```
local speedBoost = script.Parent

local function onTouch(otherPart)
    -- Looks for a humanoid and stores it
    local character = otherPart.Parent
    local humanoid = character:FindFirstChildWhichIsA ("Humanoid")

    -- Checks for Humanoid without speed boost
    if humanoid and humanoid.WalkSpeed <= 16 then
        -- Assumes you have a ParticleEmitter named SpeedParticles
        local speedParticles = ServerStorage.SpeedParticles:Clone()
        speedParticles.Parent = otherPart
        humanoid.WalkSpeed = 50
        wait(2.0)
        humanoid.WalkSpeed = 16
        speedParticles:Destroy()
    end
end

speedBoost.Touched:Connect(onTouch)
```

Hour 6

Exercise 1

It's important to start thinking about the ways in which your code can be improved in the future. Here are a few ways some of the code you've worked with so far can be improved. Maybe you can think of more:

- Players might get confused at still being able to use the ProximityPrompt while it's on cooldown.
- Some people might have low vision or color blindness. Adding sound while mining to let them know it's working might make the experience easier for them.
- People lose their points in between sessions.
- It's a bit hard to tell if gold was received after mining. Maybe a particle or a sound would make it more obvious what happened.
- In the speed part code solution, it only allows for players to go a certain speed. Maybe it can be modified to allow players to go faster and faster each time they touch a part.

Exercise 2

Make anyone who touches a part gigantic (or tiny).

Location and type: Part > Script

```
-- Scales anyone who touches the part
local growthPotion = script.Parent
local originalColor = growthPotion.Color -- Get original color of part
local isEnabled = true
local COOLDOWN = 3.0
local SCALE_AMOUNT = 2.0 -- Change to decimal to shrink avatar

local function onTouch(otherPart)
    local otherPartParent = otherPart.Parent
    local humanoid = otherPartParent:FindFirstChildWhichIsA("Humanoid")

    if isEnabled == true and humanoid then
        isEnabled = false
        growthPotion.Color = Color3.fromRGB(7, 30, 39)

        local headScale = humanoid.HeadScale
        local bodyDepthScale = humanoid.BodyDepthScale
        local bodyWidthScale = humanoid.BodyWidthScale
        local bodyHeightScale = humanoid.BodyHeightScale

        headScale.Value = headScale.Value * SCALE_AMOUNT
        bodyDepthScale.Value = bodyDepthScale.Value * SCALE_AMOUNT
        bodyWidthScale.Value = bodyWidthScale.Value * SCALE_AMOUNT
        bodyHeightScale.Value = bodyHeightScale.Value * SCALE_AMOUNT

        wait(COOLDOWN)

        isEnabled = true
        growthPotion.Color = originalColor
    end
end

growthPotion.Touched:Connect(onTouch)
```

Hour 7

For this solution, it'll be assumed that players can harvest logs and gold ore, so both will be reflected in the harvesting and leaderboard scripts.

Use a part or a mesh as a tree, and add a custom attribute:

- Name: ResourceType
- Type: String
- Value: Logs

Campfire Script

Location and type: ServerScriptService > Script

```
local ProximityPromptService = game:GetService("ProximityPromptService")

-- How long each log lasts
local BURN_DURATION = 3

local function onPromptTriggered(prompt, player)
    if prompt.Enabled and prompt.Name == "AddFuel" then
        local playerstats = player.leaderstats

        local logs = playerstats.Logs
        if logs.Value > 0 then

            logs.Value -= 1
            local campfire = prompt.Parent
            local fire = campfire.Fire

            local currentFuel = campfire:GetAttribute("Fuel")
            campfire:SetAttribute("Fuel", currentFuel + 1)

            if not fire.Enabled then
                fire.Enabled = true
                while campfire:GetAttribute("Fuel") > 0 do
                    local currentFuel = campfire:GetAttribute("Fuel")
                    campfire:SetAttribute("Fuel", currentFuel - 1)
                    wait(BURN_DURATION)
                end
                fire.Enabled = false
            end
        end
    end
end

ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)
```

Leaderboard Script

Location and type: ServerScriptService > Script

```
local Players = game:GetService("Players")

local function statsSetup(player)
    local leaderstats = Instance.new("Folder")
    leaderstats.Name = "leaderstats"
    leaderstats.Parent = player

    local gold = Instance.new("IntValue")
    gold.Name = "Gold"
    gold.Value = 0
    gold.Parent = leaderstats

    local logs = Instance.new("IntValue")
    logs.Name = "Logs"
    logs.Value = 5
    logs.Parent = leaderstats
end

Players.PlayerAdded:Connect(statsSetup)
```

Harvesting Script

Location and type: ServerScriptService > Script

```
local ProximityPromptService = game:GetService("ProximityPromptService")

local DISABLED_DURATION = 10

local function onPromptTriggered(prompt, player)
    local node = prompt.Parent
    local resourceType = node:GetAttribute("ResourceType")
    if resourceType and prompt.Enabled then
        prompt.Enabled = false
        node.Transparency = 0.8

        local leaderstats = player.leaderstats
        local resourceStat = leaderstats:FindFirstChild(resourceType)
        resourceStat.Value += 1

        wait(DISABLED_DURATION)

        prompt.Enabled = true
        node.Transparency = 0
    end
end

ProximityPromptService.PromptTriggered:Connect(onPromptTriggered)
```

Hour 8

Exercise 1

Create a hit box for a campfire, and do damage over time to anyone who touches it.

Location and type: Part > Script

```
-- Does damage over time to anyone who touches hitBox
local hitBox = script.Parent
local BURN_DURATION = 3
local DAMAGE_PER_TICK = 10

local enabled = true

local function onTouch(otherPart)
    local otherPartParent = otherPart.Parent
    local humanoid = otherPartParent:FindFirstChildWhichIsA("Humanoid")

    if humanoid and enabled == true then
        enabled = false
        for burnCount = 0 , BURN_DURATION, 1 do
            humanoid.Health = humanoid.Health - DAMAGE_PER_TICK
            wait(1.0)
        end
        enabled = true
    end
end

hitBox.Touched:Connect(onTouch)
```

Exercise 2

Here's a few ways loops can be used:

- **Day/night cycles:** You can use a `while` loop to cycle through the time of day. There are lots of existing tutorials that can help you if you want to try!
- **Seasonal cycles:** Similar to day time cycles. Inside of the `while` loop, you can trigger environmental changes for each season.
- **Applying changes to multiple objects:** Loops can be used to go through a bunch of objects and make updates. For example, you can change the appearance of a number of objects to make seasonal cycles like the one previously mentioned.
- **Creating rounds and lobbies for games:** Some experiences use a round-based structure to control when a game starts and to wait for players.

- **Creating a vanishing staircase or bridge:** Loops can make objects continuously reappear or disappear.
- **Weapon cooldowns:** Control how long spells and skills need to be recharged between uses. You could use `wait()`, but loops give you greater control.
- **Making objects move back and forth:** It could be an NPC that walks on a set path or a platform that slides from point A to point B.

Hour 9

Create a script that changes the appearance of a number of parts. This example shows a pine tree going from normal green to snowy white.

Location and type: ServerScriptService > Script

```
local treeFolder = workspace.Trees

local trees = treeFolder:GetChildren()

for index, tree in ipairs(trees) do

    local leaves = tree:GetChildren()
    for index, value in ipairs(leaves) do
        if value:isA("BasePart") then
            value.Color = Color3.fromRGB(129, 157, 146)
            -- Add a wait if you want to watch it change
            wait(0.005)
        else
            print("Not a BasePart")
        end
    end
end
```

Hour 10

Assign newcomers to either "red" or "blue" as they join and print the resulting teams.

Location and type: SeverScriptService > Script

```
Players = game:GetService("Players")
local AssignRed = true

local teamAssigments = {

}

local function printTeamAssignments()
    print("Teams are:")
```

```
    for player, team in pairs(teamAssigments) do
        print(player.name .. " is on " .. team .. " team" )
    end
end

local function assignTeam(newPlayer)

    local name = newPlayer.Name
    print("hello " .. name)
    if AssignRed == true then
        teamAssigments[newPlayer] = "Red"
        AssignRed = false

    else
        teamAssigments[newPlayer] = "Blue"
        AssignRed = true
    end

    printTeamAssignments()
end

Players.PlayerAdded:Connect(assignTeam)
```

Hour 11

Leaderstat Code for the Try It Yourself

The following script can be used to set up a leaderboard for use with the Try It Yourself if you don’t have one set up already.

