3

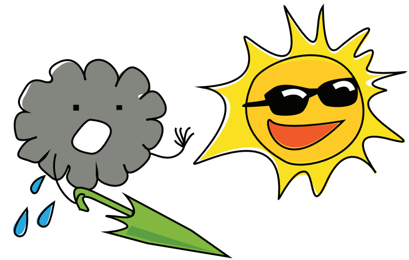
MAKING CHOICES

Now that we've covered how to create constants and variables, you're ready to learn how to tell your computer to make choices. This chapter is about controlling the flow of a computer program by telling the computer which path to take. When we talk about *flow*, we're referring to the order in which the statements of the program are executed.

Up to this point, you've only seen statements performed in the order you've typed them. You've done some cool things with this, but by telling the computer how to make choices about the order of executing

statements, you can do even more. To get the computer to make a choice, we'll use *conditional statements* that tell the computer to run some code based on a condition's value.

You already use conditional statements to make choices every day! For example, before you leave home in the morning, you check the weather. If it's sunny, you may put on a pair of sunglasses. If it's raining, you grab your umbrella. In each case, you're checking a condition. If the condition "it is raining" is true, then you take your umbrella when you leave the house. When the condition could be true or false, it's called a *Boolean expression*. The `Bool` data type that you learned about in Chapter 2 is used to represent the value `true` or `false`.



BOOLEAN EXPRESSIONS

A common type of Boolean expression is one that compares two values using a *comparison operator*. There are six comparison operators. Let's start with two simple ones: *is equal* and *is not equal*.

IS EQUAL AND IS NOT EQUAL

You'll use the *is equal* and *is not equal* comparison operators a lot. *Is equal* is written with two equal signs next to each other, like this: `==`. *Is not equal* is written with an exclamation mark and one equal sign, like this: `!=`.

Let's try them both out in the playground!

```
❶ 3 + 2 == 5                                  true
   4 + 5 == 8                                  false
❷ 3 != 5                                      true
   4 + 5 != 8                                  true
   // This is wrong and will give you an error
❸ 3 + 5 = 8                                   error
```

In plain English, the line at ❶ says, "three plus two equals five," which is a `true` statement, and the output in the right pane will confirm this as soon as you finish typing it. At ❷, the line says, "three is not equal to five," which is also a `true` statement. Note that ❸ is an error. Do you know why? While `=` and `==` look a lot alike, remember that a single equal sign (`=`) assigns values. That statement reads, "Put the value of 8 into something called 3 + 5," which doesn't work.

In Swift, the `==` operator also works with other data types, not just numbers. Let's try making some other comparisons.

```
  // Comparing strings
  let myName = "Gloria"                                "Gloria"
  myName == "Melissa"                                  false
  myName == "Gloria"                                   true
❶ myName == "gloria"                                   false
  var myHeight = 67.5                                  67.5
  myHeight == 67.5                                     true
  // This is wrong and will give you an error
❷ myHeight == myName                                   error
```

The line at ❶ is a tricky one; did you expect it to be `true`? Those two strings are close but not exactly the same, and an *is equal* comparison is `true` only if the two values match exactly. The constant `myName` has a value of `"Gloria"` with a capital *G*, which is not the same as `"gloria"` with a lowercase *g*.

Remember in Chapter 2 when we said that you can't use math operators like `+` and `*` on things that aren't the same data type? The same is true for comparisons. You can't compare things that are different types. The line at ❷ will cause an error because one is a `String` and the other is a `Double`.

GREATER THAN AND LESS THAN

Now let's look at four other comparison operators. We'll start with *greater than* (written as `>`) and *less than* (written as `<`). You probably already have a good idea of how these work. A Boolean expression like `9 > 7`, which reads "9 is greater than 7," is `true`. Often, you'll also want to know if something is greater than or equal to something or less than or equal to something. There are two more operators that cover those cases: *greater than or equal to* (which looks like `>=`) and *less than or equal to* (which looks like `<=`). Let's try these out with some more examples:

```
  // Greater than
  9 > 7                               true
  // Less than
  9 < 11                              true
  // Greater than or equal to
❶ 3 + 4 >= 7                          true
❷ 3 + 4 > 7                           false
  // Less than or equal to
❸ 5 + 6 <= 11                         true
❹ 5 + 6 < 11                          false
```

Note the difference between *greater than or equal to* at ❶ and *greater than* at ❷. The sum of `3 + 4` is not greater than `7`, but it is greater than or equal to `7`. Similarly, `5 + 6` is less than or equal to `11` ❸, but it's not less than `11` ❹.

Table 3-1 summarizes the six comparison operators.

Table 3-1: Comparison Operators

Symbol	Definition
==	Is equal to
!=	Is not equal to
>	Is greater than
<	Is less than
>=	Is greater than or equal to
<=	Is less than or equal to

You'll find yourself using these operators often when you write conditional statements.

