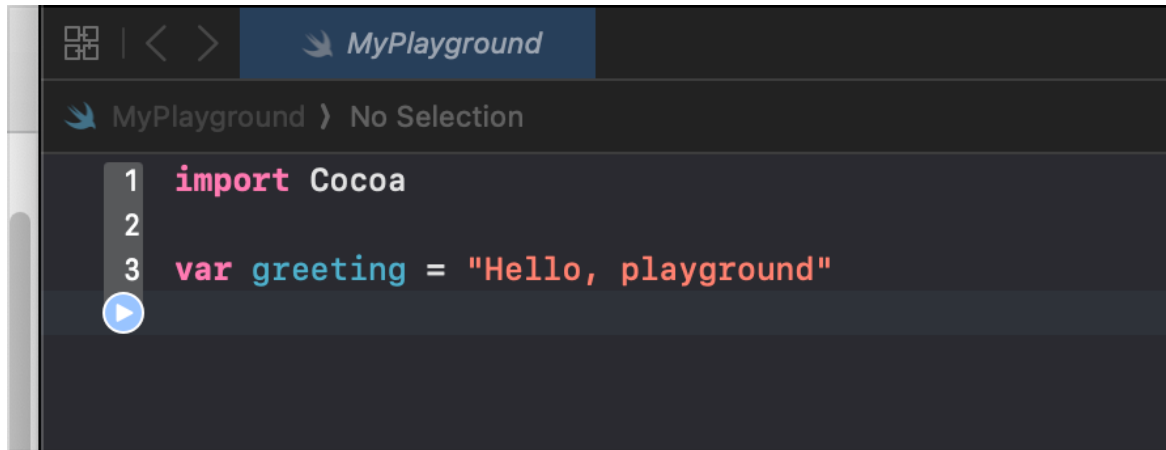


DAY 1



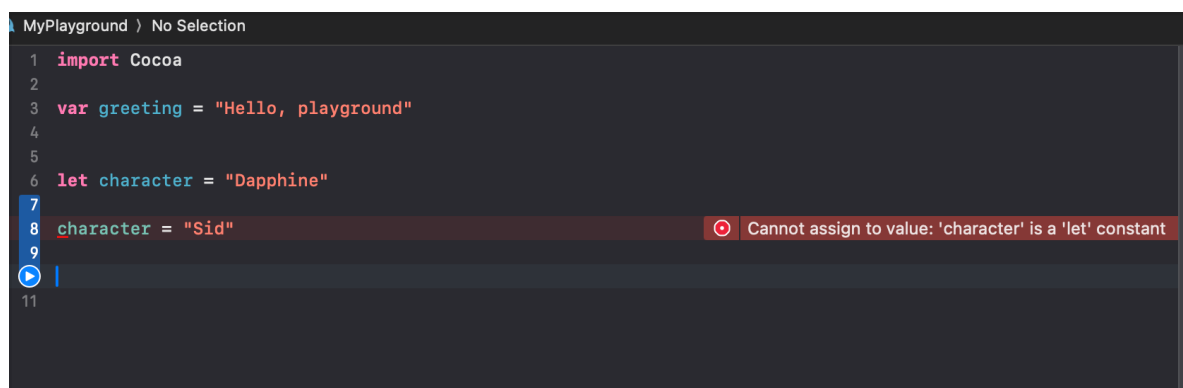
```
1 import Cocoa
2
3 var greeting = "Hello, playground"
```

Here , semicolon is not necessary .We can put it but generally we don't .

We only put semicolon when w e have to put 2 lines in one .

How to create variables and constants

```
var name = "Ted"  
name = "Rebecca"  
name = "Keeley"
```



The screenshot shows a Swift playground window titled "MyPlayground" with "No Selection". The code is as follows:

```
1 import Cocoa  
2  
3 var greeting = "Hello, playground"  
4  
5  
6 let character = "Daphne"  
7  
8 character = "Sid"  
9  
10  
11
```

A red error message is displayed on line 8: "Cannot assign to value: 'character' is a 'let' constant". A blue play button icon is visible on the left margin next to line 8.

