Control Flow | Documentation

Structure code with branches, loops, and early exits.

Swift provides a variety of control flow statements. These include `while` loops to perform a task multiple times; `if`, `guard`, and `switch` statements to execute different branches of code based on certain conditions; and statements such as `break` and `continue` to transfer the flow of execution to another point in your code. Swift provides a `for`-`in` loop that makes it easy to iterate over arrays, dictionaries, ranges, strings, and other sequences. Swift also provides `defer` statements, which wrap code to be executed when leaving the current scope.

Swift's `switch` statement is considerably more powerful than its counterpart in many C-like languages. Cases can match many different patterns, including interval matches, tuples, and casts to a specific type. Matched values in a `switch` case can be bound to temporary constants or variables for use within the case's body, and complex matching conditions can be expressed with a `where` clause for each case.

For-In Loops

You use the `for`-`in` loop to iterate over a sequence, such as items in an array, ranges of numbers, or characters in a string.

This example uses a `for`-`in` loop to iterate over the items in an array:

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
    print("Hello, \(name)!")
}
// Hello, Anna!
// Hello, Alex!
// Hello, Brian!
// Hello, Jack!
```

You can also iterate over a dictionary to access its key-value pairs. Each item in the dictionary is returned as a `(key, value)` tuple when the dictionary is iterated, and you can decompose the `(key, value)` tuple's members as explicitly named constants for use within the body of the `for`-`in` loop. In the code example below, the dictionary's keys are decomposed into a constant called `animalName`, and the dictionary's values are decomposed into a constant called `legCount`.

```
let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
for (animalName, legCount) in numberOfLegs {
    print("\(animalName)s have \(legCount) legs")
}
// cats have 4 legs
// ants have 6 legs
// spiders have 8 legs
```

The contents of a `Dictionary` are inherently unordered, and iterating over them doesn't guarantee the order in which they will be retrieved. In particular, the order you insert items into a `Dictionary` doesn't define the order they're iterated. For more about arrays and dictionaries, see Collection Types.

You can also use `for`-`in` loops with numeric ranges. This example prints the first few entries in a five-times table:

```
for index in 1...5 {
    print("\(index) times 5 is \(index * 5)")
}
// 1 times 5 is 5
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

The sequence being iterated over is a range of numbers from `1` to `5`, inclusive, as indicated by the use of the closed range operator (`...`). The value of `index` is set to the first number in the range (`1`), and the statements inside the loop are executed. In this case, the loop contains only one statement, which prints an entry from the five-times table for the current value of `index`. After the statement is executed, the value of `index` is updated to contain the second value in the range (`2`), and the `print(_:separator:terminator:)` function is called again. This process continues until the end of the range is reached.

In the example above, `index` is a constant whose value is automatically set at the start of each iteration of the loop. As such, `index` doesn't have to be declared before it's used. It's implicitly declared simply by its inclusion in the loop declaration, without the need for a `let` declaration keyword.

If you don't need each value from a sequence, you can ignore the values by using an underscore in place of a variable name.

```
let base = 3
let power = 10
var answer = 1
for _ in 1...power {
    answer *= base
}
print("\(base) to the power of \(power) is \(answer)")
// Prints "3 to the power of 10 is 59049"
```

The example above calculates the value of one number to the power of another (in this case, `3` to the power of `10`). It multiplies a starting value of `1` (that is, `3` to the power of `0`) by `3`, ten times, using a closed range that starts with `1` and ends with `10`. For this calculation, the individual counter values each time through the loop are unnecessary — the code simply executes the loop the correct number of times. The underscore character (`_`) used in place of a loop variable causes the individual values to be ignored and doesn't provide access to the current value during each iteration of the loop.

In some situations, you might not want to use closed ranges, which include both endpoints. Consider drawing the tick marks for every minute on a watch face. You want to draw `60` tick marks, starting with the `0` minute. Use the half-open range operator (`..<`) to include the lower bound but not the upper bound. For more about ranges, see Range Operators.

```
let minutes = 60
for tickMark in 0..<minutes {
    // render the tick mark each minute (60 times)
}
```

Some users might want fewer tick marks in their UI. They could prefer one mark every `5` minutes instead. Use the `stride(from:to:by:)` function to skip the unwanted marks.

```
let minuteInterval = 5
for tickMark in stride(from: 0, to: minutes, by: minuteInterval) {
    // render the tick mark every 5 minutes (0, 5, 10, 15 ... 45, 50, 55)
}
```

Closed ranges are also available, by using `stride(from:through:by:)` instead:

```
let hours = 12
let hourInterval = 3
for tickMark in stride(from: 3, through: hours, by: hourInterval) {
    // render the tick mark every 3 hours (3, 6, 9, 12)
}
```

The examples above use a `for`-`in` loop to iterate ranges, arrays, dictionaries, and strings. However, you can use this syntax to iterate *any* collection, including your own classes and collection types, as long as those types conform to the `Sequence` protocol.