COMPOUND BOOLEAN EXPRESSIONS

Compound Boolean expressions are simple Boolean expressions that have been joined together. It's a lot like making compound sentences in English with the words *and* and *or*. In programming, there is a third case: *not*. In Swift, we call these words *logical operators*. A logical operator either combines a Boolean expression with another one or negates it. The three logical operators in Swift are shown in Table 3-2.

Table 3-2: Logical Operators

Symbol	Definition
&&	Logical AND
||	Logical OR
!	Logical NOT

With logical operators, you can write statements that test if a value falls within a range, such as, "Is this person's age between 10 and 15?" You would do this by testing if the age is greater than 10 *and* less than 15 at the same time, like this:

```
var age = 12                    12
age > 10 && age < 15            true
```

The statement `age > 10 && age < 15` is `true` because both the conditions are `true`: `age` is greater than 10 and less than 15. An AND statement is `true` only if both conditions are `true`.

Try changing the value of age to 18 to see what happens:

Code	Result
`var age = 18`	`18`
`age > 10 && age < 15`	`false`

Because we changed age to 18, only one side of the statement is `true`. The variable age is still greater than 10, but it's no longer less than 15, so our expression evaluates to `false`.

Now test out OR by entering this code in your playground:

	Code	Result
	`let name = "Jacqueline"`	`"Jacqueline"`
	`name == "Jack"`	`false`
❶	`name == "Jack" || name == "Jacqueline"`	`true`

First, we make up a person named Jacqueline by setting the constant `name` to `"Jacqueline"`. Next, we test some conditions to see if they are `true` or `false`. Because `name == "Jacqueline"` is `true`, the OR statement at ❶ is `true` even though `name == "Jack"` is `false`. In English, this statement says, "This person's name is Jack *or* this person's name is Jacqueline." In an OR statement, only one of the conditions needs to be `true` for the whole expression to be `true`.

Let's try using some NOT statements. Enter the following into your playground:

	Code	Result
	`let isAGirl = true`	`true`
❶	`!isAGirl && name == "Jack"`	`false`
	`isAGirl && name == "Jacqueline"`	`true`
❷	`(!isAGirl && name == "Jack") || (isAGirl && name == "Jacqueline")`	`true`

The `!` operator is used in the compound Boolean statement ❶, which you could read as "Our person is *not* a girl *and* our person is named Jack." That statement has two logical operators, `!` and `&&`. You can combine as many logical operators as you want when you write compound Boolean expressions.

Sometimes it's a good idea to use parentheses to let the computer know what to evaluate first. Parentheses also make the code easier to read. This is similar to how you use parentheses when you use several math operations in one equation, as described in "Ordering Operations with Parentheses" on page 30. At ❷, we use parentheses to tell the computer to first check `!isAGirl && name == "Jack"` and then check `isAGirl && name == "Jacqueline"`. After it has evaluated both parts, the computer can evaluate the OR part for the entire statement, which will be `true` because the second part is `true`. Again, in an OR statement, the whole expression is `true` if any of the conditions is `true`.

Table 3-3 shows the three logical operators and the compound expressions you can make with them, as well as their corresponding Boolean values.

Table 3-3: Compound Boolean Expressions with Logical Operators

Logical operator	Compound expression	Value
NOT (!)	`!true`	`false`
NOT (!)	`!false`	`true`
AND (&&)	`true && true`	`true`
AND (&&)	`true && false`	`false`
AND (&&)	`false && true`	`false`
AND (&&)	`false && false`	`false`
OR (||)	`true || true`	`true`
OR (||)	`true || false`	`true`
OR (||)	`false || true`	`true`
OR (||)	`false || false`	`false`

The first item of the table shows that something that is NOT `true` is `false`. Similarly, something that is NOT `false` is `true`.

With the AND operator, only something that is `true && true` is `true`. This means that the expressions on both sides of the `&&` operator must be `true` for the `&&` expression to be `true`. A compound expression that is `true && false` will evaluate to `false`. And a compound `&&` expression in which both conditions are `false` will also evaluate to `false`.

When it comes to the OR operator, only one of the expressions on either side of the `||` operator must be `true` for the `||` expression to be `true`. Therefore, a `true || true` is `true`, and a `true || false` is also `true`. Only a compound OR expression in which both sides are `false` ends up being `false`.

CONDITIONAL STATEMENTS

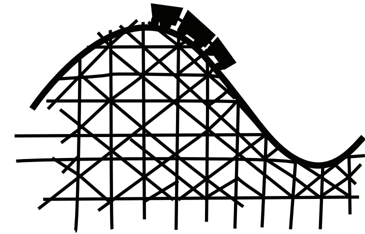
Conditional statements fall into two categories: the `if` statement and the `switch` statement. These statements present the computer with a condition that the computer makes a choice based on.

