Memoization

We'll cover the following

I invite you to work with me on a small math problem. Grab a paper and pencil. All set? OK, here we go. What's `321 + 174`? Without a calculator on hand it's not something I can answer spontaneously, and if you need a moment to find the answer, that's perfectly normal.

Did you find it? If you wrote down `495` you got it without any errors. Good job.

Now, one more question. What's `321 + 174`? Now, you answered that in a snap, didn't you? How did you get so fast so quickly?

It's ok to admit, you looked up the result from the previous computation. That's pretty smart; you wouldn't redundantly compute the expression that you evaluated only a moment ago. That's memoization when done in code—we don't want our programs to recompute functions that were executed already for the same input, provided the output will be the same no matter how many times we call for the same input. By using saved values we can avoid recomputation and make executions faster. A caveat, though—memoization may only be used for pure functions, which are functions with no side effects.

In dynamic programming, an algorithmic technique, the problem is solved recursively using solutions to subproblems, but in a way that the results of the subproblem are memoized and reused. This technique brings the computational complexity of problems from exponential to linear and results in huge performance gains while keeping the code expressive.

Kotlin doesn't directly support memoization, but we can build it with the tools you've learned so far. We'll implement memoization in two different ways in

Kotlin: one like the solution provided in the Groovy language's library and one using Kotlin delegates.

Repetitive computation

We'll consider two problems in this chapter to illustrate memoization, a simple one first and then something a bit more complicated.

The first problem, the Fibonacci number, has been beaten to death in programming examples. One reason, and why we'll use it here, is the problem is very simple, well understood, and we can keep our eyes on the solution without getting dragged into the complexity of a problem. Once we learn how to use memoization with this simple problem, we'll apply it to the rod-cutting problem, but we'll discuss that later.

Let's implement a simple recursion in Kotlin to find the Fibonacci number.

```
import kotlin.system.measureTimeMillis

fun fib(n: Int) : Long = when (n) {
  0, 1 -> 1L
  else -> fib(n - 1) + fib(n - 2)
}

println(measureTimeMillis { fib(40) }) //About 3 millisconds
println(measureTimeMillis { fib(45) }) //More than 4 seconds
```

fib.kts

The `fib()` function returns `1` if the value provided is less than `2`. Otherwise, it computes the result by making two recursive calls.

Call to `fib(4)` will invoke `fib(3)` and `fib(2)`. But, when `fib(3)` is evaluated, it again calls `fib(2)`. For small values of `n` the code will run quickly. As `n` increases, the computation time will increase exponentially. For instance, for `n` equal to `40` the code took about 3 milliseconds on my system. For `45` that jumped to well over 4 seconds. I don't dare to run it for `n` equals `100`.

We can reduce the computation time significantly by memoizing the result of the call before returning it. Then on subsequent calls, if the value is already present, we can return it and save a slew of recursive calls.

Memoization, the Groovy way in Kotlin

We have a few options to store the values of computations. We could create a class

and put the cache of data in a field or property. Then the functions of the class can make use of the cache. That's a reasonable approach, but that would force us down the path of creating classes, which is fine if we already are heading in that direction. If we're dealing with standalone functions, we shouldn't be forced to create classes.

We need some dynamic behavior in code to handle memoization. When a function is called, we need to check the cache to see if the data exists and call the function only if it doesn't. This isn't possible with standalone functions, since the function call will bind to the function, and it's hard to replace that call dynamically to implement the conditional logic we discussed. We can work around this limitation by using lambda expressions—we'll soon see how we replace the call to lambda to include this check.

In the Groovy language, memoization is implemented as part of the library. You can call a `memoize()` function on any lambda expression and it will return a memoized lambda. We'll follow a similar approach here in Kotlin.

In Chapter 13, Fluency in Kotlin, you learned to inject methods into classes and functions. That technique comes in handy here to create a `memoize()` method on a lambda expression.

```
fun <T, R> ((T) -> R).memoize(): ((T) -> R) {
  val original = this
  val cache = mutableMapOf<T, R>()

  return { n: T -> cache.getOrPut(n) { original(n) } }
}
```

fibmemoize.kts

In the first line we inject the `memoize()` method into a generic lambda expression that takes a parameter of the parametric type `T` and returns a result of the parametric type `R`. The `memoize()` function's return type is a lambda expression of the same type as the type of the method into which `memoize()` is injected. In other words, the result of calling `memoize()` on a function is a function with the same signature as the function.

