Welcome to Swift

When it comes to learning programming for the first time, the task can seem almost daunting. Swift makes it easier for programmers of all levels to understand and play with powerful features.

While being safer and more modern, Swift is similar enough to C and Objective-C so that, once you know Swift, you can easily learn other languages. Swift's applications, and the applications you can make with it, are endless.

Constants

When creating a program, your top basic need will be to *store data* so that it may be used over and over again. Simply 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 at least once, leading to code that doesn't do what it is supposed to and is hard to **debug**.

> Debugging a program or a piece of code happens when you find a problem, or bug, in how the problem works. To debug the code, you need to find the source of the problem, too often a typo, and correcting the problem or finding a different way to approach the task you want the code to complete.

One of the preferred ways to store data in Swift is 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 keyword you'll type to use the value and `value` is information like `"Hello, world!"`, `0`, `-6 * 10`, or `-1000.429`. Think of it like a dictionary. The name is the word for the idea, and the value is the definition, or the actual idea.

As its name suggests, the value of a constant cannot be changed and cannot be empty. You can use the constant's value to create a new value in an expression, like `todays_date + 7`, but its own value cannot change.

Examples:

```
let x = -5 * 10
let my_name = "Ash"
let favoriteAnimal = "🐱!"
let my_age = 19.4167
```

Examples with errors:

5>	`x = 42`	Error! 'x' is a constant and cannot be reassigned a value
6>	`let y`	Error! 'y' must be initialized with a value
7>	`let my_age = 21.1`	Error! Invalid redeclaration of 'my_age'
8>	`let 1st_cat = "Koby"`	Error! Constant name cannot start with a digit

Comments

An important part of making a good, working program is making the code clear and easy to understand. It's not just important because you want others reviewing your code for errors to be able to understand what's going on, it's also important for when you either take a break from your code or you have a very long program. Debugging is a huge part of programming, so it's essential 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 the code does by how clear you have written it.

One way of making your code easy to follow is by simply making your constant names descriptive, like

```
let national_pokedex_count = 802
   let my_cats_name = "Ascii"
```

Many times descriptive names just aren't enough. To add more clarity, actually explain what each part of your code does and *why* it does it with **comments**. A one line comment looks like this:

```
// The program skips over everything on the line after the slashes
```

Comments are parts of your code that aren't actually run. They are skipped over/ignored when the code runs. This function is extremely useful for when you are debugging your code and testing which parts have bugs in them. If you suspect a part of your code is causing problem, 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 data you can have in your code. You can have values with quotation marks (" ") around a variety of characters, values that are whole numbers, and values that are decimal numbers.

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.

One of the most important uses for types is knowing what types your constants are. Constants are given the type their initial value has or looks like.

Examples:

```
// These constants are Strings
let chikorita_type = "Grass"
let bayleef_type = chikorita_type
                // bayleef_type now has the value "Grass"

// These constants are Ints
let year_of_luigi = 2015
let year_of_creepy_clowns = 2016

// These constants are Doubles
let hours_sleep_expectation = 8.0
let hours_sleep_reality = 4.5
```

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

1>	`let your_butter_amount = 4`	Swift guesses this constant to be of type Int
2>	`let my_butter_amount = 0.15`	Swift guesses this constant to be of type Double
3>	`let crepe_butter_amount = your_butter-amount + my_butter_amount`	Error! Binary operator '+' cannot be applied to operands of type 'Int' and 'Double'

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

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

```
    let your_butter_amount: Double = 4
```

What if, though, `your_butter_amount` *had* to be an Int for whatever reason? It wouldn't make sense to make `my_butter_amount` an Int, as when a Double is converted to an Int, its value is **truncated**.

% (Modulo) Operator

How can you tell that a number is divisible by another number? Usually, by looking at the quotient 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.

```
let remainder1 = 4 % 2 // = 0
let remainder2 = 17 % 5 // = 2
```

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 constant with the same value as `your_butter_amount`, you would write

```
let crepe_butter_amount =
    Double(your_butter_amount) + my_butter_amount
```

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

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,

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.

`let current_month = "December"`	Type = String
`let current_date = 22`	Type = Int
`let hours_left_today = 24 – 19.033`	Type = Double
`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.
`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

It can be difficult to see that `birthday_excitement2` ends up having that value, so Swift has another way of putting constants into Strings. Simply write `\(constant)` in the String itself, like so

```
let birthday_excitement3 = "Today is \(current_month)
    \(current_date) and there are \(hours_left_today) hours left
    until my birthday!"
```

Output

It's a little difficult to know for sure that `birthday_excitement2` has the same value as `birthday_excitement3` or even if both of them equal `"Today is December 22 and there are 4.967 hours left until my birthday!"` 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 might say,

```
print(birthday_excitement2)
print(birthday_excitement3)
print(8 + 9 / 3 * 17 - 56 + 1.2)
print("Hello, world!")
```

Standard PEMDAS still applies.

and your **console**, the display screen where Swift sends output from `print()` and where often times it gets input from, will say

```
Today is December 22 and there are 4.967 hours left until my
birthday!
Today is December 22 and there are 4.967 hours left until my
birthday!
4.2
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_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 party correct, however, in this case, an `!` is needed right after `Int()` and `Double()` as well. 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()!)!
```

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
```