IF STATEMENTS

An `if` statement starts with the keyword `if` followed by a *condition*, which is always a Boolean expression. The computer examines the condition and executes the code inside the `if` statement if the condition is `true` or skips over that code if the condition is `false`. Let's write some code that tests whether a kid is tall enough to ride a roller coaster. Enter the following code into your playground:

```
let heightToRideAlone = 48.0
var height = 49.5
❶ if height >= heightToRideAlone{
❷     print("You are tall enough to ride this roller coaster.")
}
```

Here, we set 48 inches as the minimum height at which a kid can ride our roller coaster alone, and we set our rider's height to 49.5 inches. At ❶, we test whether the rider's `height` is greater than or equal to `heightToRideAlone`. If it is, the program says that they are tall enough to ride the roller coaster. To write our `if` statement, we put the keyword `if` in front of the condition `height >= heightToRideAlone`. Then we wrap the code that we want to execute when that condition is `true` in a set of braces ❷. Because our rider is tall enough, the computer will print `"You are tall enough to ride this roller coaster."` Hooray!



Let's see what happens if we change our rider's height. Change `height` to a number less than 48.0. This time, because the condition in the `if` statement evaluates to `false`, the program skips all of the code in the `if` statement and nothing happens.

else Statements

Often, you'll want to tell the computer to do one thing if a statement is `true` but something else if that statement is `false`. To do this, after the `if` statement and block of code, just type the keyword `else` followed by another block of code that you want to execute when the `if` condition isn't `true`. If the rider isn't tall enough to meet the condition, let's have the computer tell them they can't ride the roller coaster:

```
if height >= heightToRideAlone {
    print("You are tall enough to ride this roller coaster.")
❶ } else {
    print("Sorry. You cannot ride this roller coaster.")
}
```

Now if you change the rider's height to less than 48 inches, you'll see `"Sorry. You cannot ride this roller coaster."` That's because the `else` statement at ❶ tells the computer to print that message if the statement evaluates to `false`. In plain English, this is like saying, "If the rider is tall enough to ride the roller coaster, say they can ride it. Else, say they can't."

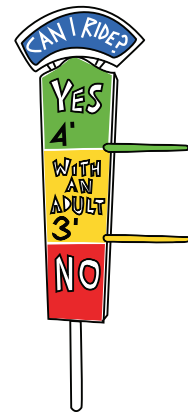
else if Statements

We could also test different conditions for the rider's height to create more rules for riding the roller coaster. We can do this by adding `else if` conditions. Let's add a new minimum height that requires the kid to ride with an adult:

```
let heightToRideAlone = 48.0
let heightToRideWithAdult = 36.0
var height = 47.5
```

```
if height >= heightToRideAlone {
    print("You are tall enough to ride this roller coaster alone.")
❶ } else if height >= heightToRideWithAdult {
    print("You can ride this roller coaster with an adult.")
} else {
    print("Sorry. You cannot ride this roller coaster.")
}
```

The else if statement at ❶ checks whether the rider's height is greater than or equal to heightToRideWithAdult. If a rider is shorter than 48 inches but taller than 36 inches, then the line "You can ride this roller coaster with an adult." appears in the results pane. If they are too short to ride alone or with an adult, then the computer prints "Sorry. You cannot ride this roller coaster."



else if statements are neat because you can use them to test lots of different conditions, but it's very important that you pay attention to the order of these conditions. To show you what we mean, change the rider's height to 50.0 so that they are tall enough to ride alone. Then, change the order of the conditions in our if else statement by making height >= heightToRideWithAdult the first condition and height >= heightToRideAlone the second condition. What do you think will be printed? Take a look at Figure 3-1 to find out.

```
MyPlayground
//: Playground - noun: a place where people can play

let heightToRideAlone = 48.0                                   48
let heightToRideWithAdult = 36.0                               36
var height = 50.0                                              50

if height >= heightToRideWithAdult {

    print("You can ride this roller coaster with an adult.")  "You can ride this roller coaster with an adult.\n"

} else if height >= heightToRideAlone {

    // This will NEVER get printed out
    print("You are tall enough to ride this roller coaster
        alone.")

} else {

    print("Sorry. You cannot ride this roller coaster.")
}

You can ride this roller coaster with an adult.
```

Figure 3-1: Be careful with your ordering of else if statements.

You can see that even though the rider is taller than heightToRideAlone, the program prints "You can ride this roller coaster with an adult." which is the expected output for a rider whose height is greater than heightToRideWithAdult but less than heightToRideAlone. We get this result because the rider's height

matches the first condition, so the computer prints the first sentence and doesn't bother checking anything else.