In swift we assign constant with let.

```
MyPlayground } No Selection
1 import Cocoa
2
3 var greeting = "Hello, playground"
4
5
6 let character = "Daphne"
7
8 print(greeting)
9 print(character)
```

"Hello, playground"

"Daphne"

"Hello, playground\n"

"Daphne\n"

General tip : Use const more so that code is optimised better so that swift knows that value is not going to be changed. Also it prevents developer from changing value by accident.

Swift's strings start and end with double quotes, but what you put inside those quotes is down to you. You can use short pieces of alphabetic text, like this:

```
let actor = "Denzel Washington"
```

You can use punctuation, emoji and other characters, like this:

```
let filename = "paris.jpg"
let result = "★ You win! ★"
```

And you can even use other double quotes inside your string, as long as you're careful to put a backslash before them so that Swift understands they are *inside* the string rather than *ending* the string:

```
let quote = "Then he tapped a sign saying \"Believe\" and walk
```

Don't worry – if you miss off the backslash, Swift will be sure to shout loudly that your code isn't quite right.

```
10 ~
11 ~
12 let stat = "how are you mister \"Bean\" ji.?"
13 ~
14 print(stat)
15 ~
```

3 Unprintable ASCII character found in source file

This answer is for the people who are using windows keyboard with mac system.

3

While typing you might have unknowingly clicked the **right click** key in your keyboard, which will create the Unprintable ASCII character that is causing the **Unprintable ASCII character found in source file** error.

To check this go to find and click the **right click** key and you can find them in your class and delete those characters and the error will be gone.

Thanks

```
16  
17  
18  
19 //let movie = "A day in the life of  
20 //apple  
21 //engineer"  
22  
23  
24 let movie = """  
25  
26 A day in the life  
27 of apple engineer  
28  
29 """  
30
```

This is how multiline strings are written.

```
34
35 let actor = "Siddharth"
36
37 print(actor.count)
```

"Siddharth"

"9\n"

```
print(actor.count)
```

Because **actor** has the text "Denzel Washington", that will print 17 – one for each letter in the name, plus the space in the middle.

You don't need to print the length of a string directly if you don't want to – you can assign it to another constant, like this:

```
let nameLength = actor.count
print(nameLength)
```

The second useful piece of functionality is **uppercase()**, which sends back the same string except every one of its letter is uppercased:

```
print(result.uppercase())
```

Yes, the open and close parentheses are needed here but *aren't* needed with **count**. The reason for this will become clearer as you learn, but at this early stage in your Swift learning the distinction is best explained like this: if you're asking Swift to read some data you don't need the parentheses, but if you're asking Swift to do some work you *do*. That's not wholly true as you'll learn later, but it's enough to get you moving forward for now.

The last piece of helpful string functionality is called **hasPrefix()**, and lets us know whether a string starts with some letters of our choosing:

```
print(movie.hasPrefix("A day"))
```

There's also a **hasSuffix()** counterpart, which checks whether a string ends with some text:

```
print(filename.hasSuffix(".jpg"))
```

This will give answer in true or false.

```
23 ~
24 let movie = """
25 A day in the life
26 of apple engineer
27 ~
28 """
29 ~
30 ~
31 print(movie)
32 ~
33 ~
34 let actor = "Siddharth"
35 ~
36 print(actor.count)
37 print(actor.uppercased())
38 ~
39 ~
40 ~
41 ~
42 print(movie.hasPrefix("A day"))
```

```
"A day in the life\nof apple engineer\n"
"A day in the life\nof apple engineer\n\n"
"Siddharth"
9
SIDDHARTH
true
```

Why does Swift need multi-line strings?

