# LEARN TO CODE USING SWIFT

The absolute beginner's programming book

By Kitti Bödecs and Tibor Bödecs

# Optionals

So far, the values we've used in our programs were written directly into the code — fixed and predictable. But in many real-world situations, we don't know in advance what kind of data the program will work with. We might want the user to enter their name, choose a number, or respond to a question.

To make this possible, Swift gives us a way to read input.
However, there's a small catch: user input isn't always guaranteed. Someone might hit Return without typing anything, or provide something unexpected. That's where optionals come in.

This chapter is the first step into writing more dynamic and interactive Swift programs, while also learning how Swift keeps your code safe when data might be missing or invalid.

You'll learn how to ask for input from the user, how to deal with the fact that input can sometimes be missing, and how Swift helps you write reliable code even when you don't have all the answers up front.

By the end of this chapter, you'll be able to:

- Read a text input from the user using the `readLine` function.
- Understand why `readLine` doesn't always give you a straightforward value.
- Learn what an optional is and how to safely work with it.
- Write programs that respond to user input with appropriate logic.

## Lessons

# The nil value

In Swift, every variable must have a value before you use it. The process of giving a variable or constant its first value is called *initialization*.

So you must initialize every variable before using it. This is because Swift is a safe language and it enforces strict rules around initialization to ensure memory safety and prevent crashes because of missing or unexpected data. If a variable or constant is accessed without being initialized, the compiler will produce an error to help you catch potential issues early:

```
var name: String
print(name) // Variable 'name' used before being initialized
```

If you try to run this code, the compiler will raise an error because `name` is declared but hasn't been assigned an initial value before it is used in the print statement.

Even if you don't yet know the exact value a variable should hold, Swift still requires it to be in a valid state. You can do this by giving it a default value, or by making it optional to indicate that *it might not have a value at all*.

Optionals are Swift's way to handle situations where a variable might not have a value at all. They allow you to declare a variable that can either hold a value or be `nil` (no value).

This helps you write safer code by making it clear when a value might be missing — without breaking Swift's rules about keeping things safe and predictable.

# Optionals and nil values

So an optional in Swift is a type that can either hold a value or have no value (no value is represented as `nil`). It's like a box that might be empty or have something inside.

An optional always has an underlying data type, such as `String`, `Int`, `Double`, etc. You need to declare this data type explicitly, even if you don't know the exact value just yet.

An optional value is indicated by the ? symbol, like this:

```
var myOptionalStringVariable: String?
print(myOptionalStringVariable) // nil
// Warning: Expression implicitly coerced from 'String?' to 'Any'
```

If you define an optional variable, Swift automatically gives it a value of `nil` if you don't set one yourself.

However, constants (declared with `let`) work a bit differently. Once a constant gets a value, it can't be changed — so you must assign a value (like `nil` or a real value) at the time you create it. Swift won't automatically set it to `nil` like it does with variables.

```
let x: Int? = nil
print(x) // nil
// Warning: Expression implicitly coerced from 'Int?' to 'Any'
```

Optionals and constants seem contradictory by nature, but a constant can be an optional too. A constant that's an optional needs to be assigned a value during the initialization process. That value can be `nil`, or some other value.

Once assigned, it is fixed at that value because it is a constant. In such cases, `nil` acts as a "this property intentionally left blank" indicator, which cannot be changed later.

That's about optional variables and constants.

# Non-optionals vs optionals

Non-optional variables must always start with a decent value. Swift won't allow you to leave them empty — they must be set when you create them. Otherwise, you'll get a compiler error:

```
// this is not possible, x is a non-optional integer
var x: Int
print(x) // Variable 'x' used before being initialized
```

The main difference between the `Int` and the `Int?` type is that the non-optional type is guaranteed to have a value.