Once any part of an if or else if statement is found to be true, the rest of the conditions won't be checked. In our example in Figure 3-1, the first condition is true, so the rest of the conditions are skipped. This can lead to unexpected results in your programs, so if you ever run into problems in your if or else if statements, check the order of the conditions!

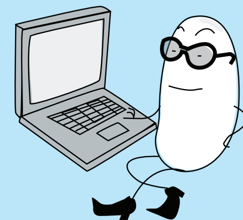
When you're working with if, else, or else if statements, there are a few important rules. The first is that you can't have an else or an else if statement unless you write an if statement first. The second is that although you can have as many else ifs as you want after an if, you can have only one else—and that else must be last. The else is the catch-all case if none of the other things has happened.

CODE WITH STYLE

Pay close attention to the coding style that we use in this book. By *coding style*, we mean the way that the code is written, the number of spaces used, the indentation of certain lines, and what things go on a new line. Take a look at this code:

```
// The opening brace, {, of a block of code goes on
// the same line as the condition
if height >= heightToRideAlone {

    // Statements inside a block of code should be indented by 4 spaces
    print("You are tall enough to ride this roller coaster alone.")

// The closing brace, }, of a block of code goes at the start of a new line
} else if height >= heightToRideWithAdult {

    // Extra blank lines can be added between statements
    // to make the code look less crowded

    print("You can ride this roller coaster with an adult.")

} else {

    print("Sorry. You cannot ride this roller coaster.")
}
```

Notice that after the if condition we leave a space and then place the opening brace, {, on the same line. The block's closing brace, }, always goes at the start of the next new line. The statements contained within the braces are indented by four spaces. This is something that Xcode does automatically for you to make the code more readable. Feel free to add blank lines if it makes it easier for you to read. In general, you should always have at least one blank line before a chunk of code such as an if statement.



SWITCH STATEMENTS

Whereas an `if` statement is used only to evaluate a Boolean expression (something that must be `true` or `false`), a `switch` statement can evaluate and branch out on any number of conditions. You could use a `switch` to check the value of an integer and tell the computer to do one thing if the integer equals `1`, something else if the integer equals `2`, and so on. Or, you could create a string called `dayOfTheWeek` and write a `switch` statement that makes the computer do something different based on the value of `dayOfTheWeek`.

When the computer finds the first match, that block of code is executed. Take a look at the following code, which assigns student projects for different grade levels:

```
var studentGrade = 5
var studentProject = "To be determined"

❶ switch studentGrade {
❷ case 1:
      studentProject = "A country of the student's choice"
  case 2:
      studentProject = "The Iditarod"
  case 3:
      studentProject = "Native Americans"
  case 4:
      studentProject = "A state of the student's choice"
  case 5:
      studentProject = "Colonial times"
❸ case 6, 7, 8:
      studentProject = "Student's choice"
❹ default:
      studentProject = "N/A"
  }
```

The `switch` statement starts with the keyword `switch` followed by the *control expression*. In this example, the control expression is the variable `studentGrade`.

After the control expression, a set of braces begins at ❶, and the body of the `switch` statement is inside these braces.

The body of the `switch` statement is made up of one or more cases. In this example, there are six cases total. Each case starts with the keyword `case` followed by a value and a colon, as shown at ❷. If a `case` statement matches the control expression, the code just after the case will run. Each case must have at least one line of code, or you'll get an error. In this example, `switch` is used to change the string assigned to the variable `studentProject` from `"To be determined"` to the string in the case that matches the control expression.

Note that you can have multiple cases all do the same thing. You can see that students in grades 6, 7, and 8 all get to choose their own projects ❸. We specify this by writing the keyword `case` and then a comma-separated list of values.

Finally, a `switch` statement must account for every possible case or value of the control expression. In our example, because `studentGrade` is an `Int`, our `switch` statement needs to have a case for all possible `Int` values. But this would take a really long time to write since there are so many! For example, –7 is an `Int`, as is 1,000. Do you really want to write 1,000 cases?

Instead of writing a separate case for every value, you can use the keyword `default` as the last case, as we did at ❹. You simply type `default` followed by a colon (`default:`) and then whatever code you want to run if none of the other cases match. Notice that the `default` case doesn't have the word `case` in front of it. The `default` case is really helpful for taking care of values that you might not expect and lets you avoid writing so many `case` statements. In this example, we expect a value only of 1 through 8 for `studentGrade`, so we use the `default` case to cover all other possible values.

Try running this `switch` statement and see what you get. Then try changing the values to test for different conditions. Play around with it!

WHAT YOU LEARNED

In this chapter, you learned how to program the computer to make choices based on conditions using `if` and `switch` statements. You learned how to write Boolean expressions and compound expressions, and about the different comparison operators. Conditional statements are an essential programming tool and are seen in almost every useful program. In Chapter 4, we're going to tackle another important type of programming statement—the loop. Loops tell the computer to do something over and over again until it is time to stop the loop.