Paul Hudson [@twostraws](#) May 28th 2020

Updated for Xcode 16

Swift's standard strings start and end with quotes, but must never contain any line breaks. For example, this is a standard string:

```
var quote = "Change the world by being yourself"
```

That works fine for small pieces of text, but becomes ugly in source code if you have lots of text you want to store. That's where multi-line strings come in: if you use triple quotes you can write your strings across as many lines as you need, which means the text remains easy to read in your code:

```
var burns = """
The best laid schemes
O' mice and men
Gang aft agley
"""
```

Swift sees those line breaks in your string as being part of the text itself, so that string actually contains three lines.

Tip: Sometimes you will want to have long strings of text in your code without using multiple lines, but this is quite rare. Specifically, this is most commonly important if you plan to share your code with others – if they see an error message in your program they might want to search your code for it, and if you've split it across multiple lines their search might fail.


```
let score = 10
```

Integers can be really big – past billions, past trillions, past quadrillions, and well into *quintillions*, but they can be really small too – they can hold *negative* numbers up to quintillions.

When you're writing out numbers by hand, it can be hard to see quite what's going on. For example, what number is this?

```
let reallyBig = 100000000
```

If we were writing that out by hand we'd probably write "100,000,000" at which point it's clear that the number is 100 million. Swift has something similar: you can use underscores, `_`, to break up numbers however you want.

So, we could change our previous code to this:

```
let reallyBig = 100_000_000
```

THESE FORMATS ARE ONLY ON THE SWIFT BOOKS AND DOCUMENTS

If we were writing that out by hand we'd probably write "100,000,000" at which point it's clear that the number is 100 million. Swift has something similar: you can use underscores, `_`, to break up numbers however you want.

So, we could change our previous code to this:

```
let reallyBig = 100_000_000
```

Swift doesn't actually care about the underscores, so if you wanted you could write this instead:

```
let reallyBig = 1_00_00_00_00_00
```

The end result is the same: `reallyBig` gets set to an integer with the value of 100,000,000.

```
let lowerScore = score - 2
let higherScore = score + 10
let doubledScore = score * 2
let squaredScore = score * score
let halvedScore = score / 2
print(score)
```

Rather than making new constants each time, Swift has some special operations that adjust an integer somehow and assigns the result back to the original number.

For example, this creates a **counter** variable equal to 10, then adds 5 more to it:

```
var counter = 10
counter = counter + 5
```

Rather than writing **counter = counter + 5**, you can use the shorthand operator **+=**, which adds a number directly to the integer in question:

```
counter += 5
print(counter)
```

That does exactly the same thing, just with less typing. We call these *compound assignment operators*, and they come in other forms:

```
counter *= 2
print(counter)
counter -= 10
print(counter)
counter /= 2
print(counter)
```

Before we're done with integers, I want to mention one last thing: like strings, integers have some useful functionality attached. For example, you can call **isMultiple(of:)** on an integer to find out whether it's a multiple of another integer.

So, we could ask whether 120 is a multiple of three like this:

```
let number = 120
print(number.isMultiple(of: 3))
```

I'm calling **isMultiple(of:)** on a constant there, but you can just use the number directly if you want:

```
print(120.isMultiple(of: 3))
```

This **isMultiple of** will give value true or false .

Strings and integers

Question 2/6: Which code creates an integer?

Hint: Click to show.

Option 1:

```
var speed = 88
```

This creates an integer called speed.

Option 2:

