Control Flow

You wouldn't go out in the cold, pouring rain with your usual flip-flops on (unless you would, no judgements here.) When going outside, you probably tend to choose your outfits based on the weather. If it's raining outside, then you'd wear your rain jacket. If it's sunny outside **and** you're trying to impress someone, then you'd wear your coolest shades. If you're cold, keep layering on clothing until you are not cold anymore.

Swift provides tools so that your code can make this kind of decision-making and task repetition that is essential to programming. Yay weather-appropriate clothing!

True or False?

...you're going to learn about another type in Swift. True! It's called a boolean, or **Bool** for short, and can only have one of two values: true or false. For example,

```
1> let emolga_is_cute = true        Swift guesses this constant to be of type Bool
2> var magikarp_is_useful: Bool = false
3>
4> print("\(emolga_is_cute) vs. \(magikarp_is_useful)!")
```

Output:

```
true vs. false!
```

Comparison Operators

So far, the **operators** you know are `+`, `-`, `*`, and `/`. Those operators create new Ints, Doubles, and Strings based on what type their operands (the values on either side of the operator) are. What about operators that create Bools, though?

==

This operator returns `true` if the two values it is applied to are equal to each other and `false` if they are not equal.

```
let math_is_broken = 0 == 1.8
// false
let donald_adj = "cool"
let donald_is_cool = "cool" ==
   donald_adj
// true
```

!=

This operator returns the opposite of the equality operator: `true` if its two operands are not equal and `false` if they are.

```
let donald_is_muggle = "Donald"
   != "Harry Potter"
// true
```

= Vs. ==

== and = look very similar to each other, but they aren't the same. You should be careful about which you are using.

Just remember, when there are *two* values to compare to each other, use ==. When there is just *one* value to assign, use =.

<
This operator returns `true` if the left value is less than the value on the right and `false` if the left value is greater than or equal to the right value.

>
This operator returns `true` if the left value is greater than the value on the right and `false` if the left value is less than or equal to the right value.

<=
This operator returns `true` if the left value is less than or equal to the value on the right and `false` if the left value is greater than the right value.

Example:

```
let math_is_great = 4 <= 4
// true
```

>=
This operator returns `true` if the left value is greater than or equal to the value on the right and `false` if the left value is less than the right value.

Example:

```
let Donald_is_cooler = "Donald"
   >= "Ash"
// true
```

To Be || !(To Be)

It is the coldest day in the history of your city. You will only go out if you have a long sleeved shirt, arm warmers, sweater, and casual hoodie on. Anything less, and you'd be cold. However, you wouldn't need all of that if you had a super warm winter coat on. If you had that coat, why bother seeing if you have the rest of those clothing items?

Swift, like a lot of programming languages, has logical operators that combine Bool values (true and false) to create one Bool value. These operators make it possible for your code to choose between wearing a lot of layers or just one on a really cold day.

&&
This operator returns `true` if the both Bools being compared by this operator are `true` and `false` if one or both of the Bools are `false`.

```
let im_layered_warm:
   Bool =
   long_sleeved_shirt
   && arm_warmers &&
   sweater &&
   casual_hoodie
```

If any of the above Bools are false, then im_layered_warm will be false

||
This operator returns `true` if the either of the Bools being compared by this operator are `true` and `false` if both of the Bools are `false`.

```
let im_warm =
   winter_coat ||
   im_layered_warm
```

If either (or both) winter_coat or im_layered_warm are true, then im_warm is true

!
This operator is applied to just one value, unlike other logical operators. It returns `true` if the Bool this operator is applied to is `false` and `false` if the Bool is `true`.

```
let im_cold =
   !im_warm
```

If im_warm is true, im_cold is false. If im_warm is false, im_cold is true.
Warning! This is not the same operator as you saw in Lesson 2. This is a new operator in this class.

If you're having trouble understanding this right now, don't worry. It might help to draw a truth table where you can see the outcomes of certain operations on specific values.

For example, let p and q be two Bool constants, our independent constants:

p	q	!p	!q	p && q	p || q
true	true	false	false	true	true
true	false	false	true	false	true
false	true	true	false	false	true
false	false	true	true	false	false

It's basically like creating x and y tables in Algebra 2, but with `true` and `false`.

Operator Precedence

Swift still uses standard PEMDAS, so `8 + 9 / 3 * 17 - 56 + 1.2` still equals `4.2`, but its order of operations includes a lot more operators, like the ones we just learned.

When in doubt, use parentheses.