Script Name: PlayerStats

Location and type: ServerScriptService > Script

```
local Players = game:GetService("Players")

local function statsSetup(player)
    local leaderstats = Instance.new("Folder")
    leaderstats.Name = "leaderstats"
    leaderstats.Parent = player

    local gold = Instance.new("IntValue")
    gold.Name = "Gold"
    gold.Value = 40
    gold.Parent = leaderstats

    local logs = Instance.new("IntValue")
    logs.Name = "Logs"
    logs.Value = 5
```

```
    logs.Parent = leaderstats
end

Players.PlayerAdded:Connect(statsSetup)
```

Exercise

The goal of this exercise was to retrieve information about the shop item from the server instead of hard coding it as shown in the Try It Yourself earlier in the hour.

1. Add a new RemoteFunction named GetShopInfo (see Figure A.1).

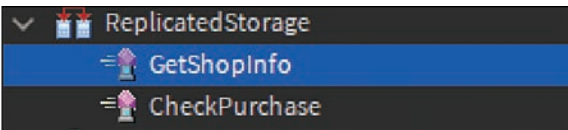


FIGURE A.1
Add an additional RemoteFunction.

2. Add attributes to the items you want to sell for StatName, Price, and NumberToGive (see Figure A.2).

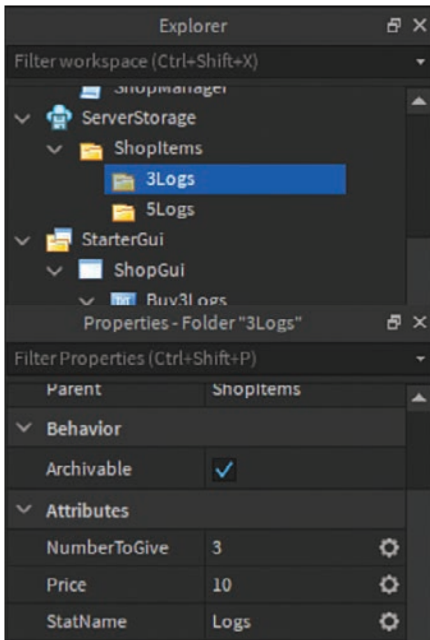


FIGURE A.2
Each item should have its own folder.

Highlighted code is for the exercise; nonhighlighted was from the earlier Try It Yourself.

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local Players = game:GetService("Players")
local ServerStorage = game:GetService("ServerStorage")

local checkPurchase = ReplicatedStorage:WaitForChild("CheckPurchase")
local getShopInfo = ReplicatedStorage:WaitForChild("GetShopInfo")

local shopItems = ServerStorage.ShopItems

local function confirmPurchase(player, purchaseType)
    local leaderstats = player.leaderstats
    local currentGold = leaderstats:FindFirstChild("Gold")

    local purchaseType = shopItems:FindFirstChild(purchaseType)
    local resourceStat =
        leaderstats:FindFirstChild(purchaseType:GetAttribute("StatName"))
    local price = purchaseType:GetAttribute("Price")
    local numberToGive = purchaseType:GetAttribute("NumberToGive")

    local serverMessage

    if currentGold.Value >= price then
        currentGold.Value = currentGold.Value - price
        resourceStat.Value += numberToGive

        serverMessage = ("Purchase Successful!")
    elseif currentGold.Value < price then
        serverMessage = ("Not enough Gold")
    else
        serverMessage = ("Didn't find necessary info")
    end
    return serverMessage
end

-- New For Exercise
local function getButtonInfo(player, purchaseType)
    local purchaseType = shopItems:FindFirstChild(purchaseType)

    local numberToGive = purchaseType:GetAttribute("NumberToGive")
    local statName = purchaseType:GetAttribute("StatName")
    local price = purchaseType:GetAttribute("Price")

    return numberToGive, statName, price
end

checkPurchase.OnServerInvoke = confirmPurchase
getShopInfo.OnServerInvoke = getButtonInfo -- New For Exercise
```

ButtonManager

Location and type: StarterGui > ShopGui (ScreenGui) > Buy3Logs (TextButton) > ButtonManager (LocalScript)

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")

local checkPurchase = ReplicatedStorage:WaitForChild("CheckPurchase")
local getShopInfo = ReplicatedStorage:WaitForChild("GetShopInfo")

local button = script.Parent
local purchaseType = button:GetAttribute("PurchaseType")
-- New for Exercise
local numberToGive, statName, price = getShopInfo:InvokeServer(purchaseType)
local defaultText = "Buy " .. numberToGive .. statName .. " for " .. price

button.Text = defaultText

local COOLDOWN = 2.0

local function onButtonActivated()

    local confirmationText = checkPurchase:InvokeServer(purchaseType)
    button.Text = confirmationText
    button.Selectable = false
    wait(COOLDOWN)
    button.Text = defaultText
    button.Selectable = true
end

button.Activated:Connect(onButtonActivated)
```

Hour 12

For this exercise, you were asked to announce to all clients what map was picked.

The complete setup is shown in Figure A.3.

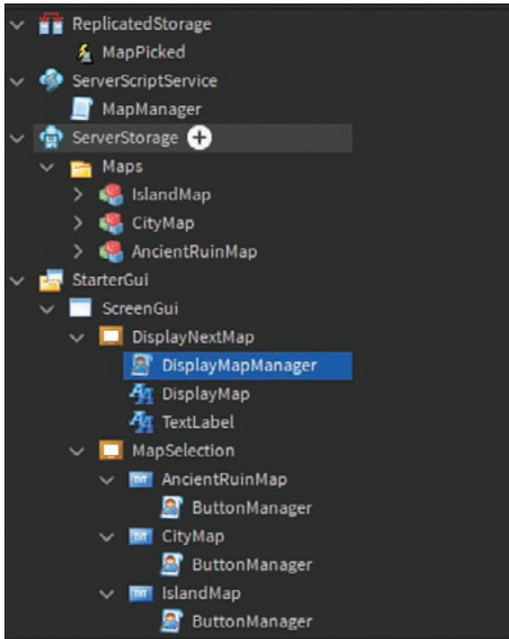


FIGURE A.3
In the complete setup, notice that only one RemoteEvent is needed.

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local mapPicked = ReplicatedStorage:WaitForChild("MapPicked")

local ServerStorage = game:GetService("ServerStorage")

local mapsFolder = ServerStorage:WaitForChild("Maps")
local currentMap = nil

local function announceMap(player, chosenMap)
    print("server says".. chosenMap)
    mapPicked:FireAllClients(chosenMap)
end

local function onMapPicked(player, chosenMap)
    local mapChoice = mapsFolder:FindFirstChild(chosenMap)

    if mapChoice then
        if currentMap then
            currentMap:Destroy()
        end
        currentMap = mapChoice:Clone()
        currentMap.Parent = workspace
```

```
    else
        print("Map choice not found")
    end
end

mapPicked.OnServerEvent:Connect(announceMap)
mapPicked.OnServerEvent:Connect(onMapPicked)
```

DisplayMapManager

Location and type: StarterGui > Frame > LocalScript

```
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local nextMap = ReplicatedStorage:WaitForChild("MapPicked")

local frame = script.Parent
local displayMap = frame.DisplayMap

local DISPLAY_DURATION = 4.0

local function onMapPicked(chosenMap)
    displayMap.Text = chosenMap
    frame.Visible = true
    wait(DISPLAY_DURATION)
    frame.Visible = false
end
nextMap.OnClientEvent:Connect(onMapPicked)
```

Hour 13

For this exercise, you were asked to take something you've worked with before—trap parts—and convert them to module scripts. This module could be extended not only to take all of a humanoid's health, but also to create traps that take partial health or even heal the player.

PickupManager

Location and type: ServerStorage > ModuleScript

```
local PickupManager = {}

function PickupManager.modifyHealth(part)
    local character = part.Parent
    local humanoid = character:FindFirstChild("Humanoid")

    if humanoid then
        humanoid.Health = 0
    end
end

return PickupManager
```

OnTouch

Location and type: Part > Script

```
local ServerStorage = game:GetService("ServerStorage")
local PickupManager = require(ServerStorage.PickupManager)

local trap = script.Parent

local function onTouch(part)
    PickupManager.modifyHealth(part)
end

trap.Touched:Connect(onTouch)
```

Hour 14

For this exercise, you were asked to teleport players from one part to another part.

Jump pad starting code

Name: JumpPadManager

Location and type: ServerStorage > ModuleScript

```
local JumpPadManager = {}

-- Local because they're not needed outside of this ModuleScript
local JUMP_DURATION = 1.0
local JUMP_DIRECTION = Vector3.new(0, 6000, 0)