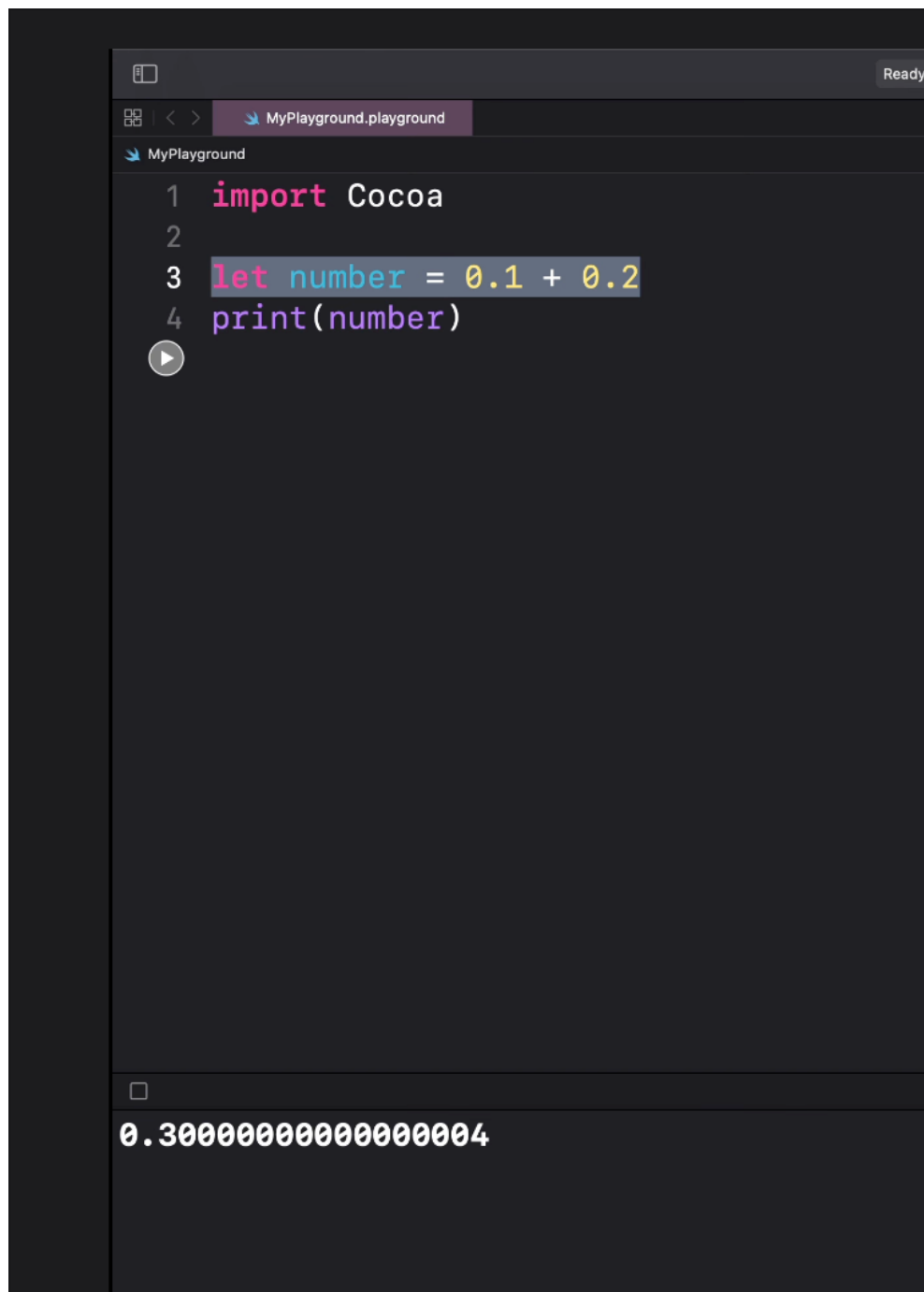
```
var age = "23"
```

Even though this has a number inside the quotes, this still creates a string.

Choose Option 1

Choose Option 2

Correct!



The screenshot shows a Jupyter Notebook interface with a dark theme. The top bar includes a 'Ready' status indicator. The notebook's title bar reads 'MyPlayground.playground'. Below the title bar, the code editor contains the following Python code:

```
1 import Cocoa
2
3 let number = 0.1 + 0.2
4 print(number)
```

A play button icon is visible to the left of the code. The output area at the bottom of the notebook displays the result of the print statement:

```
0.30000000000000004
```

This will print 0.30000000000000004 instead of 0.3

Second, Swift considers decimals to be a wholly different type of data to integers, which means you can't mix them together. After all, integers are always 100% accurate, whereas decimals are not, so Swift won't let you put the two of them together unless you specifically ask for it to happen.

