When It's Time to Use when

We'll cover the following

- when as an expression
- when as a statement
- when and variable scope

Kotlin doesn't have `switch`; instead it has `when`, and that comes in different flavors: as expression and as statement.

`when` as an expression

Here's an implementation of the function to determine if a cell will be alive in the next generation in Conway's Game of Life.

```
fun isAlive(alive: Boolean, numberOfLiveNeighbors: Int): Boolean {
    if (numberOfLiveNeighbors < 2) { return false }
    if (numberOfLiveNeighbors > 3) { return false }
    if (numberOfLiveNeighbors == 3) { return true }
    return alive && numberOfLiveNeighbors == 2
}
```

boring.kts

This code tells if a cell will live in the next generation, but any programmer reading such code may quickly lose motivation to live; the code is too noisy, verbose, repetitive, and error prone.

In the simplest form, when can replace a series of `if-else` statements or expressions. Now is a great time to refactor that code to use Kotlin's when.

```
fun isAlive(alive: Boolean, numberOfLiveNeighbors: Int) = when {
    numberOfLiveNeighbors < 2 -> false
    numberOfLiveNeighbors > 3 -> false
    numberOfLiveNeighbors == 3 -> true
    else -> alive && numberOfLiveNeighbors == 2
}
```

when.kts

The previous version of the function specified the return type and used the block structure for the method body. In this version, the refactored code uses type inference and the single-expression function syntax. Here, `when` is used as an expression. The value returned by the function is the value returned by one of the branches in `when`.

The two versions of the `isAlive()` function produce the same results for the same inputs, but the one using `when` is less noisy in comparison. `when` is succinct when compared to `if` in general, but as an observant reader you may further refactor the above code to replace the entire `when` expression with a simple `alive` && `numberOfLiveNeighbors == 2 || numberOfLiveNeighbors == 3` after the `=` separator.

In the case where `when` is used as an expression, the Kotlin compiler will verify that either the else part exists or that the expression will result in a value for all possible input values. This compile-time check has a direct impact on the accuracy of code and reduces errors that often arise from conditions that were accidentally overlooked.

In the previous example, the `when` expression isn't taking any arguments, but we may pass a value or an expression to it. Let's look at an example of that by using it in a function that takes a parameter of Any type—see more about this in Any, the Base Class.

We'll use `Any` in the next example to illustrate the versatility of `when`. Occasionally, you may find it useful to receive `Any`—for example, in an application tier where a message broker may receive messages of different types. Consider those cases to be an exception rather than a norm, and, in general, avoid defining methods that take `Any` as a parameter. With those words of caution, let's take a look at the code.

```
fun whatToDo(dayOfWeek: Any) =  when (dayOfWeek) {
  "Saturday", "Sunday" -> "Relax"
  in listOf("Monday", "Tuesday", "Wednesday", "Thursday") -> "Work hard"
  in 2..4 -> "Work hard"
  "Friday" -> "Party"
  is String -> "What?"
  else -> "No clue"
}

println(whatToDo("Sunday")) //Relax
println(whatToDo("Wednesday")) //Work hard
println(whatToDo(3)) //Work hard
println(whatToDo("Friday")) //Party
```

```
println(whatToDo("Munday")) //What?
println(whatToDo(8)) //No clue
```

activity.kts

In this example, we passed to `when` the variable `dayOfWeek`. Unlike the previous example, where each of the conditions within `when` were Boolean expressions, in this example we have a mixture of conditions.

The first line within when checks if the given value is one of the comma separated values—you're not restricted to two values. In the next two lines we check if the given parameter is a member of the list or the range, respectively. In the line with "Friday" we look for an exact match. In addition to matching if the given value is in a list or range, you may also perform type checking—that's what we're doing in the line starting with is `String`. That branch is taken if the given input doesn't match with any of the previous conditions and is of type `String`. Finally, the line with `else` takes care of the default situation. Once again, the compiler will insist on having the `else` part in this case.

