Explicit Type Casting

We'll cover the following

When to use explicit casting

Use explicit casts only in situations where smart casts aren't possible—that is, when the compiler can't determine the type with confidence. For example, if a `var` variable were to change between the check and its use, then Kotlin can't guarantee the type. In such cases it won't apply smart casts, and we have to take the responsibility for casts.

Kotlin provides two operators to perform explicit cast: `as` and `as?`. Let's create an example that illustrates how to use both of them.

Using `as`

Suppose we have a function that returns a message of different types, like this one:

```kotlin
// unsafecast.kts
fun fetchMessage(id: Int): Any =
  if (id == 1) "Record found" else StringBuilder("data not found")
```

Now suppose that we want to receive the message by calling the above `function`, and print some details about it, but only if the result is a `String` type. One approach to writing such code would be to store the result of the call to `fetchMessage()` in a temporary variable. Then we can use is to check the type, and if it is of `String` type, then we can use smart casts to get the details we need from that temporary variable.

That will work. It's defensive coding, and so it's a prudent solution. But at times, we all are tempted to reduce the number of lines of code to an unhealthy extent and in

the process may introduce unintended behavior in code. If we give in to that urge here, we could be tempted to use `as` to write code like this:

```
// unsafecast.kts
for (id in 1..2) {
  println("Message length: ${(fetchMessage(id) as String).length}")
}
```

Casting using `as` is like putting all your money into a lottery—the outcome won't be pleasant. If the type of the object is different from what's expected, the result is a runtime exception. No fun.

```
Message length: 12
java.lang.ClassCastException: java.base/java.lang.StringBuilder cannot be cast
        to java.base/java.lang.String
```

To avoid this possibility we may fall back and use the `is` operator, but the safe alternative `as?` works in this particular example.

Using `as?`

The `as` operator results in a reference of the same type as the one specified to the right of the operator, as in this example:

```
val message: String = fetchMessage(1) as String
```

On the other hand, `as?` results in a nullable reference type, like in this one:

```
val message: String? = fetchMessage(1) as? String
```

Whereas the `as` operator blows up if the casting fails, the safe cast operator `as?` will assign `null` to the reference upon failure.

Instead of using the `as` operator, let's switch over to using the safe alternative `as?`. Here's the one line of code that has changed from the previous example:

```
println("Message length: ${(fetchMessage(id) as? String)?.length ?: "---"}")
```

Since the safe cast operator assigns the reference to null if the cast fails, we use the

Elvis operator to provide an alternative to length when necessary.

This is the output after the change:

```
Message length: 12
Message length: ---
```

The safe cast operator `as?` is better than the unsafe `as`. Here are some recommendations to take to the office:

- Use smart casts as much as possible.
- Use safe cast only when smart cast isn't an option.
- Use unsafe cast if you want to see the application crash and burn.

Kotlin's support for making your code type safe doesn't end with simple types. The language walks a few extra miles to make code that uses generics type safe as well. We often use generic types in code, and learning about the flexibility that Kotlin offers with generics' parametric types will help us not only create better code but also, more importantly, understand code that uses these features.

Dealing with generics in general isn't easy, and when we mix in terms like covariance and contravariance, it can get frustrating. The following are some advanced capabilities of the language that are worth the effort to learn, but take it slow for the concepts to sink in. Take a short walk first, refill your caffeinated beverage, and when you come back, get ready to practice the code as you read along—that will help you absorb this advanced topic more easily.

The next lesson explains how to reuse code without worrying about type safety.