When you print an optional value, Swift may show a compiler warning. This happens because the print function expects a regular, non-optional value:

```
// this is possible, x is an optional integer
var x: Int?
print(x) // Expression implicitly coerced from 'Int?' to 'Any'
```

To fix this warning, provide a default value using the nil coalescing operator `??`. This will be explained in more detail later in this section:

```
var userName: String? = "Kitti"
print(userName ?? "") // Kitti

userName = nil
print(userName ?? "") // ""
```

Sometimes, you know the value you want to assign to an optional at first, but later the variable may become `nil`. For example, as shown above, imagine a user is logged in, so you know the value of `userName`. When the user logs out, `userName` becomes `nil`.

> When we're talking about optional strings, it's important to note that `nil` is not equal to an empty string `""`. An empty string is an actual value of a string which happens to be empty.

When working with optionals, you can use a fallback value to convert the optional into a regular value. To do this, use the nil coalescing operator. Let's take a closer look at how this operator works.

# The nil coalescing operator

When something might be `nil`, we often want to provide a default value as a fallback. The nil coalescing operator `??` lets us do that.

The nil coalescing operator `??` provides a default value when an optional is `nil`. It unwraps the optional if it has a value or returns the provided default value if it doesn't.

To unwrap an optional means accessing the value inside the optional if it exists. Since an optional might not have a value (it could be `nil`), Swift forces you to unwrap it safely before using it as a non-optional value. This prevents unexpected crashes when you try to use a `nil` value.

Imagine it like having a look into the above mentioned box, to see if there is the value we want to work with or not. If it contains the value, we can start using it. If it doesn't contain the value we're looking for, we safely checked it so that the program won't crash due to this.

This is a very simple example:

```
var favoriteColor: String? = nil
let colorToUse = favoriteColor ?? "blue"

print(colorToUse) // blue
```

It's like telling Swift to use `favoriteColor` — or "blue" if it's `nil`.

Let's look at the previous `userName` example again, this time with a small change:

```
var userName: String? = "Anne"
let name: String = userName ?? "Guest"
print(name) // "Anne"

// user logs out
userName = nil
print(userName ?? "Guest") // "Guest"
```

In this example, `userName` is an optional variable. It can either store a value or be `nil`. At first, `userName` contains the name Anne. When we check if `userName` has a value, it does — so `name` is set to Anne. Because `userName` had a value and we used the `??` operator, Anne is printed. When the user logs out, `userName` becomes `nil`. Now, since `userName` is `nil`, Swift prints Guest instead.

As these examples show, the nil coalescing operator lets you create a new constant or variable and provide a fallback value if the original value is `nil`.

# Type conversion

As you learned earlier, you need to be conscious about types and type conversions. Swift can't automatically convert between types because it could result in a loss of data or precision.

> Type conversion in Swift is the process of converting a value from one type to another and it needs to be explicit.

While some conversions might fail and result in an optional value, others are guaranteed to succeed because they involve a safe, predictable transformation.

Let's look at this example:

```
let doubleValue: Double = 3.14
let intValue: Int = Int(doubleValue)  // Type conversion from Double to Int
print(intValue)  // Output: 3
```

The conversion from a `Double` to an `Int` will always work, so `intValue` is not an optional because there is no uncertainty. Swift safely removes the decimal part, but this also means you lose any precision after the decimal point. The conversion simply cuts off everything after the decimal point, without rounding.

Sometimes, when you try to convert one type into another, the operation might fail. Swift is built to handle this uncertainty safely — instead of crashing your program, it gives you back an optional, which either contains a value or is `nil`.

```
let str = "5"
let num = Int(str)
print(num) // Optional(5)
```

In the example above `str` constant of `String` type is defined and it has a value assigned. Then another constant `num` is created, which has the value of `str` but converted to `Int` type. Since we cannot be 100% sure that the value can be converted from `String` type to `Int`, the result will be an optional integer.

