Built-in Standard Delegates

We'll cover the following

Kotlin provides a few built-in delegates that we can readily benefit from. The Lazy delegate is useful to defer creating objects or executing computations until the time the result is truly needed. The `observable` delegate is useful to observe or monitor changes to the value of a property. The `vetoable` delegate can be used to reject changes to properties based on some rules or business logic. We’ll explore these features in this section.

It’s OK to get a little lazy

Decades ago John McCarthy introduced *short-circuit* evaluation to eliminate redundant computations in Boolean logic—the execution of an expression is skipped if the evaluation of an expression ahead of it is enough to yield the result. Most programming languages support this feature, and programmers quickly learn about the efficiency of this approach. The Lazy delegate pushes the boundaries of this approach—let’s see how. Suppose we have a function that gets the current temperature for a city (the following implementation merely returns a fake response):

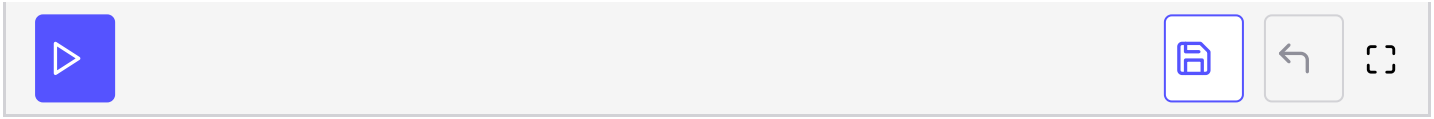
```
fun getTemperature(city: String): Double {
  println("fetch from webservice for $city")
  return 30.0
}
```

shortcircuit.kts

Calling this hypothetical function `getTemperature()` is going to consume some time due to the remote access it requires to a web service. Also, there may be cost associated with the service usage. It’s better to avoid calls where possible. The

short-circuit evaluation naturally does that, like in this example:

```
val showTemperature = false
val city = "Boulder"

if (showTemperature && getTemperature(city) > 20) //(nothing here)
  println("Warm")
else
  println("Nothing to report") //Nothing to report
```



shortcircuit.kts

Since the value of `showTemperature` variable is `false`, due to short-circuit evaluation the execution of `getTemperature()` method will be skipped. That's efficient—if the result of a task isn't used, we don't bother doing that work.

However, a slight refactoring of this code will result in loss of that efficiency.

```
// eagerevaluation.kts
val temperature = getTemperature(city) //fetch from webservice

if (showTemperature && temperature > 20)
  println("Warm")
else
  println("Nothing to report") //Nothing to report
```

We stored the result of `getTemperature()` into a local temporary variable and then used that within the Boolean expression of the `if` expression. But we incur the execution overhead with this change, even though we're not using the value of the `temperature` variable, again due to short-circuit evaluation. How sad.

Why can't Kotlin be smart to avoid this call? you may wonder. Because a function or method call may have side effects, its execution isn't skipped, in general. In the context of short-circuit, though, the execution may be skipped. The reason, simply put, is a well-known behavior of programs—programmers are familiar with short-circuit evaluation in Boolean expressions, and language specifications convey that expressions within Boolean expressions are not guaranteed to execute.

But, you protest, there must be a way to skip execution of expressions outside the context of Boolean expressions. Kotlin heard you—you can tell the compiler to be lazy about executing an expression until the point where its result is truly needed;

otherwise, skip the execution entirely.

Let's modify the previous code to use the `Lazy` delegate, but instead of directly using the `Lazy` delegate class, we'll use the `lazy` convenience wrapper function.

```
// lazyevaluation.kts
val temperature by lazy { getTemperature(city) }

if (showTemperature && temperature > 20) //(nothing here)
  println("Warm")
else
  println("Nothing to report") //Nothing to report
```

We turned the simple variable `temperature` into a delegate property using the `by` keyword. The `lazy` function takes as argument a lambda expression that will perform the computation, but only on demand and not eagerly or immediately. The computation within the lambda expression will be evaluated if and when the value of the variable is requested. It's deferred until that time and, potentially, the execution may never happen, as in this example.

If we change the value of `showTemperature` to true, then the execution of `getTemperature()` will take place—not where `temperature` is defined, but where the comparison to > 20 happens, which is after the evaluation of the expression `showTemperature` in the Boolean condition.

Once the expression within the lambda is evaluated, the delegate will memoize the result and future requests for the value will receive the saved value. The lambda expression is not reevaluated.

