Welcome to Swift

Learning programming for the first time can seem daunting. Swift makes it easier for programmers of all levels to learn basic concepts and play with powerful features.

While being more modern, Swift is similar enough to a common language C 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 most basic need will be to *store data* so that it may be used over and over again. Repeatedly typing the same value over and over again 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 involves finding a problem, or bug, in the program and correcting it.

Make sure the names of your constants are indicative of the values they hold! You wouldn't label a box full of spiders "x". Save yourself from an unpleasant surprise 🕷 .

One of the preferred ways to store data in Swift to make 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 box. The name is the label for the box, and the value is what is actually stored inside of it.

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:

```
1>  let x = -5 * 10
2>  let my_name = "Ash"
3>  let favoriteAnimal = "🐱!"
4>  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 program is making the code easy to understand. It's important for when you want others reviewing your code to be able to understand what's going on and for when you take a break from your code or have a very long program. If your code is unclear, it makes it that much harder to find usually easy-to-fix errors.

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 two slashes
```

Comments are parts of your code that aren't actually run. They are skipped over/ ignored when the code runs. This trait 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 that are whole numbers, like `1`, `2`, `3`, `4`, `5`..., are called integers, or **Int**s for short.

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

Finally, values with quotation marks around a series of characters, like `"Ash"`, `"🐱!"`, and `"Hello, world!"`, are called **String**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 Ints
let year_of_luigi = 2015
let year_of_creepy_clowns = 2016
```

```
// These constants are Doubles
let meters_to_goal = 8.0
let meters_to_safety_zone = 4 + 0.5

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

In Swift, the concept of **type safety** is pretty important. Type safety means that the type a constant is created as is its type for the constant's entire lifespan, helping you avoid errors like trying to divide "Beyoncé" by 3. 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 are called binary operators and include `+`, `−`, `*`, and `/`. These operators 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
```

And now you can have your crepes and eat them too!

Output

It's a little difficult to know for sure what the values of some constants are. 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,

```
let amazement = "Woah!!"
print(amazement)
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

```
Woah!!
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 there will be a new line whenever you are using `print()`.