-- Not local because the jump pads need these functions
function JumpPadManager.jump(part)
    local character = part.Parent
    local humanoid = character:FindFirstChildWhichIsA("Humanoid")

    if humanoid then
        local humanoidRootPart = character:FindFirstChild("HumanoidRootPart")
        local vectorForce = humanoidRootPart:FindFirstChild("VectorForce")
        if not vectorForce then
            vectorForce = Instance.new("VectorForce")
            vectorForce.Force = JUMP_DIRECTION
            vectorForce.Attachment0 = humanoidRootPart.RootRigAttachment
            vectorForce.Parent = humanoidRootPart
            wait(JUMP_DURATION)
            vectorForce:Destroy()
        end
    end
end

return JumpPadManager
```

Name: OnTouchManager

Location and type: Part > Script

```
local ServerStorage = game:GetService("ServerStorage")
local JumpPadManager = require(ServerStorage.JumpPadManager)

local jumpPad = script.Parent

local function onTouch(otherPart)
    JumpPadManager.jump(otherPart)
end

jumpPad.Touched:Connect(onTouch)
```

Exercise

For this exercise, you were asked to teleport players from one part to another part.

Set up: Create a part named Origin and a part named Destination.

Script name: OnTouchTeleport

Location and type: Part > Script

```
local ServerStorage = game:GetService("ServerStorage")

local origin = script.Parent
local destination = workspace.Destination

-- Teleports player from origin part to destination part
local function teleportPlayer(otherPart)
    local character = otherPart.Parent
    local humanoid = character:FindFirstChild("Humanoid")
    if humanoid then
        character:SetPrimaryPartCFrame(CFrame.new(destination.Position))
    end
end
origin.Touched:Connect(teleportPlayer)
```

Hour 15

Create a SpotLight that transitions from one color to another indefinitely. This particular solution goes back and forth between the light's original color and the one set in the goal table. The EasingStyle, Bounce, gives a slight flickering effect that would make this same script good for things like campfires.

ScriptName: SpotLightManager

Location and type: Part > Script

Set up: Script assumes there is a SpotLight inside of a part.

```
local TweenService = game:GetService("TweenService")
local lightModel = script.Parent
local spotLight = lightModel:FindFirstChild("SpotLight")

local tweenInfo = TweenInfo.new(
    3.0,
    Enum.EasingStyle.Bounce,
    Enum.EasingDirection.InOut,
    -1,
    true
)

local goal = {}
goal.Color = Color3.fromRGB(255, 0, 255)

local spotLightTween = TweenService:Create(spotLight, tweenInfo, goal)

spotLightTween:Play()
```

Hour 16

Take a dictionary storing how many kills, deaths, and assists participants have in a match and sort the dictionary according to who has the most kills. If tied, prioritize how many assists they have.

Location and type: ServerScriptService > Script

```
-- Example dictionary
local playerKDA = {
    Ana = {kills = 0, deaths = 2, assists = 20},
    Beth = {kills = 7, deaths = 5, assists = 0},
    Cat = {kills = 7, deaths = 0, assists = 5},
    Dani = {kills = 5, deaths = 20, assists = 8},
    Ed = {kills = 1, deaths = 1, assists = 8},
}

-- Insert into array
local sortedKDA = {}

for key, value in pairs(playerKDA) do
    table.insert(sortedKDA, {playerName = key, kills = value.kills, deaths = value.
deaths, assists = value.assists})
end
```

```
-- Print at this point if you need to troubleshoot
print("Original array:")
print(sortedKDA)

-- Sort first by most kills, then by most assists
local function sortByKillsAndAssists(a,b)
    return (a.kills > b.kills) or (a.kills == b.kills and a.assists > b.assists)
end

table.sort(sortedKDA, sortByKillsAndAssists)
print("Sorted array:")
print(sortedKDA)
```

Hour 17

Give people gold each time they join. This version just prints each person's current wealth in the Output window, but the code can be expanded to update a leaderboard or GUI.

Location and type: ServerScriptService > Script

```
local DataStoreService = game:GetService("DataStoreService")
local goldDataStore = DataStoreService:GetDataStore("Gold")
local Players = game:GetService("Players")

local GOLD_ON_JOIN = 5.0

local function onPlayerAdded(player)
    local playerKey = "Player_" .. player.UserId

    -- Use UpdateAsync to get the old value, and send an updated value.
    local updateSuccess, errorMessage = pcall(function ()
        goldDataStore:UpdateAsync(playerKey, function (oldValue)
            local newValue = oldValue or 0
            newValue = newValue + GOLD_ON_JOIN
            return newValue
        end)
    end)

    -- Check to see if there was any errors
    if not updateSuccess then
        print(errorMessage)
    end

    -- Use a pcall() when accessing data as well
    local getSuccess, currentGold = pcall(function()
        return goldDataStore:GetAsync(playerKey)
    end)
```

```
    if getSuccess then
        print(player.Name .. " has " .. currentGold)
    end
end

Players.PlayerAdded:Connect(onPlayerAdded)
```

Hour 18

Add an Announcements module that prints “Round Starting” and “Round Ending” at the beginning and end of the round.

Announcements Module

Location and Type: ServerStorage > ModuleScripts > ModuleScript

```
local Announcements = {}

-- Services
local Players = game:GetService("Players")
local ServerStorage = game:GetService("ServerStorage")

local events = ServerStorage.Events
local roundEnd = events.RoundEnd
local roundStart = events.RoundStart

local function onRoundStart()
    print("Match starting")
end

local function onRoundEnd()
    print("Match over")
end

roundStart.Event:Connect(onRoundStart)
roundEnd.Event:Connect(onRoundEnd)

return Announcements
```

Updated RoundManager

```
-- Services
local ServerStorage = game:GetService("ServerStorage")
local Players = game:GetService("Players")

-- Module Scripts
local moduleScripts = ServerStorage.ModuleScripts
local playerManager = require(moduleScripts.PlayerManager)
```

```
local announcements = require(moduleScripts.Announcements)
local roundSettings = require(moduleScripts.RoundSettings)

-- Events
local events = ServerStorage.Events
local roundStart = events.RoundStart
local roundEnd = events.RoundEnd

while true do
    repeat
        wait(roundSettings.intermissionDuration)
    until Players.NumPlayers >= roundSettings.minimumPeople
    roundStart:Fire()
    wait(roundSettings.roundDuration)
    roundEnd:Fire()
end
```

Hour 19

You were asked to create a pass that could be purchased by users. The actual functionality of your pass will be unique, so no solution is given here.

Hour 20

Create an NPC class that prints out its own name.

```
local Person = {}
    Person.__index = Person

function Person.new(name)
    local self = {}
    setmetatable(self, Person)

    self.name = name

    return self
end

function Person:sayName()
    print("My name is", self.name)
end

local sam = Person.new("Sam")
sam:sayName() -- Prints "My name is Sam"
```

Hour 21

Imagine you're creating an RPG world where you want characters to be able to take on different jobs. Create a parent job class and two different child classes to represent roles people can take on within the game.

```
local Job = {}
Job.__index = Job

function Job.new()
    local self = {}
    setmetatable(self, Job)
    self.experience = 0
    return self
end

function Job:gainExperience(experience)
    self.experience = experience
end

function Job:attack()
    print("I attack the enemy!")
end

local Warrior = {}
Warrior.__index = Warrior
setmetatable(Warrior, Job)

function Warrior.new()
    local self = Job.new()
    setmetatable(self, Warrior)
    self.stamina = 5
    return self
end

function Warrior:attack()
    if self.stamina > 0 then
        print("I swing my weapon at the enemy!")
        self.stamina -= 1
    else
        print("I am too tired to attack")
    end
end

local Mage = {}
Mage.__index = Mage
setmetatable(Mage, Job)
```

```
function Mage.new()
    local self = Job.new()
    setmetatable(self, Mage)
    self.mana = 10
    return self
end

function Mage:attack()
    if self.mana > 0 then
        print("I cast a spell at the enemy!")
        self.mana -= 1
    else
        print("I am out of mana")
    end
end

local warrior = Warrior.new()
print("Warrior experience", warrior.experience)
print("Warrior stamina:", warrior.stamina)
print("Warrior mana:", warrior.mana) -- Should be nil
warrior:attack()
warrior:gainExperience(1)
print("Warrior experience", warrior.experience) -- Should be 1 greater
print("Warrior stamina:", warrior.stamina) -- Should be 1 lower
print("Warrior mana:", warrior.mana) -- Should be nil

local mage = Mage.new()
print("Mage experience:", mage.experience)
print("Mage stamina:", mage.stamina) -- Should be nil
print("Mage mana:", mage.mana)
mage:attack()
mage:gainExperience(1)
print("Mage experience:", mage.experience) -- Should be 1 greater
print("Mage stamina:", mage.stamina) -- Should be nil
print("Mage mana:", mage.mana) -- Should be 1 lower
```

Hour 22

Create a player detector. Use a ray to detect when a player is near a part.

Location and type: Part > Script