While Loops

A `while` loop performs a set of statements until a condition becomes `false`. These kinds of loops are best used when the number of iterations isn't known before the first iteration begins. Swift provides two kinds of `while` loops:

- `while` evaluates its condition at the start of each pass through the loop.
- `repeat`-`while` evaluates its condition at the end of each pass through the loop.

While

A `while` loop starts by evaluating a single condition. If the condition is `true`, a set of statements is repeated until the condition becomes `false`.

Here's the general form of a `while` loop:

```
while <#condition#> {
   <#statements#>
}
```

This example plays a simple game of *Snakes and Ladders* (also known as *Chutes and Ladders*):

The rules of the game are as follows:

- The board has 25 squares, and the aim is to land on or beyond square 25.
- The player's starting square is "square zero", which is just off the bottom-left corner of the board.

- Each turn, you roll a six-sided dice and move by that number of squares, following the horizontal path indicated by the dotted arrow above.
- If your turn ends at the bottom of a ladder, you move up that ladder.
- If your turn ends at the head of a snake, you move down that snake.

The game board is represented by an array of `Int` values. Its size is based on a constant called `finalSquare`, which is used to initialize the array and also to check for a win condition later in the example. Because the players start off the board, on "square zero", the board is initialized with 26 zero `Int` values, not 25.

```
let finalSquare = 25
var board = [Int](repeating: 0, count: finalSquare + 1)
```

Some squares are then set to have more specific values for the snakes and ladders. Squares with a ladder base have a positive number to move you up the board, whereas squares with a snake head have a negative number to move you back down the board.

```
board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
```

Square 3 contains the bottom of a ladder that moves you up to square 11. To represent this, `board[03]` is equal to `+08`, which is equivalent to an integer value of `8` (the difference between `3` and `11`). To align the values and statements, the unary plus operator (`+i`) is explicitly used with the unary minus operator (`-i`) and numbers lower than `10` are padded with zeros. (Neither stylistic technique is strictly necessary, but they lead to neater code.)

```
var square = 0
var diceRoll = 0
while square < finalSquare {
    // roll the dice
    diceRoll += 1
    if diceRoll == 7 { diceRoll = 1 }
    // move by the rolled amount
    square += diceRoll
    if square < board.count {
        // if we're still on the board, move up or down for a snake or a ladd
```

```
er
        square += board[square]
    }
}
print("Game over!")
```

The example above uses a very simple approach to dice rolling. Instead of generating a random number, it starts with a `diceRoll` value of `0`. Each time through the `while` loop, `diceRoll` is incremented by one and is then checked to see whether it has become too large. Whenever this return value equals `7`, the dice roll has become too large and is reset to a value of `1`. The result is a sequence of `diceRoll` values that's always `1`, `2`, `3`, `4`, `5`, `6`, `1`, `2` and so on.

After rolling the dice, the player moves forward by `diceRoll` squares. It's possible that the dice roll may have moved the player beyond square 25, in which case the game is over. To cope with this scenario, the code checks that `square` is less than the `board` array's `count` property. If `square` is valid, the value stored in `board[square]` is added to the current `square` value to move the player up or down any ladders or snakes.

The current `while` loop execution then ends, and the loop's condition is checked to see if the loop should be executed again. If the player has moved on or beyond square number `25`, the loop's condition evaluates to `false` and the game ends.

A `while` loop is appropriate in this case, because the length of the game isn't clear at the start of the `while` loop. Instead, the loop is executed until a particular condition is satisfied.

Repeat-While

The other variation of the `while` loop, known as the `repeat-while` loop, performs a single pass through the loop block first, *before* considering the loop's condition. It then continues to repeat the loop until the condition is `false`.

Here's the general form of a `repeat-while` loop:

```
repeat {
   <#statements#>
} while <#condition#>
```

Here's the *Snakes and Ladders* example again, written as a `repeat-while` loop rather than a `while` loop. The values of `finalSquare`, `board`, `square`, and `diceRoll` are initialized in exactly the same way as with a `while` loop.

```
let finalSquare = 25
var board = [Int](repeating: 0, count: finalSquare + 1)
board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
var square = 0
var diceRoll = 0
```

In this version of the game, the *first* action in the loop is to check for a ladder or a snake. No ladder on the board takes the player straight to square 25, and so it isn't possible to win the game by moving up a ladder. Therefore, it's safe to check for a snake or a ladder as the first action in the loop.

At the start of the game, the player is on "square zero". `board[0]` always equals `0` and has no effect.

```
repeat {
    // move up or down for a snake or ladder
    square += board[square]
    // roll the dice
    diceRoll += 1
    if diceRoll == 7 { diceRoll = 1 }
    // move by the rolled amount
    square += diceRoll
} while square < finalSquare
print("Game over!")
```

After the code checks for snakes and ladders, the dice is rolled and the player is moved forward by `diceRoll` squares. The current loop execution then ends.

The loop's condition (`while square < finalSquare`) is the same as before, but this time it's not evaluated until the *end* of the first run through the loop. The structure of the `repeat`-`while` loop is better suited to this game than the `while` loop in the previous example. In the `repeat`-`while` loop above, `square += board[square]` is always executed *immediately after* the loop's `while` condition confirms that `square` is still on the board. This behavior removes the need for the array bounds check seen in the `while` loop version of the game described earlier.

