Introduction to Swift

Constants

When creating a program, one of the basic needs you'll have is the need to *store data* to be used over and over again. Repeating the same value over and over again without storing it is a recipe for mistakes. You are liable to forget to change one of the copies of the value in your code or to mistype the value in one of its copies, leading to code that doesn't do what it is supposed to.

One of the ways to store data in Swift, and one of the preferred ways to do so, is by making a **constant**. A constant stores a value in a name, like x, y, my_constant, myConstant, or num1. To create a constant, type

```
let name = value
```

where `name` is the "word" you'll type to use the value and `value` is something like `"Hello, world!"`, `5`, `6 * 10`, or `-1000.429`, in your code.

Note that a constant has a *constant* value, meaning its value cannot be changed. The value can be used to create a new value in an expression like `my_num + 10`, but you cannot make a constant have a new value.

Also note that you *must* give a constant a value when creating it. You cannot just say `let i` and be done; you have to give it its initial value.

Examples:

```
let x = -5 * 10
let my_name = "Ash"
let theCoolestAnimal = "🐱!!!"
let my_age = 18.667

x = 42 // Error! 'x' is a constant and cannot be reassigned a value
let y // Error! 'y' must be initialized with a value
```

Comments

An important part of making a good program is making the code clear. It's not just important for others when they look at your work and have no idea what's going on because there's no clarity, but it's also important for when you either take a break from your code or have a very long program. Debugging is a huge part of programming and it's important that when you come back to each section of code, even if you have forgotten what it does (and you will, trust me), you are able to tell what it does by how clear you have written it.

One way of being clear in your code is simply making your constant names descriptive, like `let my_team_number = 1678` or `let question_mark = "?"`.

Sometimes descriptive names just aren't enough. Another way of being clear with your code is by actually explaining what each part of your code does and *why* it does it with **comments**. A one line comment looks like this:

```
// The most important thing you can do with comments is explain why!
```

Comments are parts of your code that aren't actually run. They are skipped over/ignored when the code runs. That makes them also useful for when you are debugging your code and testing which part has bugs in it. If you suspect a part of your code is causing problems, try commenting it out if possible.

Commenting a large block of code would be very tedious if you just used one line comments. Luckily, there are such things called multiline comments, which look like this:

```
/* This can also be used for
     /*small*/ bits in your code
     that need to be commented
     out.
*/
```

Types

As you can see, there are different types of values you can have. You can have values with quotation marks (" ") around a variety of characters, values that are whole numbers, and values that have decimal points.

Values with quotation marks around a series of characters, like `"Ash"`, `"🐱!!!"`, and `"Hello, world!"`, are called **String**s.

Values that are whole numbers, like `1`, `2`, `3`, `4`, `5`..., are called Integers, or **Int**s for short.

Finally, values that have decimal points, like `100.0`, `58.23`, `0.111`, and `99.032`, are called **Double**s.

The one of the most important uses for types is knowing what types your constants are. Constants are given the type their initial value has.

Examples:

```
// These constants are Strings
let lead_robot_programmer = "Kelly"
let lead_app_programmer = "Bryton"

// These constants are Ints
let spartan_robotics_num = 971
let the_cheesy_poofs_num = 254

// These constants are Doubles
let my_bulbasaur_iv = 46.7
let my_squirtle_iv = 95.6
```

In Swift, **type safety** is pretty important, meaning that the type a constant is created with is its type forever. It also means you cannot do

```
let wesley_butter_amount = 4
let ash_butter_amount = 0.15

let crepe_butter_amount = wesley_butter_amount + ash_butter_amount
// Error! Binary operator '+' cannot be applied to operands of type
// 'Int' and 'Double'
```

Operators that are used with two values, like `+`, `-`, `*`, `%`, and `/`, cannot be used with two variables of different types. So what do you do when you need to add `wesley_butter_amount` and `ash_butter_amount` for deciding how many crepes to make?

One way to make this situation *butter* is by declaring `wesley_butter_amount` to be a Double from the start. That's where **type annotation** comes in. Instead of just letting Swift guess wrong that `wesley_butter_amount` is an Int, say explicitly that `wesley_butter_amount` is a Double:

```
let wesley_butter_amount: Double = 4
```

What if, though, `wesley_butter_amount` *had* to be an Int for whatever reason? It wouldn't make sense to make `ash_butter_amount` an Int, as when a Double is converted to an Int, its value is **truncated** (the numbers after the decimal point are cut off (i.e. 0.15 becomes 0, -6.7 becomes -6)).