```
local Players = game:GetService("Players")

local detector = script.Parent

local DETECTION_RANGE = 20
local DETECTION_INTERVAL = 0.25
```

```lua
local DETECTED_COLOR = Color3.new(0, 1, 0)
local NOT_DETECTED_COLOR = Color3.new(1, 0, 1)

local function findCharacter(character)
    if character then
        local humanoidRootPart = character:FindFirstChild("HumanoidRootPart")
        if humanoidRootPart then
            local toCharacter = humanoidRootPart.Position - detector.Position
            local toCharacterWithRange = toCharacter.Unit * DETECTION_RANGE
            local raycastResult = game.Workspace:Raycast(detector.Position,
                toCharacterWithRange)
            if raycastResult then
                local hitPart = raycastResult.Instance
                if hitPart:IsDescendantOf(character) then
                    return true
                end
            end
        end
    end
    return false
end

local function checkForPlayers()
    for _, player in ipairs(Players:GetPlayers()) do
        local character = player.Character
        if findCharacter(character) then
            return true
        end
    end
    return false
end

detector.Color = NOT_DETECTED_COLOR
while wait(DETECTION_INTERVAL) do
    if checkForPlayers() then
        detector.Color = DETECTED_COLOR
    else
        detector.Color = NOT_DETECTED_COLOR
    end
end
```

Hour 23

You were asked to update the existing code for the Try It Yourself and add a second ploppable object. The code only needs to get up to the point where users can drag a ghost object around.

Location and type: StarterPlayer > StarterPlayerScripts > LocalScript

```
local Players = game:GetService("Players")
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local RunService = game:GetService("RunService")

local player = Players.LocalPlayer
local camera = game.Workspace.Camera
local mouse = player:GetMouse()


local playerGui = player:WaitForChild("PlayerGui")
local plopScreen = playerGui:WaitForChild("ScreenGui")
local plopChairButton = plopScreen:WaitForChild("PlopChairButton")
local plopTableButton = plopScreen:WaitForChild("PlopTableButton")

local ghostObjects = ReplicatedStorage:WaitForChild("GhostObjects")
local ghostChair = ghostObjects:WaitForChild("GhostChair")
local ghostTable = ghostObjects:WaitForChild("GhostTable")
local events = ReplicatedStorage.Events
local plopEvent = events:WaitForChild("PlopEvent")

local raycastParameters = RaycastParams.new()
raycastParameters.FilterType = Enum.RaycastFilterType.Whitelist
raycastParameters.FilterDescendantsInstances = { game.Workspace.Surfaces }

local plopCFrame = nil
local activeGhost = nil

local PLOP_MODE = "PLOP_MODE"
local RAYCAST_DISTANCE = 200

local function onRenderStepped()
    local mouseRay = camera:ScreenPointToRay(mouse.X, mouse.Y, 0)
    local raycastResults = game.Workspace:Raycast(mouseRay.Origin,
        mouseRay.Direction * RAYCAST_DISTANCE, raycastParameters)
    if raycastResults then
        plopCFrame = CFrame.new(raycastResults.Position)
        activeGhost:SetPrimaryPartCFrame(plopCFrame)
        activeGhost.Parent = game.Workspace
    else
        activeGhost.Parent = ReplicatedStorage
    end
end
```

```
local function onPlopButtonActivated()
    plopChairButton.Visible = false
    plopTableButton.Visible = false
    RunService:BindToRenderStep(PLOP_MODE, Enum.RenderPriority.Camera.Value + 1,
onRenderStepped)
end

local function onPlopChairButtonActivated()
    activeGhost = ghostChair
    onPlopButtonActivated()
end

local function onPlopTableButtonActivated()
    activeGhost = ghostTable
    onPlopButtonActivated()
end

plopChairButton.Activated:Connect(onPlopChairButtonActivated)
plopTableButton.Activated:Connect(onPlopTableButtonActivated)
```

Hour 24

Take the plopping code you have so far and add the ability for users to rotate the objects as they place them.

Location and type: StarterPlayer > StarterPlayerScripts > LocalScript

```
local ContextActionService = game:GetService("ContextActionService")
local Players = game:GetService("Players")
local ReplicatedStorage = game:GetService("ReplicatedStorage")
local RunService = game:GetService("RunService")

local player = Players.LocalPlayer
local camera = game.Workspace.Camera
local mouse = player:GetMouse()

local playerGui = player:WaitForChild("PlayerGui")
local plopScreen = playerGui:WaitForChild("ScreenGui")
local plopButton = plopScreen:WaitForChild("PlopButton")

local raycastParameters = RaycastParams.new()
raycastParameters.FilterType = Enum.RaycastFilterType.Whitelist
raycastParameters.FilterDescendantsInstances = { game.Workspace.Surfaces }


local ghostObjects = ReplicatedStorage:WaitForChild("GhostObjects")
local ghostChair = ghostObjects:WaitForChild("GhostChair")
```

```
local events = ReplicatedStorage:WaitForChild("Events")
local plopEvent = events:WaitForChild("PlopEvent")

local plopCFrame = nil
local rotationAngle = 0

local PLOP_CLICK = "PLOP_CLICK"
local PLOP_ROTATE = "PLOP_ROTATE"
local PLOP_MODE = "PLOP_MODE"
local RAYCAST_DISTANCE = 200
local ROTATION_STEP = 45

local function onRenderStepped()
    local mouseRay = camera:ScreenPointToRay(mouse.X, mouse.Y, 0)
    local raycastResults = game.Workspace:Raycast(mouseRay.Origin,
        mouseRay.Direction * RAYCAST_DISTANCE, raycastParameters)
    if raycastResults then
        local rotationAngleRads = math.rad(rotationAngle)
        local rotationCFrame = CFrame.Angles(0, rotationAngleRads, 0)
        plopCFrame = CFrame.new(raycastResults.Position) * rotationCFrame
        ghostChair:SetPrimaryPartCFrame(plopCFrame)
        ghostChair.Parent = game.Workspace
    else
        plopCFrame = nil
        ghostChair.Parent = ReplicatedStorage
    end
end

local function onMouseInput(actionName, inputState)
    if inputState == Enum.UserInputState.End then
        ghostChair.Parent = ReplicatedStorage
        RunService:UnbindFromRenderStep(PLOP_MODE)
        ContextActionService:UnbindAction(PLOP_CLICK)
        ContextActionService:UnbindAction(PLOP_ROTATE)
        plopButton.Visible = true
        rotationAngle = 0
        if plopCFrame then
            plopEvent:FireServer(plopCFrame)
        end
    end
end

local function onRotate(actionName, inputState)
    if inputState == Enum.UserInputState.End then
        rotationAngle += ROTATION_STEP
        if rotationAngle >= 360 then
            rotationAngle -= 360
        end
```

```
    end
end

local function onPlopButtonActivated()
    plopButton.Visible = false
    RunService:BindToRenderStep(PLOP_MODE,
        Enum.RenderPriority.Camera.Value + 1, onRenderStepped)
    ContextActionService:BindAction(PLOP_CLICK, onMouseInput, false,
        Enum.UserInputType.MouseButton1)
    ContextActionService:BindAction(PLOP_ROTATE, onRotate, false, Enum.KeyCode.R)
end