The `lazy` function by default synchronizes the execution of the lambda expression so that at most one thread will execute it. If it's safe to execute the code concurrently from multiple threads or if you know the code will be executed only on a single thread, like in the case of Android UI application code, then you may provide an argument of the `enum` type `LazyThreadSafetyMode` to the `lazy` function to specify different synchronization options.

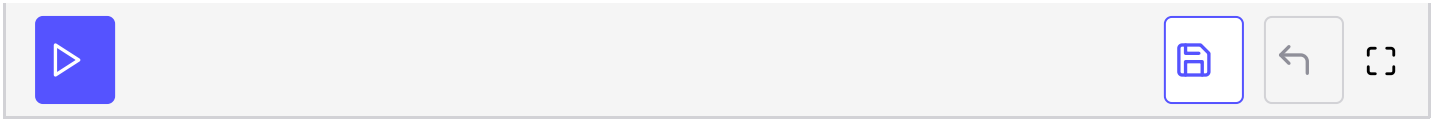
It's OK to get a little lazy where that can lead to more efficiency. Next, we'll see how to observe change to variables' values.

The observable delegate

The singleton object `kotlin.properties.Delegates` has an `observable()` convenience function to create a `ReadWriteProperty` delegate that will intercept any change to the variable or property it's associated with. When a change occurs, the delegate will call an event handler you register with the `observable()` function.

The event handler receives three parameters of type `KProperty` which hold the metadata about the property, the old value, and the new value. It doesn't return anything—that is, it's a `Unit` or void function. Let's create a variable with the `observable` delegate.

```
import kotlin.properties.Delegates.observable

var count by observable(0) { property, oldValue, newValue ->
  println("Property: $property old: $oldValue: new: $newValue")
}

println("The value of count is: $count")

count = count + 1 // Need to be changed later

println("The value of count is: $count")

count = count - 1 // Need to be changed later

println("The value of count is: $count")
```



observe.kts

The variable `count` is initialized with the value `0` that is provided as the first argument to the `observable()` function. The second argument to `observable()` is a lambda expression, the event handler. It merely prints the details of the property: the value of the variable before the change and the new value given for the variable. The increment operation on `count` will result in a call to that event handler. Here's the output of the code:

```
The value of count is: 0
Property: var Observe.count: kotlin.Int old: 0: new: 1
The value of count is: 1
Property: var Observe.count: kotlin.Int old: 1: new: 0
The value of count is: 0
```

Use the `observable` delegate to keep an eye on changes to local variables or properties within objects. It can be useful for monitoring and debugging purposes.

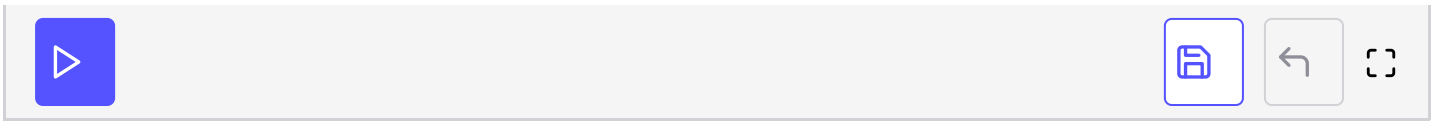
Instead of merely observing, if you want to partake in deciding if a change should be accepted or rejected, then use the `vetoable` delegate.

Exercise your vetoable rights

Unlike the handler registered with `observable`, whose return type is `Unit`, the handler we register with `vetoable` returns a `Boolean` result. A return value of `true` means a favorable nod to accept the change; `false` means reject. The change is discarded if we reject.

Let's modify the previous code to use `vetoable()` instead of `observable()`.

```
import kotlin.properties.Delegates.vetoable

var count by vetoable(0) { _, oldValue, newValue -> newValue > oldValue }

println("The value of count is: $count")

count = count + 1 // Changed later

println("The value of count is: $count")

count = count - 1 // Changed later

println("The value of count is: $count")
```



veto.kts

In this version of code we're not using the `property` parameter—to avoid Kotlin compiler's warning for unused variables we use `_` in that position. The lambda expression returns `true` if the new value for the variable is more than the previous value; otherwise it returns `false`. In other words, only increasing changes to the value of `count` are accepted. Let's take a look at the output of the code:

```
The value of count is: 0
The value of count is: 1
The value of count is: 1
```

Use the `vetoable` delegate anywhere you like to keep an eye on a local variable or property and want to reject certain changes based on some business logic.

QUIZ

1 What is the function of the observable delegate?

2 What is the function of vetoable delegates?

Retake Quiz

The next lesson concludes the discussion for this chapter.