Conditions as Expressions

Learn how to use conditions in Kotlin as expressions, including assigning different values to a variable depending on a condition.

WE'LL COVER THE FOLLOWING

In Kotlin, both `if` and `when` can be used as *expressions* instead of *statements*. An *expression* is a piece of code that has a value, e.g. `"Kotlin"`, `42 * 17`, or `readInput()`. In contrast, a statement is a piece of code with *no* value, such as `fun foo() { ... }` or `while (active) { ... }`. In many programming languages, `if` and `when`/`switch` are statements. But in Kotlin, they are expressions! Let's explore how this works.

Expressions with `if`

Recall this listing from the lesson on `if` statements:

```
if (planet == "Jupiter") {
  println("Radius of Jupiter is 69,911km")
} else if (planet == "Saturn") {
  println("Radius of Saturn is 58,232km")
} else {
  println("No data for planet $planet")
}
```

If you think about it, it seems that the relevant data that changes here based on the condition is the radius. So let's store that in a variable by using an `if` expression:

```
val radiusInKm = if (planet == "Jupiter") {
        // Run code here...
        69911  // Last line defines the return value of this block
} else if (planet == "Saturn") {
        // Run code here...
        58232
} else {
        // Run code here...
        -1
}
```

Here, the entire `if-elseif-else` block has a value. This value for each branch is decided by the ***last line* in each branch**. You may run arbitrary code in each condition block but ultimately, the last line defines the value.

> **Note:** When assigning such an expression to a variable, the conditions must be *exhaustive*. This means that one of the condition blocks must match. The easiest way to achieve this is to have an `else` branch as fallback. Why must it be exhaustive? Because otherwise the `if` expression would not have a well-defined value in all cases, and thus it could not be assigned to a variable such as `radiusInKm`.

Ternary Conditional Operator in Kotlin

Kotlin has no separate language construct for a ternary conditional operator of the form `someCondition ? thenThis : elseThat`, which you may know from other languages. Instead, `if` expressions are used:

```
val category = if (age >= 18) "adult" else "child"
```

Note that you don't need curly braces around the condition blocks if they're a single expression.

Expressions with `when`

Naturally, `when` can be used as an expression in the same way:

```
val radiusInKm = when (planet) {
    "Jupiter" -> 69911
    "Saturn"  -> 58232
    else      -> -1
}
```

> In these simple examples, an `else` block is required for exhaustiveness. However, this is usually avoided in more advanced code by switching on values with a fixed finite set of possible values, such as enums or sealed classes in Kotlin. The compiler then infers whether you have covered all possible values.

Quiz

Conditional expressions with `if` and `when`

1 Which of the following correctly defines expressions vs statements? You can select multiple answers.

Exercises

In the following code listing, you are given a variable `priority`. Initialize a read-only variable `priorityText` and set it to a different value depending on the value of `priority`:

- `priority` is 1 => `priorityText` should be `"Trivial"`
- `priority` is 2 => `priorityText` should be `"Minor"`
- `priority` is 3 => `priorityText` should be `"Normal"`
- `priority` is 4 => `priorityText` should be `"Major"`
- `priority` is 5 => `priorityText` should be `"Critical"`
- Otherwise, it should be `"Unknown"`

Problem | Solution (if) | Solution (when)

```
// Insert your code here
```

Implement the previous exercise using both an `if` expression and a `when` expression. Which one seems better for this use case? Why?

Summary

- *Expressions* have a value. *Statements* do not.
- In Kotlin, `if` and `when` are *expressions*.
  - If you don't use the value, they work exactly like conditional *statements* in other languages.
- Conditional expressions are useful to assign a value directly instead of splitting up declaration and initialization.
  - Consider how you would solve the exercise above in a language like Java or C.

Congrats! You now know how to use conditions idiomatically in Kotlin.

In the following section, you will learn about the different types of collections

available in Kotlin, how to use them, and when to use which of them.