You'll often need to do type conversion when working with data from different sources (e.g., converting user input from a string to an integer, double or boolean value). There's a nearly exhaustive list demonstrating these conversions:

```swift
let boolValue = true
let boolToString = String(boolValue) // "true"

// String to Bool
let stringValueForBool = "true"
let stringToBool = Bool(stringValueForBool) // true

// Int to String
let intValue = 42
let intToString = String(intValue) // "42"

// String to Int
let stringValueForInt = "123"
let stringToInt = Int(stringValueForInt) // 123 (optional)

// Double to String
let doubleValue = 3.14
let doubleToString = String(doubleValue) // "3.14"

// String to Double
let stringValueForDouble = "3.14"
let stringToDouble = Double(stringValueForDouble) // 3.14 (optional)

// Int to Double
let intToDouble = Double(intValue) // 42.0

// Double to Int
let doubleToInt = Int(doubleValue) // 3 (truncated)

// Int to Bool
let intValueForBool = 1
let intToBool = intValueForBool != 0 // true
```

Converting a `Bool` value to an `Int` is a more interesting example, so let's discuss it a bit more:

```
// Bool to Int
let boolToInt = boolValue ? 1 : 0 // 1
```

Here we instruct Swift to convert `boolToInt` which is of type Bool to type Int. It's like saying "If `boolValue` is true, `boolToInt` will be 1. If `boolValue` is false, `boolToInt` will be 0". This is called ternary conditional operator. Its' general formula looks like this:

```
condition ? valueIfTrue : valueIfFalse
```

You'll learn more about this along with conditionals in the lesson about if-else.

# Force unwrapping

Optionals represent values that might be missing. But sometimes, you're certain that an optional actually does contain a value. In those situations, Swift gives you the option to force unwrap the optional — which means telling Swift, "I know there's a value here, go ahead and use it." You can force unwrap an optional value by writing `!` after it, like this:

```
let str = "5"
let num = Int(str)! // force unwrapping
print(num) // 5
```

Here we're confident that `Int(str)` contains 5 which is a valid number, we use `!` to force unwrap the result. This tells Swift to take the value out of the optional and treat it as a regular `Int`.

Swift will immediately unwrap the optional and make num a regular Int rather than an `Int?`. But if you're wrong – if `str` was something that couldn't be converted to an integer – your code will crash.

> If you're wrong and the optional is actually `nil`, your program will crash. You should only use it when you're absolutely sure that the optional contains some value, so be

> careful – there's a reason it's often called the crash operator.

Without `!` Swift would have returned an optional `Int?` because the string might not be convertible to a number.

# Implicitly unwrapped optionals

Implicitly unwrapped optionals are closely related to force unwrapping, but they slightly differ. They are used in different situations.

Implicitly unwrapped optionals are optionals that are declared with a `!` instead of a `?`, meaning Swift treats them like non-optionals once they're initialized.

An implicitly unwrapped optional – written as `String!` – may also contain a value or be `nil`, but they don't need to be checked before they are used. Checking an optional value is called "unwrapping", because we're looking inside the "optional box" to see what it contains. Implicitly unwrapped optionals are not checked before being used:

```
var x: Int! = 10
print(x)

x = nil

let unwrappedX: Int = x // Fatal error
print(unwrappedX)
```

Only use implicitly unwrapped optionals when a variable starts out `nil`, but will definitely have a value by the time you use it, it just can't be assigned immediately. You will learn about this in more detail later on.

> Just like in the case of force unwrapping, if you try to use a value that contains nil your code will crash. You can't catch the error and you can't stop it from happening: your code will crash. Implicitly unwrapped optionals require you to be absolutely sure there's a value there before you use them.

You should avoid both force unwrapping and implicitly unwrapped optionals unless you're certain they are safe – and even then you should think twice.

# Exercises

---

Declare an optional `String` constant named `result` and assign it a `nil` value.

Print the value, if the value is `nil`, print `No result.`

---

Declare one regular `String` constant named `greeting` and one optional `String` called `name`.

Assign `Hello` to the greeting and `nil` to the optional string.

Handle the optional string with a default value `Guest`, if necessary.

Concatenate the two strings using the addition operator and print the result. Use an extra space between the two strings.

---

Write a program that declares one optional `Double` variable called `a` and one optional constant `Double`, named: `b`.

Assign `a` with a `nil` value and `b` with the value of `4.2`.

If a value is `nil`, default it to `0.0`.