plopButton.Activated:Connect(onPlopButtonActivated)
```

This page intentionally left blank

Index

Symbols

[] (brackets), in key-value pairs, 128-129

: (colon)

accessing functions, 68

for function notation, 281

{} (curly brackets)

for arrays, 113

for dictionaries, 128

. (dot operator)

for dictionary values, 129-130

for embedded objects, 47

object hierarchy and, 18-19

properties and, 20

== (double equal sign) operator, 58

= (equal sign), variable values, 22

>= (greater than or equal to) operator, 59

__index, naming classes, 260

"" (quotation marks), in key-value pairs, 128

3D Editor, 3

3D space

CFrames, 189

offsetting, 191

Position property, 190

rotating with, 191

teleporting exercise, 196-197, 341-342

models, positioning, 192

relative jumps example, 194-195

world versus local coordinates, 193-194

X, Y, Z coordinates, 187-189

A

abstractions, 183-184

accessing

Data Stores, 220

functions, 68

ModuleScripts, 177-178, 182-183

adding
- class functions, 263-268
- class properties, 261-263
- items to arrays, 114
- key-value pairs to dictionaries, 130-132

algorithms
- for sorting
  - alphabetically, 210-211
  - arrays, 210
  - ascending, 210-212
  - descending, 212-213
  - dictionaries, 213-215, 218, 343
  - mixed data types, 212
  - multiple pieces of information, 216-218, 343
  - numerically, 211-212
- purpose of, 209-210

alphabetical sorts, 210-211

anchoring blocks, 10

and operator, 62

animal sounds example (polymorphism), 279-282

animation
- CFrames, LoadCharacter() function versus, 241
- easing, 325
- tweens
  - chaining, 205-206
  - changing colors, 199-200, 208, 342
  - elevator doors example, 202-205
  - setting parameters for, 201-202
  - TweenService, 199

anonymous functions, 52-55, 328

arguments
- definition of, 43
- mismatched, 51-52
- multiple, 45-49
- value types, 86

arithmetic operators, 324

arrays
- adding items, 114
- converting dictionaries to, 213-215
- creating, 113-114
- finding and removing all specific values, 123
- indexes, 113
  - finding from values, 121
  - retrieving specific values, 114-115
- printing with ipairs() function, 115
- purpose of, 113
- removing items, 122
- searching part of, 123-124
- sorting, 210
  - alphabetically, 210-211
  - ascending, 210-212
  - descending, 212-213
  - mixed data types, 212
  - by multiple pieces of information, 216-218, 343
  - numerically, 211-212
- voting simulator, 133-142

ascending sorts, 210-212

assets, organizing, 231-234

assigning variable values, 41

attributes, 64-67
- checking values, 85
- code reusability and, 79

autocomplete feature, 20

B

Baseplate template, 3

BindableEvents, 230

BindAction() function, 314

BindToRenderStep() function, 303-305

blacklists versus whitelists in raycasting, 310

blocks, anchoring, 10

boolean data type, 22, 36

brackets ([]), in key-value pairs, 128-129

break keyword, 110

bridges
- reactivating, 38-40
- solidifying, 42, 328
- vanishing, 34-36

burning fire, 93-97

buttons
- for placing objects, creating, 302-303
- testing, 170
- viewing/hiding, 320

buying items. *See* monetization; Robux

C

calling functions, 32
- with events, 33-36
- parent functions, 282

camera, moving, 4, 321

camouflage raycasting example, 288-289

car class example
- adding properties, 262-263
- property inheritance, 275-277

case-sensitivity of keywords, 19

cashing out Robux, 243

CFrame.Angles() function, 191

CFrames, 189
- LoadCharacter() function versus, 241
- offsetting, 191
- Position property, 190
- rotating with, 191
- teleporting exercise, 196-197, 341-342

chaining tweens, 205-206

changing
- gravity, 233
- properties, 25

changing seasons exercise, 125-126, 334

child classes, 271-272
- calling parent functions, 282
- function inheritance, 278
- inheritance setup, 272-274
- multiple, 277
- polymorphism, 278-282
- property inheritance, 274-277

child objects, 18
- searching, 223

classes. ***See also*** **child classes; parent classes**
- calling parent functions, 282
- creating, 260-261, 270, 346
- functions of, 263-268
- inheritance, 271-272
  - of functions, 278
  - job roles exercise, 285, 347
  - multiple child classes, 277
  - of properties, 274-277
  - setup, 272-274
- naming, 260
- polymorphism, 278-282
- properties of, 261-263
- purpose of, 259

clients, 145
- GUIs. *See* GUIs
- RemoteEvent object, 161-162
  - client-to-client communication, 171
  - client-to-server communication, 165-170
  - server-to-all-clients communication, 162-165
  - server-to-single-client communication, 170-171
- RemoteFunction object, 149-151
- server/client divide, 149
- store purchases, 151-158

cloning particle emitters, 100, 330-332

code organization with OOP, 259

collecting firewood, 100, 330-332

colon (:)
- accessing functions, 68
- for function notation, 281

color picker, 25

colors, changing, 25, 199-200, 208, 342

comments, 12

concatenation, 23

concatenation operator, 325

conditional structures, 57
- elseif keyword, 62
- else keyword, 63
- if/then statements, 58-59
- portals, creating, 63-70

configuring passes, 246-249

connect() function, 33

constants, 84

constructors, 260, 265

ContextActionService, 314-316, 320

control variables in for loops, 103, 111

converting dictionaries to arrays, 213-215

coordinates in 3D space
- CFrames, 189
  - offsetting, 191
  - Position property, 190
  - rotating with, 191
  - teleporting exercise, 196-197, 341-342
- relative jumps example, 194-195
- world versus local, 193-194
- X, Y, Z coordinates, 187-189

copying meshes, 78

countdowns, creating with RemoteEvent object, 163-165

crown sales example, 248-255

curly brackets ({})

- for arrays, 113
- for dictionaries, 128

custom leaderboards, 87

D

Damage Over Time (DoT), 111-112, 333

dance floor, creating, 92-93

Data Stores

- accessing, 220
- creating, 220
- enabling, 219
- limiting network calls, 225
- unique key names, 224
- updating, 220-228, 344

data types, 22, 27

- in Lua, 322
- in Roblox Studio, 323

debouncing

- Humanoid objects, 73-75, 88-89, 330
- with ProximityPrompts, 78-79

debugging

- argument value types, 86
- attribute values, 85
- exercise, 88, 329
- string debugging, 82-84
- variable order and placement, 84

decals, inserting, 28-29, 327

descending sorts, 212-213

descriptions, 255

destroy() function, 18-19

detecting mouse input, 314-316

detector exercise, 295, 348

Developer Exchange Program, 243

Developer Products, 256

dictionaries

- converting to arrays, 213-215
- creating, 128
- key-value pairs, 128
  - adding, 130-132
  - formatting keys, 128-129
  - removing, 130-131
  - unique keys, 130
  - value usage, 129-130
- pairs() function, 132-133
- purpose of, 127-128
- sorting, 213-215, 218, 343
- voting simulator, 133-142

direction parameter for raycasting, 289-290

distance, limiting for raycasting, 293

doors, creating for elevator, 202-205

DoT (Damage Over Time), 111-112, 333

dot operator (.)

- for dictionary values, 129-130
- for embedded objects, 47
- object hierarchy and, 18-19
- properties and, 20

double equal sign (==) operator, 58

doubling and halving variables, 85

DRY coding. ***See also*** **OOP**

- abstractions, 183-184
- purpose of, 183

E

easing in animation, 325

elevator doors, creating, 202-205

else keyword, 63

elseif keyword, 62

embedded objects, finding in hierarchy, 47

enabling Data Stores, 219

end value in for loops, 103

engagement payouts, 256

equal sign (=), variable values, 22

error messages, 11-12

errors

- list of, 228
- string debugging, 82-84

event connections, order and placement, 138

events

- BindableEvents, 230
- calling functions, 33-36
- RemoteEvent object, 161-162
  - client-to-client communication, 171
  - client-to-server communication, 165-170
  - server-to-all-clients communication, 162-165
  - server-to-single-client communication, 170-171
- Touched, 34-35

exercises
- animating color changes, 208, 342
- anonymous functions, 55, 328
- changing player speed, 72, 328
- changing seasons, 125-126, 334
- cloned particle emitters, 100, 330-332
- collecting firewood, 100, 330-332
- creating NPCs, 29, 327
- debouncing, 88-89, 330
- debugging, 88, 329
- detector with raycasting, 295, 348
- dictionary sorting, 218, 343
- DoT (Damage Over Time), 111-112, 333
- inserting decals, 28-29, 327
- job roles, 285, 347
- loops, 112, 333-334
- map choice announcement, 172, 338-340
- NPC person class, 270, 346
- obstacle course, 14-15, 326
- pass creation, 257, 346
- placing objects, 311, 350
- player announcements, 242, 345
- price lists, 160, 336-338
- rotating objects, 320, 351