In practice, this means you can't do things like adding an integer to a decimal, so this kind of code will produce an error:

```
let a = 1
let b = 2.0
let c = a + b
```

```
48
49 let a = 1
50 let b = 2.1
51 let c = a + b
    print(c)
```

Binary operator '+' cannot be applied to operands of type 'Int' and 'Double'

Yes, we can see that **b** is really just the integer 2 masquerading as a decimal, but Swift still won't allow that code to run. This is called *type safety*: Swift won't let us mix different types of data by accident.

If you want that to happen you need to tell Swift explicitly that it should either treat the **Double** inside **b** as an **Int**:

```
let c = a + Int(b)
```

```
47 ↵
48 ↵
49 let a = 1 ↵
50 let b = 2.1 ↵
51 let c = a + Int(b) ↵
52 print(c) ↵
```

☐ 1
☐ 2.1
☐ 3
☐ "3\n"

```
47 ↵
48 ↵
49 let a = 1 ↵
50 let b = 2.9 ↵
51 let c = a + Int(b) ↵
52 print(c) ↵
```

☐ 1
☐ 2.9
☐ 3
☐ "3\n"

Int takes floor of the number .

```
47 ↵
48 ↵
49 let a = 1 ↵
50 let b = 2.9 ↵
51 let c = a + Int(b) ↵
52 print(c) ↵
53 ↵
54 let d = Double(a) + b ↵
55 print(d) ↵
```

☐ 1
☐ 2.9
☐ 3
☐ "3\n"
☐ 3.9
☐ "3.9\n"

Or treat the **Int** inside **a** as a **Double**:

```
let c = Double(a) + b
```

Third, Swift decides whether you wanted to create a **Double** or an **Int** based on the number you provide – if there's a dot in there, you have a **Double**, otherwise it's an **Int**. Yes, even if the numbers after the dot are 0.

So:

```
let double1 = 3.1
let double2 = 3131.3131
let double3 = 3.0
let int1 = 3
```

Third, Swift decides whether you wanted to create a **Double** or an **Int** based on the number you provide – if there's a dot in there, you have a **Double**, otherwise it's an **Int**. Yes, even if the numbers after the dot are 0.

So:

```
let double1 = 3.1
let double2 = 3131.3131
let double3 = 3.0
let int1 = 3
```

Combined with type safety, this means that once Swift has decided what data type a constant or variable holds, it must always hold that same data type. That means this code is fine:

```
var name = "Nicolas Cage"
name = "John Travolta"
```

But this kind of code is not:

```
var name = "Nicolas Cage"
name = 57
```

```

54 var d = Double(a) + b
55 print(d)
56
57
58
59 d = "Astro"
60 print(d)

```

Cannot assign value of type 'String' to type 'Double'

Finally, decimal numbers have the same range of operators and compound assignment operators as integers:

```

var rating = 5.0
rating *= 2

```

Many older APIs use a slightly different way of storing decimal numbers, called **CGFloat**. Fortunately, Swift lets us use regular **Double** numbers everywhere a **CGFloat** is expected, so although you will see **CGFloat** appear from time to time you can just ignore it.

In case you were curious, the reason floating-point numbers are complex is because computers are trying to use binary to store complicated numbers. For example, if you divide 1 by 3 we know you get 1/3, but that can't be stored in binary so the system is designed to create very close approximations. It's extremely efficient, and the error is so small it's usually irrelevant, but at least you know why Swift doesn't let us mix **Int** and **Double** by accident!

For example, this creates a new variable called `meaningOfLife` equal to 42:

```
var meaningOfLife = 42
```

Because we assigned 42 as the initial value of `meaningOfLife`, Swift will assign it the type *integer* – a whole number. It's a variable, which means we can change its value as often as we need to, but we *can't* change its type: it will always be an integer.