Conditional Statements

It's often useful to execute different pieces of code based on certain conditions. You might want to run an extra piece of code when an error occurs, or to display a message when a value becomes too high or too

low. To do this, you make parts of your code *conditional*.

Swift provides two ways to add conditional branches to your code: the `if` statement and the `switch` statement. Typically, you use the `if` statement to evaluate simple conditions with only a few possible outcomes. The `switch` statement is better suited to more complex conditions with multiple possible permutations and is useful in situations where pattern matching can help select an appropriate code branch to execute.

If

In its simplest form, the `if` statement has a single `if` condition. It executes a set of statements only if that condition is `true`.

```
var temperatureInFahrenheit = 30
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
}
// Prints "It's very cold. Consider wearing a scarf."
```

The example above checks whether the temperature is less than or equal to 32 degrees Fahrenheit (the freezing point of water). If it is, a message is printed. Otherwise, no message is printed, and code execution continues after the `if` statement's closing brace.

The `if` statement can provide an alternative set of statements, known as an *else clause*, for situations when the `if` condition is `false`. These statements are indicated by the `else` keyword.

```
temperatureInFahrenheit = 40
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else {
    print("It's not that cold. Wear a T-shirt.")
}
// Prints "It's not that cold. Wear a T-shirt."
```

One of these two branches is always executed. Because the temperature has increased to `40` degrees Fahrenheit, it's no longer cold enough to advise wearing a scarf and so the `else` branch is triggered instead.

You can chain multiple `if` statements together to consider additional clauses.

```
temperatureInFahrenheit = 90
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear sunscreen.")
} else {
    print("It's not that cold. Wear a T-shirt.")
}
// Prints "It's really warm. Don't forget to wear sunscreen."
```

Here, an additional `if` statement was added to respond to particularly warm temperatures. The final `else` clause remains, and it prints a response for any temperatures that are neither too warm nor too cold.

The final `else` clause is optional, however, and can be excluded if the set of conditions doesn't need to be complete.

```
temperatureInFahrenheit = 72
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear sunscreen.")
}
```

Because the temperature is neither too cold nor too warm to trigger the `if` or `else if` conditions, no message is printed.

Swift provides a shorthand spelling of `if` that you can use when setting values. For example, consider the following code:

```
let temperatureInCelsius = 25
let weatherAdvice: String

if temperatureInCelsius <= 0 {
    weatherAdvice = "It's very cold. Consider wearing a scarf."
} else if temperatureInCelsius >= 30 {
    weatherAdvice = "It's really warm. Don't forget to wear sunscreen."
} else {
    weatherAdvice = "It's not that cold. Wear a T-shirt."
}
```

```
print(weatherAdvice)
// Prints "It's not that cold. Wear a T-shirt."
```

Here, each of the branches sets a value for the `weatherAdvice` constant, which is printed after the `if` statement.

Using the alternate syntax, known as an `if` expression, you can write this code more concisely:

```
let weatherAdvice = if temperatureInCelsius <= 0 {
    "It's very cold. Consider wearing a scarf."
} else if temperatureInCelsius >= 30 {
    "It's really warm. Don't forget to wear sunscreen."
} else {
    "It's not that cold. Wear a T-shirt."
}

print(weatherAdvice)
// Prints "It's not that cold. Wear a T-shirt."
```

In this `if` expression version, each branch contains a single value. If a branch's condition is true, then that branch's value is used as the value for the whole `if` expression in the assignment of `weatherAdvice`. Every `if` branch has a corresponding `else if` branch or `else` branch, ensuring that one of the branches always matches and that the `if` expression always produces a value, regardless of which conditions are true.

Because the syntax for the assignment starts outside the `if` expression, there's no need to repeat `weatherAdvice =` inside each branch. Instead, each branch of the `if` expression produces one of the three possible values for `weatherAdvice`, and the assignment uses that value.

All of the branches of an `if` expression need to contain values of the same type. Because Swift checks the type of each branch separately, values like `nil` that can be used with more than one type prevent Swift from determining the `if` expression's type automatically. Instead, you need to specify the type explicitly — for example:

```
let freezeWarning: String? = if temperatureInCelsius <= 0 {
    "It's below freezing. Watch for ice!"
} else {
    nil
}
```

In the code above, one branch of the `if` expression has a string value and the other branch has a `nil` value. The `nil` value could be used as a value for any optional type, so you have to explicitly write that `freezeWarning` is an optional string, as described in Type Annotations.

An alternate way to provide this type information is to provide an explicit type for `nil`, instead of providing an explicit type for `freezeWarning`:

```
let freezeWarning = if temperatureInCelsius <= 0 {
    "It's below freezing. Watch for ice!"
} else {
    nil as String?
}
```

An `if` expression can respond to unexpected failures by throwing an error or calling a function like `fatalError(_:file:line:)` that never returns. For example:

```
let weatherAdvice = if temperatureInCelsius > 100 {
    throw TemperatureError.boiling
} else {
    "It's a reasonable temperature."
}
```

In this example, the `if` expression checks whether the forecast temperature is hotter than 100° C — the boiling point of water. A temperature this hot causes the `if` expression to throw a `.boiling` error instead of returning a textual summary. Even though this `if` expression can throw an error, you don't write `try` before it. For information about working with errors, see Error Handling.