- solidifying bridges, 42, 328
- solutions to, 326-351
- team assignments, 143, 334
- teleporting with CFrames, 196-197, 341-342
- traps with ModuleScripts, 185, 340-341
- updating player information, 228, 344

Explorer window, 3

explosion script, 9-11

F

false conditions, loops for, 98

files, saving, 13

filtering
- lists, 7
- objects for raycasting, 294

finding
- all specific array values, 123
- array indexes from values, 121
- embedded objects in hierarchy, 47
- list items, 7

fire
- burning, 93-97
- collecting firewood, 100, 330-332

folders, modifying items
- with for loops, 116-121
- with ipairs() function, 116

for loops, 98, 101-102
- default increment, 105
- examples and exercises, 105-106, 112, 333-334
- finding and removing all specific array values, 123
- generic, 115
- *i* in, 111
- numeric, 123-124
- printing arrays, 115
- searching part of arrays, 123-124
- turning lights on/off, 116-121
- values in, 102-105

formatting dictionary keys, 128-129

functions
- accessing, 68
- anonymous, 52-55, 328
- arguments
  - definition of, 43
  - mismatched, 51-52
  - multiple, 45-49
  - value types, 86
- BindAction(), 314
- BindToRenderStep(), 303-305
- calling, 32
  - with events, 33-36
  - from parent classes, 282
- CFrame.Angles(), 191
- of classes, 263-268
- connect(), 33
- constructors, 260, 265
- creating, 31-32
- definition of, 31
- destroy(), 18-19
- GetAsync(), 220, 225
- IncrementAsync(), 227
- inheriting, 278
- insert(), 114

ipairs()
  finding array indexes, 121
  with folders, 116
  pairs versus, 142
  printing arrays, 115
LoadCharacter(), 241
as methods, 33
in ModuleScripts
  accessing, 177-178
  adding, 175-176
  scope, 176
MoveTo(), 264
multiple in scripts, 41
named, 52-55, 328
naming conventions, 32, 35, 69
new(), 26
order and placement, 36-40
paint(), 44-48
pairs()
  with dictionaries, 132-133
  ipairs() versus, 142
parameters
  creating, 43-45
  definition of, 43
  maximum, 54
  mismatched, 51-52
  multiple, 45-49
pcall(), 225
polymorphism, 278-282
print(), 7-9, 23, 43
  for debugging, 82-84
RemoteFunction object, 149-151, 159
remove(), 122
require(), 177
return values
  definition of, 49
  multiple, 50, 80
  nil, 51
scope, 33, 37-38
SetAsync(), 220, 225
table.sort(), 210-213
tostring(), 212
UnbindAction(), 314
UpdateAsync(), 226-227
wait(), 42-43, 201
  default value, 86
  with while loops, 92-93
workspace:Raycast()
  camouflage example, 288-289
  direction parameter, 289-290
  limiting distance, 293
  setup, 287-288

G

game loops
  BindableEvents in, 230
  creating, 231-240
  for player announcements, 242, 345
  purpose of, 229-230
gameplay, moving camera in, 321
generic for loops, 115
GetAsync() function, 220, 225
global coordinates, local versus, 193-194
global variables, 22, 41
glowing lights, 120
goal value in for loops, 103
gold ore script (mining simulator), 79-82
gold ore setup (mining simulator), 78-79
graphical user interfaces. *See* GUIs
gravity, changing, 233
greater than or equal to (>=) operator, 59
grouping parts, 166, 192
GUIs (graphical user interfaces)
  creating, 106-109, 146-148, 335
  customizing, 147
  moving, 154
  purpose of, 146
  script placement, 148

H

Hello World! script, 7-9
hiding buttons, 320
hierarchy (of objects), 18
  finding embedded objects, 47
  instances, 26
  IntValue objects, 77
  naming conventions, 24
  properties, 20-22
    changing, 25
    data types for, 22, 27
  variables and, 28

Humanoid objects, 59-61
- changing player speed, 72, 328
- debouncing, 73-75, 88-89, 330
- VectorForce objects, adding, 179-182

HumanoidRootPart, MoveTo() function and, 264

I

***i* as control variable, 111**

if/then statements, 58-59

ignoring objects in raycasting, 290-293

IncrementAsync() function, 227

increment value in for loops, 103-105

indenting code, 32

indexes, 113
- finding from values, 121
- key-value pairs versus, 129
- retrieving specific values, 114-115

in-game purchases. *See* monetization; Robux

inheritance, 271-272
- of functions, 278
- job roles exercise, 285, 347
- multiple child classes, 277
- overriding, 278-282
- of properties, 274-277
- setup, 272-274

insert() function, 114

inserting
- decals, 28-29, 327
- scripts into parts, 6-7

installing Roblox Studio, 1-2

instances, 26

IntValue objects, 77

ipairs() function
- finding array indexes, 121
- with folders, 116
- pairs() versus, 142
- printing arrays, 115

iterations, 105

J–K

job roles exercise, 285, 347

jump pads
- creating, 178-183
- relative jumps with, 194-195

keys
- for moving camera, 321
- uniqueness in Data Stores, 224

key-value pairs, 128
- adding, 130-132
- in Data Stores, accessing, 220
- formatting keys, 128-129
- indexes versus, 129
- removing, 130-131
- unique keys, 130
- value usage, 129-130

keywords, 19-20
- break, 110
- case-sensitivity, 19
- else, 63
- elseif, 62
- nil, 51
- reserved names, 322
- return, 49-50
- script, 20
- type, 217
- workspace, 19

L

leaderboards
- creating, 75-77, 87
- maximum number of stats, 87
- value types, 86-87

leaderstats folder, 77

length operator, 325

lights
- colors, changing via tweens, 208, 342
- glowing, 120
- SpotLight objects, 117
- turning on/off, 116-121

limiting
- distance for raycasting, 293
- network calls, 225

lists, filtering, 7

LoadCharacter() function, 241

load times for scripts, 109

local object coordinates, world versus, 193-194

LocalScript object, 148, 154-155

local variables, 22, 184

logging in to Roblox Studio, 2

logical operators, 62-63, 324

loops

- break keyword, 110
- exercises, 112, 333-334
- for false conditions, 98
- for, 98, 101-102
  - default increment, 105
  - examples, 105-106
  - finding and removing all specific array values, 123
  - generic, 115
  - *i* in, 111
  - numeric, 123-124
  - printing arrays, 115
  - searching part of arrays, 123-124
  - turning lights on/off, 116-121
  - values in, 102-105
- game loops
  - BindableEvents in, 230
  - creating, 231-240
  - for player announcements, 242, 345
  - purpose of, 229-230
- nested, 109-110
- repeat until, 237
- while, 91-92
  - with ProximityPrompts, 93-97
  - scope, 98
  - with wait() function, 92-93

Lua, 1

- arrays
  - adding items, 114
  - converting dictionaries to, 213-215
  - creating, 113-114
  - finding and removing all specific values, 123
  - indexes, 113-115, 121
  - printing with ipairs() function, 115
  - purpose of, 113
  - removing items, 122
  - searching part of, 123-124
  - sorting, 210-213, 216-218, 343
  - voting simulator, 133-142
- classes. *See also* child classes; parent classes
  - calling parent functions, 282
  - creating, 260-261, 270, 346
  - function inheritance, 278
  - functions of, 263-268
  - inheritance, 271-274, 285, 347
  - multiple child classes, 277
  - naming, 260
  - polymorphism, 278-282
  - properties of, 261-263
  - property inheritance, 274-277
- conditional structures, 57
  - elseif keyword, 62
  - else keyword, 63
  - if/then statements, 58-59
  - portals, creating, 63-70
- data types, 22, 27, 322
- debugging
  - argument value types, 86
  - attribute values, 85
  - exercise, 88, 329
  - string debugging, 82-84
  - variable order and placement, 84
- dot operator
  - for dictionary values, 129-130
  - for embedded objects, 47
  - object hierarchy and, 18-19
  - properties and, 20
- functions. *See* functions
- keywords, 19-20
- loops
  - break keyword, 110
  - exercises, 112, 333-334
  - for false conditions, 98
  - for, 98, 101-106, 111, 115-124
  - game loops, 229-242, 345
  - nested, 109-110
  - repeat until, 237
  - while, 91-98
- ModuleScripts
  - accessing in scripts, 177-178, 182-183
  - code structure, 174
  - creating, 234-237
  - DRY coding and, 183
  - functions and variables in, 175-176
  - jump pad example, 179-182

naming, 174-175
placing, 174
purpose of, 173
scope in, 176
trap exercise, 185, 340-341
naming conventions, list of, 325
object hierarchy, 18
finding embedded objects, 47
instances, 26
IntValue objects, 77
naming conventions, 24, 260
properties, 20-22, 25, 27
variables and, 28
operators
arithmetic, 324
concatenation, 325
double equal sign (==), 58
greater than or equal to (>=), 59
length, 325
logical, 62-63, 324
most common, 58
purpose of, 324
relational, 324
reserved names, 322
scripts, 6
autocomplete feature, 20
comments, 12
DRY coding, 183-184
error messages, 11-12
explosion example, 9-11
GUI script placement, 148
Hello World!, 7-9
indenting code, 32
inserting into parts, 6-7
load times, 109
for mining simulator, 79-82
multiple functions in, 41
opening, 13
order and placement in, 36-40
renaming, 18-19
saving, 13
strings, 7
variables