Within the `memoize()` function, we save the reference to the original function by assigning `this` to a local variable `original`. Then we initialize an empty `cache`. Finally, we return a lambda expression that takes a parameter of type `T` and returns a result of type `R`. Within this returned function, we look up the cache, to

see if the result already exists. If it doesn't exist, we compute the result, and store it

before returning. If the result already exists, we skip the computation and return the stored value.

Let's use the `memoize()` function to memoize the computation of the Fibonacci number.

```
lateinit var fib: (Int) -> Long

fib = { n: Int ->
  when (n) {
    0, 1 -> 1L
    else -> fib(n - 1) + fib(n - 2)
  }
}.memoize()
```

fibmemoize.kts

We wrote the function as a lambda expression instead of a regular function. Since we're calling `fib()` within the lambda expression, we can't declare the variable fib in the same expression where we create the lambda expression. By using `lateinit` we're telling Kotlin that we've not forgotten to initialize the `fib` variable, and we plan to get to that momentarily. Without `lateinit`, the Kotlin compiler will give an error for unassigned variable. The `fib()` function returns `1` if the value provided is less than `2`. Otherwise, it computes the result by making two recursive calls.

The `memoize()` function took the given lambda expression, stored it into the `original` variable inside of the `memoize()` function, and returned a new lambda expression. That's what's stored in the fib variable now.

Let's use this version of code to exercise the memoized version.

```
println(measureTimeMillis { fib(40) })
println(measureTimeMillis { fib(45) })
println(measureTimeMillis { fib(500) })
```

fibmemoize.kts

Let's be brave—call the function for a large value of `500`. Fire away to see how this performs:

```
0
0

1
```

Those are time in milliseconds. The computation, that took well over `4` seconds for the non-memoized version, took no observable time. The execution for `500` as input took about `1` millisecond. Your output might be different from the above.

The `memoize()` function we wrote here may be used for any function that takes one parameter. The solution has pros and cons. What's nice is we can call `memoize()` on a lambda expression and turn it into memoized version. Also, the memoization code is concise. However, we have to first define `fib` and then assign the lambda expression to it. If we define the variable on the same expression as the one defining the lambda, we'll get a compilation error; Kotlin complains that `fib` isn't defined at the time of use.

This solution worked and it's a reasonable solution to keep, but Kotlin also has delegates. If you're curious how this solution would change if we use delegates, the answer is waiting for you next.

Memoization as delegate

In the previous section, the code `fib = {...}.memoize()` replaced the variable `fib` with the memoized lambda expression. But you saw Kotlin's facility to intercept access to properties and local variables in Delegating Variables and Properties. We can use that approach by writing a delegate. Instead of `var`, we'll use `val` for `fib`; we're going to assign to it only once. Thus, we need only a `getValue()` method in the delegate—no need for `setValue()`. Let's create the delegate.

```
import kotlin.reflect.*

class Memoize<T, R>(val func: (T) -> R) {
  val cache = mutableMapOf<T, R>()

  operator fun getValue(thisRef: Any?, property: KProperty<*>) = { n: T ->
    cache.getOrPut(n) { func(n) } }
}
```

fibdelegate.kts

The delegate holds the cache internally, keeps the original function as a property `func`, and the `getValue()` function returns a lambda expression that dispatches the call to the original function only if the value isn't in the cache.

Conceptually this solution and the previous one are similar, but this solution will be applied very differently to the `fib` function than the previous solution. Let's apply the delegate when creating the `fib` function:

```
val fib: (Int) -> Long by Memoize {n: Int ->
  when (n) {
    0, 1 -> 1L
    else -> fib(n - 1) + fib(n - 2)
  }
}
```

fibdelegate.kts

The application of memoization is a lot cleaner with delegate. Even though we're still in the middle of initializing `fib`, the Kotlin compiler doesn't complain about accessing `fib` within the lambda here. The reason is that, internally, we're using the delegate to access the variable `fib` and not directly accessing it.

Let's exercise this version of `fib()` to confirm the performance is as good as the previous version.

```
println(measureTimeMillis { fib(40) })
println(measureTimeMillis { fib(45) })
println(measureTimeMillis { fib(500) })
```

fibdelegate.kts

Here's the time the code took, in milliseconds, for each call to `fib()`:

```
0
0
1
```

We've seen two different ways to create memoization, and both the implementations are similar; but the delegate solution is more elegant than the function call solution, due to less effort to apply memoization.

QUIZ

Q How does memoization work?

Retake Quiz

In the next lesson, let's use the delegate solution further to memoize another problem that will benefit from it.