In addition to using `if` expressions on the right-hand side of an assignment, as shown in the examples above, you can also use them as the value that a function or closure returns.

Switch

A `switch` statement considers a value and compares it against several possible matching patterns. It then executes an appropriate block of code, based on the first pattern that matches successfully. A `switch` statement provides an alternative to the `if` statement for responding to multiple potential states.

In its simplest form, a `switch` statement compares a value against one or more values of the same type.

```
switch <#some value to consider#> {
case <#value 1#>:
    <#respond to value 1#>
case <#value 2#>,
    <#value 3#>:
    <#respond to value 2 or 3#>
default:
    <#otherwise, do something else#>
}
```

Every `switch` statement consists of multiple possible *cases*, each of which begins with the `case` keyword. In addition to comparing against specific values, Swift provides several ways for each case to specify more complex matching patterns. These options are described later in this chapter.

Like the body of an `if` statement, each `case` is a separate branch of code execution. The `switch` statement determines which branch should be selected. This procedure is known as *switching* on the value that's being considered.

Every `switch` statement must be *exhaustive*. That is, every possible value of the type being considered must be matched by one of the `switch` cases. If it's not appropriate to provide a case for every possible value, you can define a default case to cover any values that aren't addressed explicitly. This default case is indicated by the `default` keyword, and must always appear last.

This example uses a `switch` statement to consider a single lowercase character called `someCharacter`:

```
let someCharacter: Character = "z"
switch someCharacter {
case "a":
    print("The first letter of the Latin alphabet")
case "z":
    print("The last letter of the Latin alphabet")
default:
    print("Some other character")
}
// Prints "The last letter of the Latin alphabet"
```

The `switch` statement's first case matches the first letter of the English alphabet, `a`, and its second case matches the last letter, `z`. Because the `switch` must have a case for every possible character, not just

every alphabetic character, this `switch` statement uses a `default` case to match all characters other than `a` and `z`. This provision ensures that the `switch` statement is exhaustive.

Like `if` statements, `switch` statements also have an expression form:

```
let anotherCharacter: Character = "a"
let message = switch anotherCharacter {
case "a":
    "The first letter of the Latin alphabet"
case "z":
    "The last letter of the Latin alphabet"
default:
    "Some other character"
}

print(message)
// Prints "The first letter of the Latin alphabet"
```

In this example, each case in the `switch` expression contains the value for `message` to be used when that case matches `anotherCharacter`. Because `switch` is always exhaustive, there is always a value to assign.

As with `if` expressions, you can throw an error or call a function like `fatalError(_:file:line:)` that never returns instead of providing a value for a given case. You can use `switch` expressions on the right-hand side of an assignment, as shown in the example above, and as the value that a function or closure returns.

No Implicit Fallthrough

In contrast with `switch` statements in C and Objective-C, `switch` statements in Swift don't fall through the bottom of each case and into the next one by default. Instead, the entire `switch` statement finishes its execution as soon as the first matching `switch` case is completed, without requiring an explicit `break` statement. This makes the `switch` statement safer and easier to use than the one in C and avoids executing more than one `switch` case by mistake.

The body of each case *must* contain at least one executable statement. It isn't valid to write the following code, because the first case is empty:

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
case "a": // Invalid, the case has an empty body
```

```
case "A":
    print("The letter A")
default:
    print("Not the letter A")
}
// This will report a compile-time error.
```

Unlike a `switch` statement in C, this `switch` statement doesn't match both `"a"` and `"A"`. Rather, it reports a compile-time error that `case "a":` doesn't contain any executable statements. This approach avoids accidental fallthrough from one case to another and makes for safer code that's clearer in its intent.

To make a `switch` with a single case that matches both `"a"` and `"A"`, combine the two values into a compound case, separating the values with commas.

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
case "a", "A":
    print("The letter A")
default:
    print("Not the letter A")
}
// Prints "The letter A"
```

For readability, a compound case can also be written over multiple lines. For more information about compound cases, see Compound Cases.

Interval Matching

Values in `switch` cases can be checked for their inclusion in an interval. This example uses number intervals to provide a natural-language count for numbers of any size:

```
let approximateCount = 62
let countedThings = "moons orbiting Saturn"
let naturalCount: String
switch approximateCount {
case 0:
    naturalCount = "no"
case 1..<5:
    naturalCount = "a few"
case 5..<12:
    naturalCount = "several"
```

```
case 12..<100:
    naturalCount = "dozens of"
case 100..<1000:
    naturalCount = "hundreds of"
default:
    naturalCount = "many"
}
print("There are \(naturalCount) \(countedThings).")
// Prints "There are dozens of moons orbiting Saturn."
```

In the above example, `approximateCount` is evaluated in a `switch` statement. Each `case` compares that value to a number or interval. Because the value of `approximateCount` falls between 12 and 100, `naturalCount` is assigned the value `"dozens of"`, and execution is transferred out of the `switch` statement.