A word of caution. The Kotlin compiler won't permit the `else` part to appear anywhere but as the last option within `when`. But it doesn't complain if a more general condition, for example, if is `String` is placed before a more specific condition, like "Friday". The order in which the conditions are placed matters—the execution will follow the path corresponding to the first satisfying condition.

In the examples we've seen so far, the code that follows the `->` was a short single expression. Kotlin permits that part to be a block as well. The last expression within the block becomes the result of the execution of that path.

> Can you also figure out how to check multiple conditions using `when`?

Hide Answer

The `&&` and `||` operators can be used to check multiple conditions:

```
when {
```

```
    n != 17 && n != 42 -> doThis()
    else -> doThis()
}
```

Even though Kotlin allows blocks after `->`, from the readability point of view, avoid that. If more complex logic than a single expression or statement is needed, refactor that into a separate function or method and call it from within the path after `->`. Friends don't let friends write large, ugly when expressions.

`when` as a statement

If you want to perform different actions based on the value of one or more variables, you can use `when` as a statement instead of as an expression. Let's convert the previous code to print the activities instead of returning a `String` response and, along the way, give a new name to the function.

```
fun printWhatToDo(dayOfWeek: Any) {
  when (dayOfWeek) {
    "Saturday", "Sunday" -> println("Relax")
    in listOf("Monday", "Tuesday", "Wednesday", "Thursday") ->
      println("Work hard")
    in 2..4 -> println("Work hard")
    "Friday" -> println("Party")
    is String -> println("What?")
  }
}

printWhatToDo("Sunday") //Relax
printWhatToDo("Wednesday") //Work hard
printWhatToDo(3) //Work hard
printWhatToDo("Friday") //Party
printWhatToDo("Munday") //What?
printWhatToDo(8) //
```

printActivity.kts

The return type of the function `printWhatToDo()` is `Unit`—it returns nothing. Within `when` each of the conditions is taking an action, printing something to the standard out. Kotlin doesn't care if you don't provide the `else` condition in the case where `when` is used as a statement. No action is taken when no condition matches, like when `8` is passed.

`when` and variable scope

In the previous examples the variables used for matching came from outside the `when` expression or statement. But that's not a requirement; the variables used for matching may be limited to the scope of `when`. Designing code with such a restriction, where possible, is a good thing, as it will prevent variable scope bleeding and make the code easier to maintain.

Let's look at limiting a variable's scope with an example. Here's a function that uses when to examine the number of cores on a system.

```
fun systemInfo(): String {
  val numberOfCores = Runtime.getRuntime().availableProcessors()

  return when (numberOfCores) {
    1 -> "1 core, packing this one to the museum"
    in 2..16 -> "You have $numberOfCores cores"
    else -> "$numberOfCores cores!, I want your machine"
  }
}
```

cores.kts

The `systemInfo()` function returns a response based on the number of cores, but the code is a tad noisy. The function had to first invoke the `availableProcessors()` method of `Runtime` to determine the number of cores. Then that variable was passed to `when` for its evaluation. We can reduce the noise and limit the scope of the variable `numberOfCores` to the `when` block by rewriting the code as follows:

```
fun systemInfo(): String =
    when (val numberOfCores = Runtime.getRuntime().availableProcessors()) {
        1 -> "1 core, packing this one to the museum"
        in 2..16 -> "You have $numberOfCores cores"
        else -> "$numberOfCores cores!, I want your machine"
}

println(systemInfo())
```

cores.kts

Placing the `val` in the argument to `when` gives us some benefits. First, we can directly return the result of `when` and remove the outer block `{}` and `return`; that's less noise and fewer lines of code too. Second, in cases where we want to take the result of `when` and do further processing, the variable `numberOfCores` won't be available beyond the point it's needed. Limiting scope for variables is

good design.

QUIZ

1 `when` is equivalant to which command?

2 Which keyword is used to check a list of objects?

3

What will be the output of the following code?

```
age = 20
when (age) {
        in 0..12 -> println("Child")
    in 13..19 -> println("Teenager")
    else -> println("Adult")
}
```

Retake Quiz

The next lesson concludes the discussion for this chapter.