Calculate the sum of the two values. And print the result.

Assign the value `6.9` to the `a` variable.

Print the difference of the two variables.

---

Declare one optional `Int` constant named `a` and one non-optional `String` constant, named `b`.

Assign the value 69 to a.
Store a valid number, `420` in the `String`.

Use zero as a fallback value if needed.

Print the difference between the two numbers. (optional minus non-optional).

---

Declare an optional `userName` variable using a `String` type, assign the value `Kitti` to it.

Print the variable using the nil coalescing operator by using `Guest` as a fallback value.

Set the name variable to `nil` and print it again using the nil coalescing operator, just like before.

---

Declare the following constants:

- A `Bool` set to `true`, named `isCompleted`.
- A `Double` set to `3.14`, named `pi`.
- A `String` set to 42 named `meaningOfLife`.
- An `Int` set to `10`, named `max`.

Perform the following type conversions and print the values:

- Convert the `Bool` to an `String`.
- Convert the `Double` to an `Int`.
- Convert the `String` to an `Int` (use a fallback value of `0` if conversion fails).
- Convert the `Int` to a `Double`.

# The readLine function

In earlier examples, we introduced optionals — Swift's way of handling values that might be missing. That concept becomes especially important when your programs start being more interactive.

The `readLine()` function is your first tool for collecting input from the user. It returns an optional string, because nobody can guarantee what the user types — if they type anything at all. This section shows how to ask for input, work with it safely, and build programs that respond to users in real time.

## User input

The `readLine()` function in Swift reads a line of input from the user as a string. Let's see a simple example:

```
print("Please enter something:")
let rawInput: String? = readLine()

let input = rawInput ?? ""
print("User input: `\(input)`")
```

This code snippet starts by displaying a message asking the user to enter something. Then `readline()` pauses the program and waits for the user to type something. It reads the user input and stores it as an optional string. That is `rawInput`.

The value of line is then unwrapped using the nil-coalescing operator `??`, assigning what the user entered to `input`, or an empty string if the user hasn't typed in anything so the line is `nil`.

Finally, it prints the user input, safely handling cases where no input was provided, using an empty string as a fallback value.

> We used the `let` keyword to create constant values when capturing user input. You could technically use variables, but you'd only do that if you planned to change or overwrite the value later in your code — and in this case, we don't.

The value from `readLine()` is captured once, at the moment the user types something and presses Return. After that, we don't plan to change it — the program just needs to store what the user typed. Since the value doesn't need to change later, a constant `let` is a safe choice.

# Reading multiple lines

You've seen how the `readLine()` function lets you read a single line of text. But what if your program needs more than just one answer? For example, what if you want to ask for the user's name and their favorite color? Or have them enter two separate numbers?

In Swift, you can read multiple lines by simply calling `readLine()` more than once — each call will pause and wait for the user to type something and press Return. Here's a simple example:

```
print("Please enter two lines:")
let rawInput1: String? = readLine()
let rawInput2: String? = readLine()

print("User input:")
print(rawInput1 ?? "")
print(rawInput2 ?? "")
```

The program first prints a prompt: "Please enter two lines:". Then it calls readLine() twice. The first call waits for the user to type the first line, and stores it in `rawInput1`. The second call waits for the second line, and stores it in `rawInput2`.

Since `readLine()` always returns an optional `String?`, we use the nil coalescing operator `??` to provide a fallback value — an empty string in case the input is `nil`.

Then the program prints "User input:" and prints what the user entered without modification.

# Reading data types

The `readLine()` function works well with `String` types. But what about integer, double or bool values? That's a bit tricky and you have to use type conversion and sometimes the nil coalescing operator to get the desired type and value. Don't worry, you already know these from the previous chapters.

Now you can put your knowledge into practice, here's how:

```
print("Please enter a number:")

let rawInput: String? = readLine()
let input = rawInput ?? ""

let number = Int(input) ?? 0
print("The number: `\(number)`")
```

If the users enters an invalid number the program will use 0 as a default value, since the Int conversion would result in a `nil` value.