Tuples

You can use tuples to test multiple values in the same `switch` statement. Each element of the tuple can be tested against a different value or interval of values. Alternatively, use the underscore character (`_`), also known as the wildcard pattern, to match any possible value.

The example below takes an (x, y) point, expressed as a simple tuple of type `(Int, Int)`, and categorizes it on the graph that follows the example.

```
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    print("\(somePoint) is at the origin")
case (_, 0):
    print("\(somePoint) is on the x-axis")
case (0, _):
    print("\(somePoint) is on the y-axis")
case (-2...2, -2...2):
    print("\(somePoint) is inside the box")
default:
    print("\(somePoint) is outside of the box")
}
// Prints "(1, 1) is inside the box"
```

The `switch` statement determines whether the point is at the origin (0, 0), on the red x-axis, on the green y-axis, inside the blue 4-by-4 box centered on the origin, or outside of the box.

Unlike C, Swift allows multiple `switch` cases to consider the same value or values. In fact, the point (0, 0) could match all *four* of the cases in this example. However, if multiple matches are possible, the first matching case is always used. The point (0, 0) would match `case (0, 0)` first, and so all other matching cases would be ignored.

Value Bindings

A `switch` case can name the value or values it matches to temporary constants or variables, for use in the body of the case. This behavior is known as *value binding*, because the values are bound to temporary constants or variables within the case's body.

The example below takes an (x, y) point, expressed as a tuple of type `(Int, Int)`, and categorizes it on the graph that follows:

```
let anotherPoint = (2, 0)
switch anotherPoint {
case (let x, 0):
    print("on the x-axis with an x value of \(x)")
case (0, let y):
    print("on the y-axis with a y value of \(y)")
case let (x, y):
    print("somewhere else at (\(x), \(y))")
}
// Prints "on the x-axis with an x value of 2"
```

The `switch` statement determines whether the point is on the red x-axis, on the green y-axis, or elsewhere (on neither axis).

The three `switch` cases declare placeholder constants `x` and `y`, which temporarily take on one or both tuple values from `anotherPoint`. The first case, `case (let x, 0)`, matches any point with a `y` value of `0` and assigns the point's `x` value to the temporary constant `x`. Similarly, the second case, `case (0, let y)`, matches any point with an `x` value of `0` and assigns the point's `y` value to the temporary constant `y`.

After the temporary constants are declared, they can be used within the case's code block. Here, they're used to print the categorization of the point.

This `switch` statement doesn't have a `default` case. The final case, `case let (x, y)`, declares a tuple of two placeholder constants that can match any value. Because `anotherPoint` is always a tuple of two values, this case matches all possible remaining values, and a `default` case isn't needed to make the `switch` statement exhaustive.

Where

A `switch` case can use a `where` clause to check for additional conditions.

The example below categorizes an (x, y) point on the following graph:

```
let yetAnotherPoint = (1, -1)
switch yetAnotherPoint {
case let (x, y) where x == y:
    print("(\(x), \(y)) is on the line x == y")
case let (x, y) where x == -y:
    print("(\(x), \(y)) is on the line x == -y")
case let (x, y):
    print("(\(x), \(y)) is just some arbitrary point")
}
// Prints "(1, -1) is on the line x == -y"
```

The `switch` statement determines whether the point is on the green diagonal line where `x == y`, on the purple diagonal line where `x == -y`, or neither.

The three `switch` cases declare placeholder constants `x` and `y`, which temporarily take on the two tuple values from `yetAnotherPoint`. These constants are used as part of a `where` clause, to create a dynamic filter. The `switch` case matches the current value of `point` only if the `where` clause's condition evaluates to `true` for that value.

As in the previous example, the final case matches all possible remaining values, and so a `default` case isn't needed to make the `switch` statement exhaustive.

Compound Cases

Multiple switch cases that share the same body can be combined by writing several patterns after `case`, with a comma between each of the patterns. If any of the patterns match, then the case is considered to match. The patterns can be written over multiple lines if the list is long. For example:

```
let someCharacter: Character = "e"
switch someCharacter {
case "a", "e", "i", "o", "u":
    print("\(someCharacter) is a vowel")
case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
    "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
    print("\(someCharacter) is a consonant")
```

```
default:
    print("\(someCharacter) isn't a vowel or a consonant")
}
// Prints "e is a vowel"
```

The `switch` statement's first case matches all five lowercase vowels in the English language. Similarly, its second case matches all lowercase English consonants. Finally, the `default` case matches any other character.