Operator Precedence
Sorted by which evaluates first

```
! - (negative sign)
* / %
+ - (subtraction)
== != < > <= >=
&&
||
=
```

If Statements

If statements are statements that only run if their condition is true. For example, assuming `x` is declared,

```
1> if x % 2 == 0 {
2>     print("x is even")
3> }
```

`x % 2 == 0` is the condition

`x is even` will only print if `x` is divisible by 2, when `x` divided by 2 has no remainder. What if we also wanted to know if `x` was odd? Well, we *could* write another if statement that would run if `x` was odd, *or* we could attach an else branch to the end of our original if statement.

```
1> if x % 2 == 0 {
2>     print("x is even")
3> } else {
4>     print("x is odd")
5>     print("Try again")
6> }
```

By the nature of an if statement, either one or none of its branches will run, so if `x` is divisible by 2, `x is even` will be outputted and lines 4 and 5 will not run. However, if `x` does not satisfy `x % 2 == 0`, if it is not divisible by 2, then line 2 will be skipped and

the else branch will run. Since the else branch has no condition on it, it runs every time the original if branch does not. It works well here since a number that is not even must be odd.

What if we wanted to know if it was divisible by 3 if not 2? What about divisible by 5? We can add on else if branches: branches that will only run if the above conditions are false and their condition is true.

```
1> if x % 2 == 0 {
2>    print("x is even")
3> } else if x % 3 == 0 {
4>    print("x is divisible by 3")
5> } else if x % 5 == 0 {
6>    print("x is divisible by 5")
7> } else {
8>    print("x is odd and is not divisible by either 3 or 5.")
9> }
```

In this case, if `x` was 15, line 4 would execute. Even though `x` is also divisible by 5, only one branch in an if statement is executed. If `x` was 7, line 8 would execute, because none of the previous conditions were true.

The else branch can seem sort of useless, however, in some situations, so it is possible to forgo it altogether, like so

```
1> if x % 2 == 0 {
2>    print("x is even")
3> } else if x % 3 == 0 {
4>    print("x is divisible by 3")
5> } else if x % 5 == 0 {
6>    print("x is divisible by 5")
7> }
```

However, the else branch can be very useful for debugging purposes, even when it seems impossible that none of the other parts of the if statements would execute. If none of them execute, then you have a problem in your code!

Putting an else statement in with an error message in print() will let you know if you have run into this problem, so you don't spend hours pulling out your hair trying to find it.

TL;DR: The general form of the if statement looks like this:

```
1> if condition {
2>    statements
3> } else if condition {
4>    statements
5> } else {
6>    statements
7> }
```

Note: There is no limit to the amount of code lines/statements you can have in an if statement branch. Do be mindful, however, of keeping your code clean and organized so you can find errors more easily.

While Loops

When we started Swift instruction, we (hopefully) stressed that repeating literals over and over again and not storing them in constants and variables was a recipe for mistakes. The same thing goes for repeating code, especially since, often, you don't know when writing your code how many times you need to repeat some code. Also, making code that can adapt to what input is given will create a better, long-lasting program that can be used for other projects. All of that is why **while loops** are important.

While loops repeat code "while" a Bool condition is true. For example,

```
1> var sum = 0                          sum needs to change later, so we must make it a variable
2> var i = 1                            i needs to change later, so we must make it a variable
3>
4> while i <= 3 {                       i <= 3 is the condition
5>    sum = sum + i
6>    i = i + 1
7> }
8>
9> print(sum)                           prints 6
```

After the program has started, when `sum` and `i` are created, `sum` is 0 and `i` is 1.

Since 1 is less than or equal to 6, the statements in the while loop are executed. On line 5, `sum`'s old value, 0, is used to create a new value for `sum`. This works because the

assignment operator, =, has a lower precedence than, meaning its executed after, addition. The + operator does not actually modify values, so the value it creates needs to be stored in a constant or variable.

`sum` now has the value 0 + 1, or 1, and `i` has the value 1 + 1, or 2.

The code comes back to line 4 to check if `i` is still less than or equal to 3, and its value, 2, is, so the while loop is again executed.

`sum` is assigned the value 1 + 2, or 3, and `i` is assigned the value 2 + 1, also 3. The code returns to line 4 again after this, and the while loop executes once again, since `i`, with its value of 3, is less than or equal to 3. `sum` becomes 3 + 3, or 6; `i` becomes 3 + 1, or 4. The code returns to line 4.

Finally, we come to the point where the condition is no longer true. `i`'s value, 4, is not less than or equal to 3. The while loop is skipped, and the code continues at line 8. At line 9, the value `sum` is printed and we see 6 on the screen.

If `i` had been created with the value 4 instead of 1, the while loop would never have executed, because 4 is not less than or equal to 3, and 0 would have been printed.

Warning: be careful of creating while loop conditions that are always true.

```
1> var sum = 0
2> var i = 4
3>
4> while i >= 3 {                    i will always be greater than 3
5>    sum = sum + i
6>    i = i + 1                      i increases with every iteration of the loop
7> }
8>
9> print(sum)                        Code never reaches this point
```

In the above example, the condition is always true, and the code keeps repeating until the program finally crashes. This will also happen if line 6 wasn't there, as 4 will always be greater than 3. Remember to put in statements in your while loops that allow the while loop's condition to eventually become false.

TL;DR: The general while loop looks like this:

```
1> while condition {
2>    statements
3> }
```