# Nullable Types

After this lesson, you'll be able to recognize and use nullable types, and know how to work with nullables effectively by using Kotlin's special operators to safely access potentially null data.

WE'LL COVER THE FOLLOWING

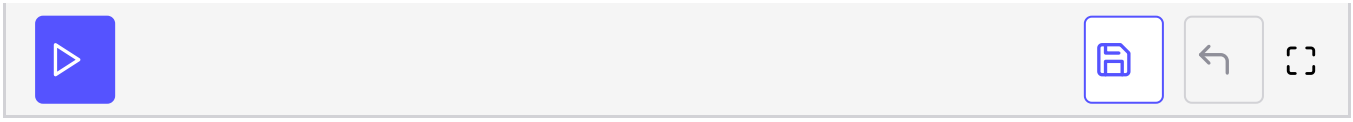


Kotlin's type system differentiates between nullable and non-nullable types. By default, types are non-nullable; you cannot get a null pointer exception from dereferencing it.

This is why Kotlin's basic data types, such as `Int`, can map safely to Java's primitive types, such as `int`, in the bytecode. Both can never be `null`.

Thus, all objects and types you've seen so far in this course were non-nullable. Trying to assign `null` to them would cause a compile-time error:

```
// Compile-time error: cannot assign null to non-nullable type
val input: String = null
```



## Declaring Nullable Types

## Declaring Nullable Types #

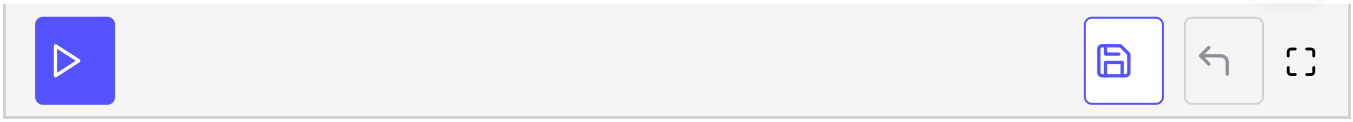
Let's see how you can use nullable types when necessary.

The nullable counterpart of some type `A` is denoted as `A?`. For example, `String?` denotes a nullable string and `Int?` denotes a nullable integer.

> If you're familiar with union types (e.g. from TypeScript), think of these as `String? = String | null`. In other words, a `String?` is either a `String` or `null`.

Assigning `null` to a nullable type works as you'd expect:

```
val input: String? = null
```

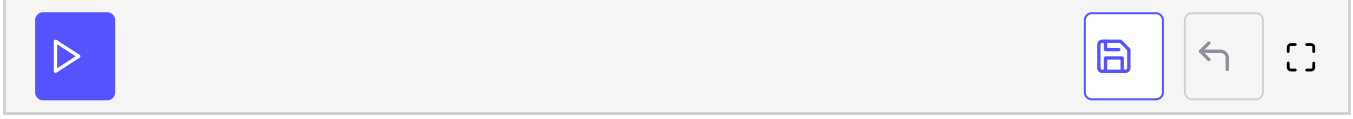


## Using Nullable Objects #

Kotlin is focused on safety. Therefore, you cannot simply access any members on a nullable object as this might cause a null pointer exception:

```
val input: String? = null

val output = input.toUpperCase()  // Compile-time error: unsafe call on nullable object
```



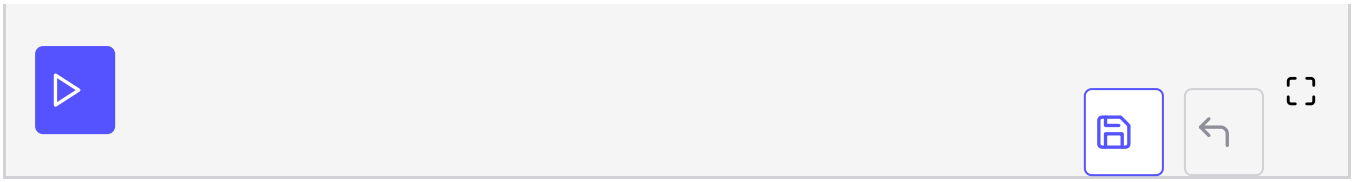
Here, the compiler complains because `input.toUpperCase()` will fail at runtime if `input` is `null`, which would indeed be the case here.

### Safe Call Operator #

To access members of a nullable object in Kotlin, you use the *safe call operator* by appending a `?` to the object:

```
val input: String? = null

val output = input?.toUpperCase()  // Works and returns null
```



The safe call operator works as follows:

- If the object is `null`, it evaluates to `null`.
- Otherwise, it dereferences the objects and evaluates to the value of the overall expression.

> **Exercise:** Try assigning an actual string to the `input` variable above and run the code again.*

In nested structures of nullable objects, you may chain the safe call operator as in `person?.spouse?.name`.

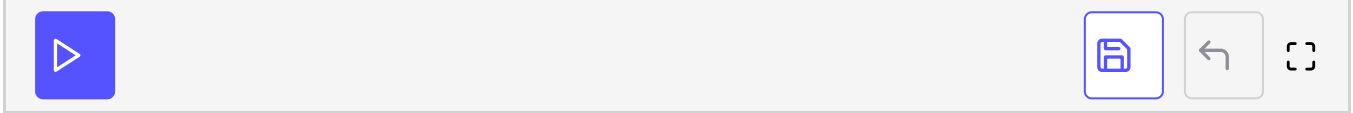
## Elvis Operator #

When working with nullable data, you often want to define default values that should be used in case an object is `null`.