Compound cases can also include value bindings. All of the patterns of a compound case have to include the same set of value bindings, and each binding has to get a value of the same type from all of the patterns in the compound case. This ensures that, no matter which part of the compound case matched, the code in the body of the case can always access a value for the bindings and that the value always has the same type.

```
let stillAnotherPoint = (9, 0)
switch stillAnotherPoint {
case (let distance, 0), (0, let distance):
    print("On an axis, \(distance) from the origin")
default:
    print("Not on an axis")
}
// Prints "On an axis, 9 from the origin"
```

The `case` above has two patterns: `(let distance, 0)` matches points on the x-axis and `(0, let distance)` matches points on the y-axis. Both patterns include a binding for `distance` and `distance` is an integer in both patterns — which means that the code in the body of the `case` can always access a value for `distance`.

Control Transfer Statements

Control transfer statements change the order in which your code is executed, by transferring control from one piece of code to another. Swift has five control transfer statements:

- `continue`
- `break`
- `fallthrough`

- `return`
- `throw`

The `continue`, `break`, and `fallthrough` statements are described below. The `return` statement is described in Functions, and the `throw` statement is described in Propagating Errors Using Throwing Functions.

Continue

The `continue` statement tells a loop to stop what it's doing and start again at the beginning of the next iteration through the loop. It says "I am done with the current loop iteration" without leaving the loop altogether.

The following example removes all vowels and spaces from a lowercase string to create a cryptic puzzle phrase:

```
let puzzleInput = "great minds think alike"
var puzzleOutput = ""
let charactersToRemove: [Character] = ["a", "e", "i", "o", "u", " "]
for character in puzzleInput {
    if charactersToRemove.contains(character) {
        continue
    }
    puzzleOutput.append(character)
}
print(puzzleOutput)
// Prints "grtmndsthnklk"
```

The code above calls the `continue` keyword whenever it matches a vowel or a space, causing the current iteration of the loop to end immediately and to jump straight to the start of the next iteration.

Break

The `break` statement ends execution of an entire control flow statement immediately. The `break` statement can be used inside a `switch` or loop statement when you want to terminate the execution of the `switch` or loop statement earlier than would otherwise be the case.

Break in a Loop Statement

When used inside a loop statement, `break` ends the loop's execution immediately and transfers control to the code after the loop's closing brace (`}`). No further code from the current iteration of the loop is executed, and no further iterations of the loop are started.

Break in a Switch Statement

When used inside a `switch` statement, `break` causes the `switch` statement to end its execution immediately and to transfer control to the code after the `switch` statement's closing brace (`}`).

This behavior can be used to match and ignore one or more cases in a `switch` statement. Because Swift's `switch` statement is exhaustive and doesn't allow empty cases, it's sometimes necessary to deliberately match and ignore a case in order to make your intentions explicit. You do this by writing the `break` statement as the entire body of the case you want to ignore. When that case is matched by the `switch` statement, the `break` statement inside the case ends the `switch` statement's execution immediately.

The following example switches on a `Character` value and determines whether it represents a number symbol in one of four languages. For brevity, multiple values are covered in a single `switch` case.

```
let numberSymbol: Character = "三"  // Chinese symbol for the number 3
var possibleIntegerValue: Int?
switch numberSymbol {
case "1", "١", "一", "๑":
    possibleIntegerValue = 1
case "2", "٢", "二", "๒":
    possibleIntegerValue = 2
case "3", "٣", "三", "๓":
    possibleIntegerValue = 3
case "4", "٤", "四", "๔":
    possibleIntegerValue = 4
default:
    break
}
if let integerValue = possibleIntegerValue {
    print("The integer value of \(numberSymbol) is \(integerValue).")
} else {
    print("An integer value couldn't be found for \(numberSymbol).")
}
// Prints "The integer value of 三 is 3."
```

This example checks `numberSymbol` to determine whether it's a Latin, Arabic, Chinese, or Thai symbol for the numbers `1` to `4`. If a match is found, one of the `switch` statement's cases sets an optional

`Int?` variable called `possibleIntegerValue` to an appropriate integer value.

After the `switch` statement completes its execution, the example uses optional binding to determine whether a value was found. The `possibleIntegerValue` variable has an implicit initial value of `nil` by virtue of being an optional type, and so the optional binding will succeed only if `possibleIntegerValue` was set to an actual value by one of the `switch` statement's first four cases.

Because it's not practical to list every possible `Character` value in the example above, a `default` case handles any characters that aren't matched. This `default` case doesn't need to perform any action, and so it's written with a single `break` statement as its body. As soon as the `default` case is matched, the `break` statement ends the `switch` statement's execution, and code execution continues from the `if let` statement.

Fallthrough

In Swift, `switch` statements don't fall through the bottom of each case and into the next one. That is, the entire `switch` statement completes its execution as soon as the first matching case is completed. By contrast, C requires you to insert an explicit `break` statement at the end of every `switch` case to prevent fallthrough. Avoiding default fallthrough means that Swift `switch` statements are much more concise and predictable than their counterparts in C, and thus they avoid executing multiple `switch` cases by mistake.