combining with strings, 23
creating, 22-25
naming conventions, 24
properties and, 28
updating, 23

M

map choice announcement exercise, 172, 338-340
map pickers, creating, 166-170
meshes, copying, 78
messages
receiving on server, 317-319
sending to server, 316
methods. *See* functions
mining simulator, 75
gold ore script, 79-82
gold ore setup, 78-79
leaderboard, creating, 75-77
mismatched arguments/ parameters, 51-52
mixed data types, sorting, 212
models
creating, 192
grouping parts into, 166
positioning, 192
modifying folder items
with for loops, 116-121
with ipairs(), 116
ModuleScripts
accessing in scripts, 177-178, 182-183
code structure, 174
creating, 234-237
DRY coding and, 183
functions and variables in, 175-176
jump pad example, 179-182
naming, 174-175
placing, 174
purpose of, 173
scope in, 176
trap exercise, 185, 340-341
monetization. *See also* Robux
Developer Products, 256
engagement payouts, 256
ideas for, 256
passes
checking for ownership, 252-255
configuring, 246-249
creating, 244-245, 257, 346
crown sales example, 248-255
prompting purchases, 247-250
purpose of, 243
testing, 251-252
updating, 245

mouse input, detecting, 314-316
mouse movements, tracking, 303-306
BindToRenderStep() function, 303-305
raycasting from mouse, 305-306
MoveTo() function, 264
moving
camera, 4, 321
GUIs, 154
multiple arguments, 45-49
multiple child classes, 277
multiple functions in scripts, 41
multiple parameters, 45-49
multiple pieces of information, sorting by, 216-218, 343
multiple player interactions, variables for, 70
multiple players, testing for, 138-139
multiple return values, 80

N

named functions, 52-55, 328
naming
classes, 260
ModuleScripts, 174-175
objects, 260
naming conventions, 24, 32, 35, 69
constants, 84
constructors, 265
list of, 325
nested loops, 109-110
network calls, 225
Network Simulator, testing for multiple people, 138-139
new() function, 26
nil keyword, 51, 157
not operator, 62
NPCs (Non Playable Characters), 17
adding face to, 28-29, 327
creating, 23-25, 29, 327
exercise, 270, 346
number data type, 22
numbers, sorting with strings, 212
numeric for loops, 123-124
numerical sorts, 211-212

O

object hierarchy, 18
finding embedded objects, 47
instances, 26
IntValue objects, 77
naming conventions, 24
properties, 20-22
changing, 25
data types for, 22, 27
variables and, 28
object-oriented programming. *See* OOP (object-oriented programming)
objects
filtering for raycasting, 294
ignoring in raycasting, 290-293
naming, 260
placing, 297-298, 313. *See also* 3D space
creating button for, 302-303
detecting mouse input, 314-316
with other object coordinates, 190
previewing placement, 307-309
receiving messages on server, 317-319
second object exercise, 311, 350
sending messages to server, 316
setup, 298-301
tracking mouse movements, 303-306
purpose of, 259
rotating
with CFrames, 191
while placing, 320, 351
obstacle course exercise, 14-15, 326
offsetting CFrames, 191
one-time purchases. *See* passes
OOP (object-oriented programming)
classes. *See also* child classes; parent classes
calling parent functions, 282
creating, 260-261, 270, 346
function inheritance, 278
functions of, 263-268
inheritance, 271-274, 285, 347

multiple child classes, 277
naming, 260
polymorphism, 278-282
properties of, 261-263
property inheritance, 274-277
purpose of, 259
code organization with, 259
objects
filtering for raycasting, 294
ignoring in raycasting, 290-293
naming, 260
placing. *See* placing, objects
purpose of, 259
rotating, 191, 320, 351
opening
Output window, 5
scripts, 13
operating system requirements, 13
operators
arithmetic, 324
concatenation, 325
double equal sign (==), 58
greater than or equal to (>=), 59
length, 325
logical, 62-63, 324
most common, 58
purpose of, 324
relational, 324
organizing
assets, 231-234
with OOP, 259
variables, 305
or operator, 62
Output window, opening, 5
overriding inheritance, 278-282

P

paint() function, 44-48
pairs() function
with dictionaries, 132-133
ipairs() versus, 142
parameters
creating, 43-45
definition of, 43
maximum, 54
mismatched, 51-52
multiple, 45-49
setting for raycasting, 290-293
for tweens, 201-202
parent classes, 271-272
calling parent functions from child classes, 282
inheritance setup, 272-274
parent objects, 18
ParticleEmitter objects, 72
particle emitters, cloning, 100, 330-332
parts, 6
colors, changing via tweens, 199-200
creating instances, 26
grouping, 166, 192
initial location, 26
inserting scripts into, 6-7
ProximityPrompts for, 78-79
textures, showing activation, 67
Touched event, 34-35
passes
checking for ownership, 252-255
configuring, 246-249
creating, 244-245, 257, 346
crown sales example, 248-255
prompting purchases, 247-250
purpose of, 243
testing, 251-252
updating, 245
pcall() function, 225
pet class example, adding functions, 264-268
placing
models, 192
ModuleScripts, 174
objects, 297-298, 313. *See also* 3D space
creating button for, 302-303
detecting mouse input, 314-316
with other object coordinates, 190
previewing placement, 307-309
receiving messages on server, 317-319
rotating while, 320, 351
second object exercise, 311, 350
sending messages to server, 316

setup, 298-301
tracking mouse movements, 303-306

player announcements exercise, 242, 345

playerID, saving data with, 226

player management, services for, 237-240

playtesting
- changes during, 12
- for multiple players, 138-139
- references, checking, 165
- scripts, 7-8

polymorphism, 278-282

portals, creating, 63-70

positioning. *See* placing

Position property (CFrames), 190

previewing object placement, 307-309

price list exercise, 160, 336-338

PrimaryParts (models), 192

print() function, 7-9, 23, 43
- for debugging, 82-84

printing arrays with ipairs() function, 115

prompting in-game purchases, 247-250

properties, 20-22
- changing, 25
- of classes, 261-263
- data types for, 22, 27
- inheriting, 274-277
- variables and, 28

Properties window, 3

protected calls, 225

ProximityPrompt objects, 64, 67-70
- debouncing with, 78-79
- with ServerScriptService, 79-82
- viewing, 268
- with while loops, 93-97

ProximityPromptService, 68-70

purchases. *See* monetization; Robux

Q–R

quotation marks ("") in key-value pairs, 128

raycasting
- camouflage example, 288-289
- detector exercise, 295, 348
- direction parameter, 289-290
- filtering objects, 294
- from mouse, 305-306
- function setup, 287-288
- limiting distance, 293
- purpose of, 287
- setting parameters, 290-293
- through windows, 292-293
- whitelists versus blacklists, 310

reactivating bridges, 38-40

receiving messages on server, 317-319

red lines in editor, 11-12

references, checking, 165

relational operators, 324

relative jumps, creating, 194-195

RemoteEvent object, 161-162
- client-to-client communication, 171
- client-to-server communication, 165-170
- server-to-all-clients communication, 162-165
- server-to-single-client communication, 170-171

RemoteFunction object, 149-151, 159

remove() function, 122

removing
- all specific array values, 123
- items from arrays, 122
- key-value pairs from dictionaries, 130-131

renaming scripts, 18-19

renderstep, 303-305

repeat until loops, 237

require() function, 177

reserved names, 322

resources for information, 13, 319

retrieving specific array values, 114-115

return keyword, 49-50

return values
- definition of, 49
- multiple, 50, 80
- nil, 51

returning table values, 133

Roblox Premium
- engagement payouts, 256
- monetization and, 247

Roblox Studio, 1
- blocks, anchoring, 10
- camera controls, 4

data types, 323
files, saving, 13
GUIs
  creating, 106-109, 146-148, 335
  customizing, 147
  moving, 154
  purpose of, 146
  script placement, 148
Humanoid objects, 59-61
  changing player speed, 72, 328
  debouncing, 73-75, 88-89, 330
  VectorForce objects, adding, 179-182
installing, 1-2
leaderboards
  creating, 75-77, 87
  maximum number of stats, 87
  value types, 86-87
leaderstats folder, 77
logging in, 2
moving camera in, 321
object hierarchy, 18
  finding embedded objects, 47
  instances, 26
  IntValue objects, 77
  naming conventions, 24
  properties, 20-22, 25-27
  variables and, 28
operating system requirements, 13
Output window, opening, 5
parts, 6
  colors, changing via tweens, 199-200
  creating instances, 26
  grouping, 166, 192
  initial location, 26
  inserting scripts into, 6-7
  ProximityPrompts for, 78-79
  textures, showing activation, 67
  Touched event, 34-35
red lines in editor, 11-12
resources for information, 13, 319
user interface, 2-4
Robux. *See also* monetization
  cashing out, 243
  engagement payouts, 256
  uses for, 243