In Kotlin, this is done using `?:`, the so-called *Elvis operator*:

```
val name: String? = null

val chatName = name ?: "Anonymous"  // Elvis operator
val displayName = chatName.toUpperCase()
```



**You can read the Elvis operator as "or else".** Here, `chatName` is equal to `name` *or else* `"Anonymous"`.

This operator is useful to get rid of nullability in your code. It's good practice to replace `null` values with useful default values as early as possible, as this will simplify calling code.

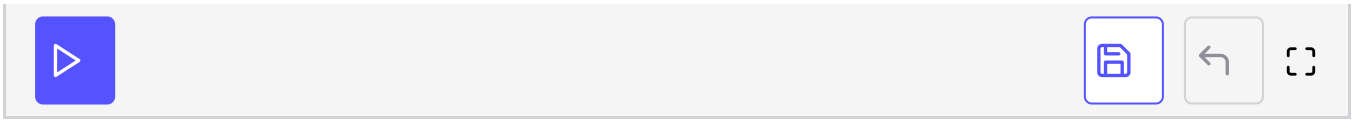
> **Fun fact:** The name "Elvis operator" stems from the fact that it looks like

an elvis emoji `?:-O`

It's worth mentioning that this operator is not just for setting default values. *Anything you put on its right-hand side is executed if the accessed value is* `null`, so another typical case is throwing an exception:

```
val input: String? = null

val userInput = input ?: throw IllegalArgumentException("Input must not be null.")
```



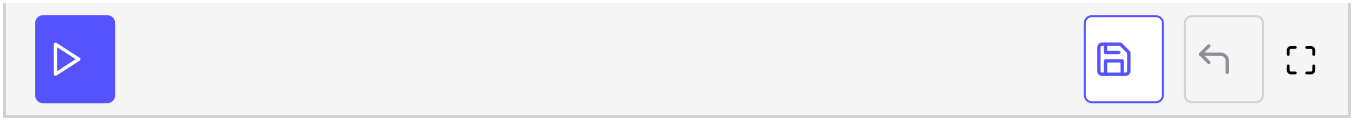
## Unsafe Call Operator #

The last way to access a member on a nullable object (and literally the last one you should typically use) is the *unsafe call operator* `!!`.

This operator represents a *non-null assertion*. With it, you're telling the compiler that the object you're trying to access cannot be `null` at this particular place in your code. Therefore, Kotlin will access the value without further safety checks. So if you're wrong, you get a null pointer exception.

```
val input: String? = null

val output = input!!.toUpperCase()
```



Using this operator is one of the few ways to get a null pointer exception in Kotlin. *Think of it as an assertion and only use it if necessary*. Usually, you can avoid this by refactoring your code to avoid nullability in the first place or by using the safe operators, as explained above.

# Effective Nullability Handling #

Explicit nullability, is in my opinion, the primary differentiating feature of Kotlin's type system when compared to Java.

Apart from the obvious benefits regarding safety and clarity, it does have a learning curve. A typical beginner's mistake is to use nullables and `!!`

throughout the code.

Of course, it takes more effort to always think about the “null case”, but Kotlin facilitates this with:

- Operators such as `?` and `?:`
- Various standard library functions such as `String?.isNullOrEmpty()`

## Quiz

Nullable types and null handling in Kotlin.

1

Given a nullable `File?` variable called `configFile`, which of the following is a safe way to get the file path?

COMPLETED 0%

1 of 3

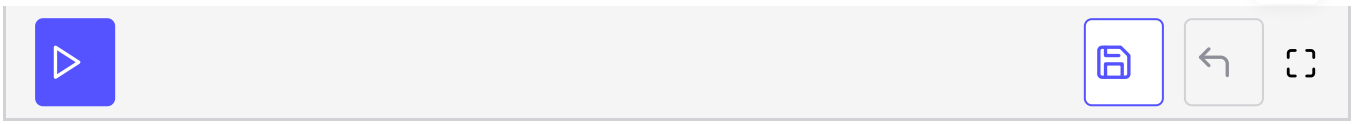
## Exercise

In the following, you’re given variables `x`, `y`, `z` of type `Double?` (in hidden code that is prepended to your code). Calculate and print the sum `x + y + z`. If a number is `null`, it should be ignored for the sum.

**Hint:** you can print to the console using `println`.

Problem | Solution

```
// Compute sum of x, y, z. Each one may be null (randomized).
```



## Summary #

Here are the key takeaways from this lesson:

- Kotlin's type system uses explicit nullable types.
  - By default, types are non-nullable (e.g. `Person`).
  - Each type has a nullable counterpart (e.g. `Person?`).
- The safe call operator `?` is used to safely access members of nullable types.
- The Elvis operator `?:` is used to handle null values conveniently (e.g. to define useful default values).
- The unsafe call operator `!!` acts like a non-null assertion and should be used judiciously.

---

Congratulations! You're now familiar with the basics of Kotlin's type system. In the following section, you'll learn how to implement conditional control flow using `if` and `when`.