Welcome to Swift... Part 2!

Last time we learned about constants, little labeled boxes for us to store values and data in; comments, documentation for keeping your code understandable and clear; types, categories for different kinds of values; and print(), a way of outputting information. Now, we'll be delving deeper into these concepts so that you too can make cool programs with Swift!

Variables

One thing is for sure, our lives are constantly changing. 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
```

or

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

It's good practice to use constants first, and change the constants that you cannot get around changing into variables later. It's a simple change from

```
let name = value
```

to

```
var name = value
```

Type Conversion

In the last lesson, we saw the basic operators, `+`, `-`, `*`, and `/`, and a little bit about how they work. We also saw that they don't work with values of differing types. So how would you go about multiplying an Int and a Double together, like when you're trying to see how many meters of exactly 0.5 m long wood blocks you have outside of your shed?

```
let wood_block_length = 0.5
let num_outside_wood_block = 36
let num_shed_wood_block = 5
```

There's a way to temporarily make a new constant with almost the same value as the constant you want to change but with a

Literals Have No Intrinsic Type

While you can't add an Int and a Double constant together without a conversion, you can add the number 2 to a Double, the number 2.3 to an Int, or just do 2 + 2.3.

This is because 2 and 2.3 (and any other number or series of characters in quotes) are **literals.** They aren't constants. They are the actual values, unassigned to any constant.

Yes, when you assign a literal to a constant without explicitly stating the type of the constant, Swift guesses what type the constant is from the literal, but that does not mean the literal has its own type. Swift just finds the best match for the constant based on the literal.

different type. To do this, you would simply write `Type(constant)`, where `constant` is the name of the constant you wish to create a temporary converted copy of, and `Type` is the type of the new copy. For example,

```
1> let total_outside_wood_block_length
      = wood_block_length *
      Double(num_outside_wood_block)
```
total_outside_wood_block_length is a Double

```
2> let total_num_wood_block =
      num_outside_wood_block +
      num_shed_wood_block
```
total_num_wood_block is an Int
num_outside_wood_block is still an Int

Notice how we did not convert `wood_block_length` to an Int instead. We did this because when a Double is converted to an Int, its value is **truncated**, meaning everything after the decimal point is, in a sense, lopped off.

> Truncating is not the same thing as rounding, or even rounding down. If the value 0.67 is truncated, it becomes 0, not 1. Although rounding down gives you the same answer as truncating when the number is positive, it does not work when the number is negative. When -6.7 is truncated, it becomes -6, not -7. Be careful of truncating errors.

Remainder Operator

How can you tell that a number is divisible by another number? By doing long division, and seeing if the remainder is 0.

In Swift code, to get the remainder of `dividend` being divided by `divisor`, we use the remainder operator, `%`, like so:

```
let remainder = dividend % divisor
```

divisor	divisor	divisor	remainder
dividend			

With actual numbers, that would look like this:

```
let remainder = 9 % 4 // = 1
```

4	4	1
9		

Because remainder does not equal 0, you can quickly tell nine is not divisible by four in your code.

More examples:

```
1> let num_points = 4
```

```
2> let remainder1 = num_points % 2
```
= 0, four is divisible by two

```
3> let remainder2 = -9 % num_points
```
= -1, negative nine is not divisible by four

```
4> let remainder3 = 9 % -num_points
```
= 1, the absolute value of divisor is always taken

Exponents

Be careful of assuming what certain characters might do in arithmetic. For example, the character `^` is not for exponents. It is called the bitwise XOR operator, a.k.a. probably not what you want. Instead, on a line at the top of your code, write `import Foundation`. Then you can use `pow(base, power)` wherever you need exponents.

Note: if you plan to use pow() with Doubles or as a Double, make sure to declare the constant made from pow() is explicitly written as a Double, or else you will get an error.

1>	`import Foundation`	Don't forget this or else you won't be able to use pow()

…

1>	`let num_kittens = pow(2, 5)`	= 32
2>	`let num_kitten_food = pow(2, 5.5)`	= 45.25483399593904
3>	`let num_puppy_food = pow(2.5, 5)`	Error! Ambiguous use of 'pow'
4>	`let num_lizard_food: Double = pow(2.5, 5)`	= 97.65625
5>	`let num_bunny_food = pow(3, 2) + 9.4`	Error! Ambiguous use of 'pow'
6>	`let num_snake_food: Double = pow(3, 2) + 9.4`	= 18.4

As you can see, pow() gets a little complicated with Doubles. It's best to just give *power* to the Doubles when using pow() with Doubles.

Concatenation

Strings can be added together as well. This is called **concatenation**, and simply results in the second String being attached to the first String in a new String. For example,

1>	`let current_month = "December"`	Type = String
2>	`let current_date = 22`	Type = Int
3>	`let hours_left_today = 24 - 19.033`	Type = Double
4>	`let birthday_excitement1 = "Today is " + current_month + " " + current_date + " and there are " + hours_left_today + " hours left until my birthday!"`	Error! Types are different and thus cannot be added.
5>	`let birthday_excitement2 = "Today is " + current_month + " " + String(current_date) + " and there are " + String(hours_left_today) + " hours left until my birthday!"`	Value = "Today is December 22 and there are 4.967 hours left until my birthday!" Type = String

Special Characters & String Interpolation

To make the value of birthday_excitement2 easier to see in your code, you can use **string interpolation** and write instead,

```
let birthday_excitement2 = "Today is \(current_month) \
   (current_date) and there are \(hours_left_today) hours left
   until my birthday!"
```

The backslash character, `\`, is a special character in Strings that allows the programmer to write in special characters, constant and variable values, and do certain things. For example,

Code:

```
1> let wise_words = "Einstein once said,
     \n\"Imagination is more important
     than knowledge.\""
```
\n creates a new line
\" is the character "

```
2> let more_wise_words = "The \'\\\'
     character will take you far."
```
\' is the character '
\\ is the character \

```
3> let all_wise_words = "\(wise_words)\n
     \(more_wise_words)"
```
\(constant) includes the value of constant inside a String. This is called string interpolation.

```
4> print(all_wise_words)
```

Output:

```
Einstein once said,
"Imagination is more important than knowledge."
The '\' character will take you far.
```

Input

One-sided conversations aren't that great. Code that is able to listen and respond to its 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_favorite_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 String constant 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."`

Code:

```
print(readLine()!)
print(readLine()!)
```

Input:

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

Output:

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

`readLine()` becomes a String, even if all of the characters inputted into the console are digits, so how would you change the input into an Int or Double? Your first inclination may be to do `Int(readLine()!)` or `Double(readLine()!)` and that would be partly correct, however, in this case, an `!` is needed right after `Int()` and `Double()` as well, which you will learn more about later. Thus, you would have

```
let inputted_text = readLine()!
let inputted_num = Int(inputted_text)!
```

or, a writing it more concisely,

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