True or False?

Booleans

Another type of variable/constant is the Boolean or **Bool** for short. Bools can take on values of either true or false. For example,

Your code:

```
let emolga_is_cute = true
var magikarp_is_useful: Bool = false

print("\(emolga_is_cute) vs. \(magikarp_is_useful)!")
```

Output:

```
true vs. false!
```

Comparison Operators

So far, the **operators** you know are +, −, ∗, /, and %. Those operators produce Ints, Doubles, and Strings based on what type their operands (the values on either side of the operator) are. What about operators that produce 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.

Example:

```
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.

Example:

```
let donald_name = "Donald"
let is_muggle = donald_name !=
     "Harry Potter"
// true

let this_year_robots_num = 3
let too_cool_for_four_robots =
     this_year_robots_num != 4
```

<

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)

Boolean expressions (any only Boolean expressions) can be combined into one Bool with && (and), || (or), and ! (not).

&&

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`.

||

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`.

!

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

```
var math_is_hard: Bool = true
var math_is_easy: Bool = true
let i_have_mixed_feelings = math_is_hard &&
math_is_easy
// true

var Ash_is_cool = false
let encourage_Ash: Bool = true
Ash_is_cool = Ash_is_cool || encourage_Ash
// true

var you_are_right = false
var i_am_right = !you_are_right
// true
```

If Statements

Another important aspect of good programs is **control flow**, which basically just controlling what code is being run and how many times it is being run. **If statements** are a crucial part of control flow.

If statements specifically control whether or not some piece of code is run based on a Bool condition. It can have else and else if statements attached to it so that if one condition is `false`, other conditions can be checked and run. For example,

```
let x = Double(readLine()!)!

if x % 2 == 0 { // if x is divisible by 2
     print("x is even")
} else {
     print("x is odd")
}
```

== 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 =.

Old Value Creates New Value

You can use the old value of a variable to create the new value for it. The expression on the right side of an = will always be evaluated *before* the assignment to the variable.

This can be used with any type you have learned as long as the operator can be used with it.

Literal Values Have No Intrinsic Type

Notice on the left that an operator is being used with a Double and the number 2. Why is this possible?

First off, we must define what a literal is. A **literal** is a value written exactly as it's meant to be interpreted, such as 2, 4.82, true, or "This is a literal, too!". It's not a variable or constant, as those are essentially aliases for literals.

Second, literals have no type assigned to them. When they are assigned to a variable, they help Swift guess what the variable is supposed to be, but literals themselves have no type.

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,

```
var sum = 0

var i = 1 // Often i will stand for
          // iteration or index
while i <= 100 {
     sum = sum + i
     i = i + 1
}

print(sum)
```

Old Value Creates New Value Even Quicker

If you're anything like me, you're always looking for some sort of neat short cut to writing anything. And if you're like that, you've probably noticed that writing `sum = sum + i` is kind of a hassle, especially when you could be writing `sum += i`!

These two statements mean the same exact thing, which makes the `+=` (or `-=`, `*=`, or `/=`) even cooler. The `+=` just says, "Take the current value of the variable on the left, add whatever value is on the right, and make that new value the current value of the variable on the left." So basically, you can use this new operator with *any* variable, as long as its type allows the first operator!