% (Modulo) Operator

How can you tell that a number is divisible by another number? Usually, by looking at the quotient (the answer to the division) and seeing if it is a whole number. But how would you do that in code?

The % operator is very helpful in that regard as it produces the remainder of one number being divided by another. For example,

```
let remainder1 = 4 % 2
// remainder1 = 0, 4 is
// divisible by 2

let remainder 17 % 5
// remainder2 = 2, 5 goes
// into 17 3 times with 2
// units left over
```

Fortunately, there's a way to *temporarily* make a new constant with almost the same value but with a different type. To temporarily create a Double `wesley_butter_amount`, you would write

```
let crepe_butter_amount = Double(wesley_butter_amount) + ash_butter_amount
```

`Type(constant)` changes `constant` into that `Type` for just that instant use. `wesley_butter_amount` is still an Int, but in this expression, it is masked as a Double.

Strings can be added together as well. When they are added together, it is called **concatenating** as the Strings are just attached to the end of one another. For example,

```
let new_swift_text: String = "We are using Swift "
let old_swift_text: String = " instead of Swift "
let sept_2016_released_swift: Int = 3
let mar_2016_released_swift: Double = 2.2

let 1678_swift_use = new_swift_text + sept_2016_released_swift +
     old_swift_text + mar_2016_released_swift + "."
// Error! Types are different and thus cannot be added.

// Instead, write this:
let 1678_swift_use = new_swift_text + String(sept_2016_released_swift)
     + old_swift_text + String(mar_2016_released_swift) + "."
```

The value of `1678_swift_use` is now `"We are using Swift 3 instead of Swift 2.2."` and the constant is of type String.

There is another way to use constants in Strings and that is by writing `\(constant)` in the String itself. For example, instead of making a constant for `new_swift_text` and `old_swift_text` and adding everything together, you could just write

```
let 1678_swift_use2 = "We are using Swift \(sept_2016_released_swift)
     instead of Swift \(mar_2016_released_swift)."
```

Output

You don't really know if `1678_swift_use` equals `1678_swift_use2` or even if both of them equal `"We are using Swift 3 instead of Swift 2.2."`! To really know for sure, you could use `print()`.

`print()` is a function that outputs whatever value is put into its parentheses, (), so that when the code you have written is run, you can see what your program does. For example,

Your code:

```
print(1678_swift_use)
print(1678_swift_use2)
print(8 + 9 / 3 * 17 - 56)
print("Hello, world!")
```

Your output

```
We are using Swift 3 instead of Swift 2.2.
We are using Swift 3 instead of Swift 2.2.
3
Hello, world!
```

Notice how, after each print output, there is a new line, like someone has pressed Enter after each output. That's because `print()` does that! Keep in mind that new line whenever you are using `print()`.

Input

One-sided conversations aren't that great. Being able to listen and respond to your user is much more fun. That's why Swift has the function `readLine()`.

To use the input given to `readLine()`, you would simply assign `readLine()` to a constant, like so

```
let user_fave_color = readLine()! // The ! is necessary
```

Each `readLine()` call takes input as a String from the user up to the next new line and acts as a temporary variable for that input. So if you copied and pasted

```
But if the while I think on thee, dear
friend,
All losses are restor'd and sorrows end.
```

`readLine()` would only become `But if the while I think on thee, dear friend,` for the one time it was called. If you called `readLine()` again, it would read in `All losses are restor'd and sorrows end.`:

```
let sonnet_line1: String = readLine()!
// sonnet_line1 = But if the while I
// think on thee, dear friend,
let sonnet_line2: String = readLine()!
// sonnet_line2 = All losses are restor'd
// and sorrows end.
```

readLine() -> Int or Double

`readLine()` produces a String, so how would you change it into an Int or a Double? Your first inclination may be to do `Int(readLine())` or `Double(readLine())` and that would be partly correct, however, in this case, a ! is needed right after `Int()` and `Double()`. Thus, you would have

```
let inputtedText = readLine()!
let inputtedNum =
      Int(inputtedText))!
```

or

```
let inputtedNum =
      Int(readLine()!)!
```

Variables

One thing is for sure, change is constant. So how do we make up for that in our code's data storage? With **variables**!

Variables are exactly like constants and *can be used exactly like constants* except variables can be given a new value. It is best to use constants, thus, because it is not possible to accidentally change them and it is less likely that you will give them a bad value. But sometimes you need to change the values of your data, so how do you create a variable?

```
var name = value
var name: Type = value
```