In Swift, unlike numbers like `Int` or `Double`, there's no built-in way to directly convert a `String` into a `Bool` for arbitrary values like "yes" or "no".

> However, `Bool("true")` works and returns `true` only if the string is exactly "true" (case-sensitive); otherwise, it returns `nil` for every other string value.

For custom values, you typically use a comparison to decide whether something is `true` or `false`. Here's how to do that:

```
print("Please enter yes or no.")

let input: String = readLine() ?? ""

let value: Bool = input == "yes"
print("The bool value: \(value)")
```

The result of the comparison is stored as a Bool in the value constant.
We compare the input to the string "yes". If the user typed "yes", the comparison is true.

If they typed anything else (like "no", "YES", "y", or even something random), the comparison is false, because this is the only scenario that value `Bool` returns true:

```
let value = "yes" == "yes"
```

Swift does not try to guess what should be considered true or false. It leaves this decision up to you, the programmer. This way, you have full control over what input is accepted and how you want to handle it.

Now that you know how to read user input and convert between standard data types, you are ready to learn about more complex data types in the next chapter.

# Exercises

---

Write a program that asks the user to enter their favorite quote, using the following text: `Please enter your favorite quote:`

Print the quote exactly as it was entered.

---

Write a program that prompts the user to enter their name and then greets them using their input.

Print `Please enter your name:` to before asking for the user input.

Define a greeting message as a String constant: `Hello, `.

Read the user's name from the input. Use the nil coalescing operator to provide a default value `Guest` if the input is nil.

Concatenate the greeting string with the user input and print the result.

Try pressing CTRL+D for entering no input.

---

Create a program that asks the user to input their age. Convert the input to an integer, defaulting to –1 if the input is invalid.

Display a message: `You are {age} years old.`

---

Write a program that prompts the user to enter a name and a price of an item. One after another.

Convert the price input to a Double, defaulting to `0.0` if the input is invalid. Print the `name` and `price` entered in a full sentence.

---

Create a program that asks the user a yes/no question: `Do you want to subscribe to our newsletter?`

Store the user input in a constant called `rawInput`. If the user enters nothing, use an empty string as a default.
Create a constant called `status` that converts `rawInput` to true if the user enters `yes`, `true`, or `1` and to false otherwise.
Print the subscription status in a full sentence.

---

Write a program that prompts the user to enter two favorite colors, one after another. Use the following prompt message: `Please enter your favorite two colors, each in a new line:`

Display the colors in a single message: `Your favorite colors are {color1} and {color2}.`

# Assessments

Write a program that prompts the user for two integers and then prints their sum. Consider a default value.

---

Write a program that asks the user for the length and width of a rectangle using readLine(). Convert the inputs to Double, defaulting to 0.0 if the input is invalid or nil. Then, calculate and print the area of the rectangle.

---

Write a program that performs the four basic arithmetic operations (addition, subtraction, multiplication, and division) on two numbers provided by the user.

Display the results of addition, subtraction, multiplication, and division to the user by printing each outcome to the console.

Use `readLine()` to read the inputs, convert them to `Double`, and handle invalid or `nil` inputs by defaulting to `0.0`.

---

Write a program that prompts the user to enter their profile information:

- Name (as a String)
- Age (as an Int; if the input is invalid or empty, default to –1)
- Email address (as a String; if the input is empty, default to `Not provided`)
- Subscribed to Newsletter (as a Bool; if the input is invalid, default to `false`)

Use optionals, the nil coalescing operator, and appropriate type conversions to handle each input.
After collecting the information, print the user's details in a clear format.

---

Create an optional price variable using a `Double` type.

Specify a discount constant, using a `Double` type, which is going to represent the percent of the discount, set it to `4.5`.

Set the value of the price to `499.99`.

Print the price, using the following format: `Original price: {price}`.

Always use a fallback value of zero for the price if its value is `nil`.

Apply the 4.5% discount to the price and calculate both the discount price and the savings.

Print the savings like this: `Savings: {savings}`
Print the discount price like this: `Discount price: {discount price}`