If you need C-style fallthrough behavior, you can opt in to this behavior on a case-by-case basis with the `fallthrough` keyword. The example below uses `fallthrough` to create a textual description of a number.

```
let integerToDescribe = 5
var description = "The number \(integerToDescribe) is"
switch integerToDescribe {
case 2, 3, 5, 7, 11, 13, 17, 19:
    description += " a prime number, and also"
    fallthrough
default:
    description += " an integer."
}
print(description)
// Prints "The number 5 is a prime number, and also an integer."
```

This example declares a new `String` variable called `description` and assigns it an initial value. The function then considers the value of `integerToDescribe` using a `switch` statement. If the value of

`integerToDescribe` is one of the prime numbers in the list, the function appends text to the end of `description`, to note that the number is prime. It then uses the `fallthrough` keyword to "fall into" the `default` case as well. The `default` case adds some extra text to the end of the description, and the `switch` statement is complete.

Unless the value of `integerToDescribe` is in the list of known prime numbers, it isn't matched by the first `switch` case at all. Because there are no other specific cases, `integerToDescribe` is matched by the `default` case.

After the `switch` statement has finished executing, the number's description is printed using the `print(_:separator:terminator:)` function. In this example, the number `5` is correctly identified as a prime number.

Labeled Statements

In Swift, you can nest loops and conditional statements inside other loops and conditional statements to create complex control flow structures. However, loops and conditional statements can both use the `break` statement to end their execution prematurely. Therefore, it's sometimes useful to be explicit about which loop or conditional statement you want a `break` statement to terminate. Similarly, if you have multiple nested loops, it can be useful to be explicit about which loop the `continue` statement should affect.

To achieve these aims, you can mark a loop statement or conditional statement with a *statement label*. With a conditional statement, you can use a statement label with the `break` statement to end the execution of the labeled statement. With a loop statement, you can use a statement label with the `break` or `continue` statement to end or continue the execution of the labeled statement.

A labeled statement is indicated by placing a label on the same line as the statement's introducer keyword, followed by a colon. Here's an example of this syntax for a `while` loop, although the principle is the same for all loops and `switch` statements:

```
<#label name#>: while <#condition#> {
    <#statements#>
}
```

The following example uses the `break` and `continue` statements with a labeled `while` loop for an adapted version of the *Snakes and Ladders* game that you saw earlier in this chapter. This time around, the game has an extra rule:

> To win, you must land *exactly* on square 25.

If a particular dice roll would take you beyond square 25, you must roll again until you roll the exact number needed to land on square 25.

The game board is the same as before.

The values of `finalSquare`, `board`, `square`, and `diceRoll` are initialized in the same way as before:

```
let finalSquare = 25
var board = [Int](repeating: 0, count: finalSquare + 1)
board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
var square = 0
var diceRoll = 0
```

This version of the game uses a `while` loop and a `switch` statement to implement the game's logic. The `while` loop has a statement label called `gameLoop` to indicate that it's the main game loop for the Snakes and Ladders game.

The `while` loop's condition is `while square != finalSquare`, to reflect that you must land exactly on square 25.

```
gameLoop: while square != finalSquare {
    diceRoll += 1
    if diceRoll == 7 { diceRoll = 1 }
    switch square + diceRoll {
    case finalSquare:
        // diceRoll will move us to the final square, so the game is over
        break gameLoop
    case let newSquare where newSquare > finalSquare:
        // diceRoll will move us beyond the final square, so roll again
        continue gameLoop
    default:
        // this is a valid move, so find out its effect
        square += diceRoll
        square += board[square]
    }
}
print("Game over!")
```

The dice is rolled at the start of each loop. Rather than moving the player immediately, the loop uses a `switch` statement to consider the result of the move and to determine whether the move is allowed:

- If the dice roll will move the player onto the final square, the game is over. The `break gameLoop` statement transfers control to the first line of code outside of the `while` loop, which ends the game.
- If the dice roll will move the player *beyond* the final square, the move is invalid and the player needs to roll again. The `continue gameLoop` statement ends the current `while` loop iteration and begins the next iteration of the loop.
- In all other cases, the dice roll is a valid move. The player moves forward by `diceRoll` squares, and the game logic checks for any snakes and ladders. The loop then ends, and control returns to the `while` condition to decide whether another turn is required.

Early Exit

A `guard` statement, like an `if` statement, executes statements depending on the Boolean value of an expression. You use a `guard` statement to require that a condition must be true in order for the code after the `guard` statement to be executed. Unlike an `if` statement, a `guard` statement always has an `else` clause — the code inside the `else` clause is executed if the condition isn't true.

```
func greet(person: [String: String]) {
    guard let name = person["name"] else {
        return
    }

    print("Hello \(name)!")

    guard let location = person["location"] else {
        print("I hope the weather is nice near you.")
        return
    }

    print("I hope the weather is nice in \(location).")
}

greet(person: ["name": "John"])
// Prints "Hello John!"
// Prints "I hope the weather is nice near you."
greet(person: ["name": "Jane", "location": "Cupertino"])
// Prints "Hello Jane!"
// Prints "I hope the weather is nice in Cupertino."
```

If the `guard` statement's condition is met, code execution continues after the `guard` statement's closing brace. Any variables or constants that were assigned values using an optional binding as part of the condition are available for the rest of the code block that the `guard` statement appears in.