rotating objects, 320, 351
  with CFrames, 191
Run command, testing code, 48

S

saving data
  in Data Stores
    accessing, 220
    creating, 220
    enabling, 219
    limiting network calls, 225
    unique key names, 224
    updating, 220-228, 344
  methods of, 227
  with playerID, 226
saving scripts, 13
scope
  of functions, 33, 37-38
  in ModuleScripts, 176
  of variables, 41
  of while loops, 98
ScreenGui object, 146-147
script keyword, 20
scripts, 6
  arrays
    adding items, 114
    converting dictionaries to, 213-215
    creating, 113-114
    finding and removing all specific values, 123
    indexes, 113-115, 121
    printing with ipairs() function, 115
    purpose of, 113
    removing items, 122
    searching part of, 123-124
    sorting, 210-213, 216-218, 343
    voting simulator, 133-142
  autocomplete feature, 20
  comments, 12
  conditional structures, 57
    elseif keyword, 62
    else keyword, 63
    if/then statements, 58-59
    portals, creating, 63-70
  DRY coding
    abstractions, 183-184
    purpose of, 183
  error messages, 11-12

explosion example, 9-11
functions. *See* functions
GUI script placement, 148
Hello World!, 7-9
indenting code, 32
inserting into parts, 6-7
load times, 109
loops
- break keyword, 110
- exercises, 112, 333-334
- for false conditions, 98
- for, 98, 101-106, 111, 115-124
- game loops, 229-242, 345
- nested, 109-110
- repeat until, 237
- while, 91-98

for mining simulator, 79-82
ModuleScripts
- accessing, 177-178, 182-183
- code structure, 174
- creating, 234-237
- functions and variables in, 175-176
- jump pad example, 179-182
- naming, 174-175
- placing, 174
- purpose of, 173
- scope in, 176
- trap exercise, 185, 340-341

multiple functions in, 41
opening, 13
order and placement in, 36-40
renaming, 18-19
saving, 13
ServerScriptService, 156-158

searching
- child objects, 223
- part of arrays, 123-124

seasons, changing, 125-126, 334
self, as naming convention, 260
sending messages to server, 316
servers, 145
- receiving messages, 317-319
- RemoteEvent object, 161-162
  - client-to-server communication, 165-170
  - server-to-all-clients communication, 162-165
  - serve-to-single-client communication, 170-171
- RemoteFunction object, 149-151
- sending messages to, 316
- server/client divide, 149
- store purchases, 151-158

ServerScriptService, 76, 79-82, 156-158
services
- ContextActionService, 314-316, 320
- definition of, 68
- player management, 237-240
- ProximityPromptService, 68-70
- ServerScriptService, 76, 79-82, 156-158

SetAsync() function, 220, 225
solidifying bridges, 42, 328
solutions to exercises, 326-351
- animating color changes, 342
- anonymous functions, 328
- changing player speed, 328
- changing seasons, 334
- cloned particle emitters, 330-332
- collecting firewood, 330-332
- creating NPCs, 327
- debouncing, 330
- debugging, 329
- detector with raycasting, 348
- dictionary sorting, 343
- DoT (Damage Over Time), 333
- inserting decals, 327
- job roles, 347
- loops, 333-334
- map choice announcement, 338-340
- NPC person class, 346
- obstacle course, 326
- pass creation, 346
- placing objects, 350
- player announcements, 345
- price lists, 336-338
- rotating objects, 351
- solidifying bridges, 328
- team assignments, 334
- teleporting with CFrames, 341-342
- traps with ModuleScripts, 340-341
- updating player information, 344

sorting
- arrays, 210
  - alphabetically, 210-211
  - ascending, 210-212
  - descending, 212-213

mixed data types, 212
by multiple pieces of information, 216-218, 343
numerically, 211-212
dictionaries, 213-215, 218, 343

SpeedBoost tweaks, 85

speed of players, changing, 72, 328

SpotLight objects, 117

StarterGUI object, 146

storage

for BindableEvents, 230
for ModuleScripts, 174

store purchases, 151-158

string debugging, 82-84

strings, 7, 22

combining with variables, 23
sorting with numbers, 212

Studio. *See* Roblox Studio

SurfaceGui objects, 106-108

T

tables, 22

arrays
adding items, 114
converting dictionaries to, 213-215
creating, 113-114
finding and removing all specific values, 123
indexes, 113-115, 121
printing with ipairs() function, 115
purpose of, 113
removing items, 122
searching part of, 123-124
sorting, 210-213, 216-218, 343
voting simulator, 133-142
dictionaries
converting to arrays, 213-215
creating, 128
key-value pairs, 128-132
pairs() function, 132-133
purpose of, 127-128
sorting, 213-215, 218, 343
voting simulator, 133-142
purpose of, 113
returning values, 133

table.sort() function, 210-213

team assignments exercise, 143, 334

teleporting exercise, 196-197, 341-342

templates, Baseplate, 3

testing

buttons, 170
changes during, 12
for multiple players, 138-139
passes, 251-252
references, checking, 165
with Run command, 48
scripts, 7-8

TextLabel objects, 106-109

textured parts, showing activation, 67

Toolbar ribbon, 3

Toolbox, 3

tostring() function, 212

Touched event, 34-35

tracking mouse movements, 303-306

BindToRenderStep() function, 303-305
raycasting from mouse, 305-306

traps exercise, 185, 340-341

turning lights on/off, 116-121

tweens

chaining, 205-206
changing colors, 199-200, 208, 342
elevator doors example, 202-205
setting parameters for, 201-202
TweenService, 199

TweenService, 199

type keyword, 217

U

UnbindAction() function, 314

unique keys

in Data Stores, 224
in dictionaries, 130

UpdateAsync() function, 226-227

updating

Data Stores, 220-228, 344
passes, 245
variables, 23

user interface for Roblox Studio, 2-4

V

vanishing bridges, 34-36
variables
combining with strings, 23
creating, 22-25
without assigning value, 41
doubling and halving, 85
in for loops, 102-105
local, 184
in ModuleScripts
accessing, 177-178
adding, 175-176
scope, 176
for multiple player interactions, 70
naming conventions, 24
order and placement, 36-40, 45, 84
organizing, 305
properties and, 28
scope, 41
updating, 23
VectorForce objects, adding to humanoids, 179-182
vehicle class example (property inheritance), 275-277
viewing
buttons, 320
ProximityPrompts, 268
voting simulator, 133-142

W

wait() function, 42-43, 201
default value, 86
with while loops, 92-93
WET coding, 183
while loops, 91-92
exercises, 112, 333-334
with ProximityPrompts, 93-97
scope, 98
with wait() function, 92-93
whitelists versus blacklists in raycasting, 310
windows
raycasting through, 292-293
in Roblox Studio, opening, 5
workspace keyword, 19
workspace:Raycast() function
camouflage example, 288-289
direction parameter, 289-290
limiting distance, 293
setup, 287-288
world coordinates, local versus, 193-194
writing scripts, 7-9

X–Z

X coordinates, 187-189
Y coordinates, 187-189
Z coordinates, 187-189

Official Roblox Programming Titles

- Learn the basics of Lua programming using Roblox's free development tools.
- Designed for anyone—even beginners—to learn how to build 3D models, code scripts, and more on Roblox in 24 hour-long lessons. Books include step-by-step instructions; Q&As, Quizzes, and Exercises to build and test knowledge; and "Try It Yourself" exercises that encourage independent practice.
- Developed with and supported by Roblox Education. Be part of an ever-growing global community of creators and discover how to build your own 3D experiences on Roblox from scratch.

Roblox Game Development in 24 Hours: The Official Roblox Guide
ISBN: 978-0-13-682973-7

Coding with Roblox Lua in 24 Hours: The Official Roblox Guide
ISBN: 978-0-13-682942-3

Pearson

Photo by izusek/gettyimages

Register Your Product at informit.com/register

Access additional benefits and **save 35%** on your next purchase

- Automatically receive a coupon for 35% off your next purchase, valid for 30 days. Look for your code in your InformIT cart or the Manage Codes section of your account page.
- Download available product updates.
- Access bonus material if available.*
- Check the box to hear from us and receive exclusive offers on new editions and related products.

**Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.*

InformIT.com—The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's foremost education company. At InformIT.com, you can:

- Shop our books, eBooks, software, and video training
- Take advantage of our special offers and promotions (informit.com/promotions)
- Sign up for special offers and content newsletter (informit.com/newsletters)
- Access thousands of free chapters and video lessons

Connect with InformIT—Visit informit.com/community

informIT®
the trusted technology learning source

Addison-Wesley • Adobe Press • Cisco Press • Microsoft Press • Pearson IT Certification • Que • Sams • Peachpit Press