If that condition isn't met, the code inside the `else` branch is executed. That branch must transfer control to exit the code block in which the `guard` statement appears. It can do this with a control transfer statement such as `return`, `break`, `continue`, or `throw`, or it can call a function or method that doesn't return, such as `fatalError(_:file:line:)`.

Using a `guard` statement for requirements improves the readability of your code, compared to doing the same check with an `if` statement. It lets you write the code that's typically executed without wrapping it in an `else` block, and it lets you keep the code that handles a violated requirement next to the requirement.

Deferred Actions

Unlike control-flow constructs like `if` and `while`, which let you control whether part of your code is executed or how many times it gets executed, `defer` controls *when* a piece of code is executed. You use a `defer` block to write code that will be executed later, when your program reaches the end of the current scope. For example:

```
var score = 1
if score < 10 {
    defer {
        print(score)
    }
    score += 5
}
// Prints "6"
```

In the example above, the code inside of the `defer` block is executed before exiting the body of the `if` statement. First, the code in the `if` statement runs, which increments `score` by five. Then, before exiting the `if` statement's scope, the deferred code is run, which prints `score`.

The code inside of the `defer` always runs, regardless of how the program exits that scope. That includes code like an early exit from a function, breaking out of a `for` loop, or throwing an error. This behavior makes `defer` useful for operations where you need to guarantee a pair of actions happen — like manually allocating and freeing memory, opening and closing low-level file descriptors, and beginning and ending transactions in a database — because you can write both actions next to each other in your code.

For example, the following code gives a temporary bonus to the score, by adding and subtracting 100 inside a chunk of code:

```
var score = 3
if score < 100 {
    score += 100
    defer {
        score -= 100
    }
    // Other code that uses the score with its bonus goes here.
    print(score)
}
// Prints "103"
```

If you write more than one `defer` block in the same scope, the first one you specify is the last one to run.

```
if score < 10 {
    defer {
        print(score)
    }
    defer {
        print("The score is:")
    }
    score += 5
}
// Prints "The score is:"
// Prints "6"
```

If your program stops running — for example, because of a runtime error or a crash — deferred code doesn't execute. However, deferred code does execute after an error is thrown; for information about using `defer` with error handling, see Specifying Cleanup Actions.

Checking API Availability

Swift has built-in support for checking API availability, which ensures that you don't accidentally use APIs that are unavailable on a given deployment target.

The compiler uses availability information in the SDK to verify that all of the APIs used in your code are available on the deployment target specified by your project. Swift reports an error at compile time if you

try to use an API that isn't available.

You use an *availability condition* in an `if` or `guard` statement to conditionally execute a block of code, depending on whether the APIs you want to use are available at runtime. The compiler uses the information from the availability condition when it verifies that the APIs in that block of code are available.

```
if #available(iOS 10, macOS 10.12, *) {
    // Use iOS 10 APIs on iOS, and use macOS 10.12 APIs on macOS
} else {
    // Fall back to earlier iOS and macOS APIs
}
```

The availability condition above specifies that in iOS, the body of the `if` statement executes only in iOS 10 and later; in macOS, only in macOS 10.12 and later. The last argument, `*`, is required and specifies that on any other platform, the body of the `if` executes on the minimum deployment target specified by your target.

In its general form, the availability condition takes a list of platform names and versions. You use platform names such as `iOS`, `macOS`, `watchOS`, and `tvOS` — for the full list, see Declaration Attributes. In addition to specifying major version numbers like iOS 8 or macOS 10.10, you can specify minor versions numbers like iOS 11.2.6 and macOS 10.13.3.

```
if #available(<#platform name#> <#version#>, <#...#>, *) {
    <#statements to execute if the APIs are available#>
} else {
    <#fallback statements to execute if the APIs are unavailable#>
}
```

When you use an availability condition with a `guard` statement, it refines the availability information that's used for the rest of the code in that code block.

```
@available(macOS 10.12, *)
struct ColorPreference {
    var bestColor = "blue"
}

func chooseBestColor() -> String {
```

```
    guard #available(macOS 10.12, *) else {
        return "gray"
    }
    let colors = ColorPreference()
    return colors.bestColor
}
```

In the example above, the `ColorPreference` structure requires macOS 10.12 or later. The `chooseBestColor()` function begins with an availability guard. If the platform version is too old to use `ColorPreference`, it falls back to behavior that's always available. After the `guard` statement, you can use APIs that require macOS 10.12 or later.

In addition to `#available`, Swift also supports the opposite check using an unavailability condition. For example, the following two checks do the same thing:

```
if #available(iOS 10, *) {
} else {
    // Fallback code
}

if #unavailable(iOS 10) {
    // Fallback code
}
```

Using the `#unavailable` form helps make your code more readable when the check contains only fallback code.

- Control Flow
- For-In Loops
- While Loops
- While
- Repeat-While
- Conditional Statements
- If
- Switch
- Control Transfer Statements
- Continue
- Break
- Fallthrough
- Labeled Statements
- Early Exit

- Deferred Actions
- Checking API Availability