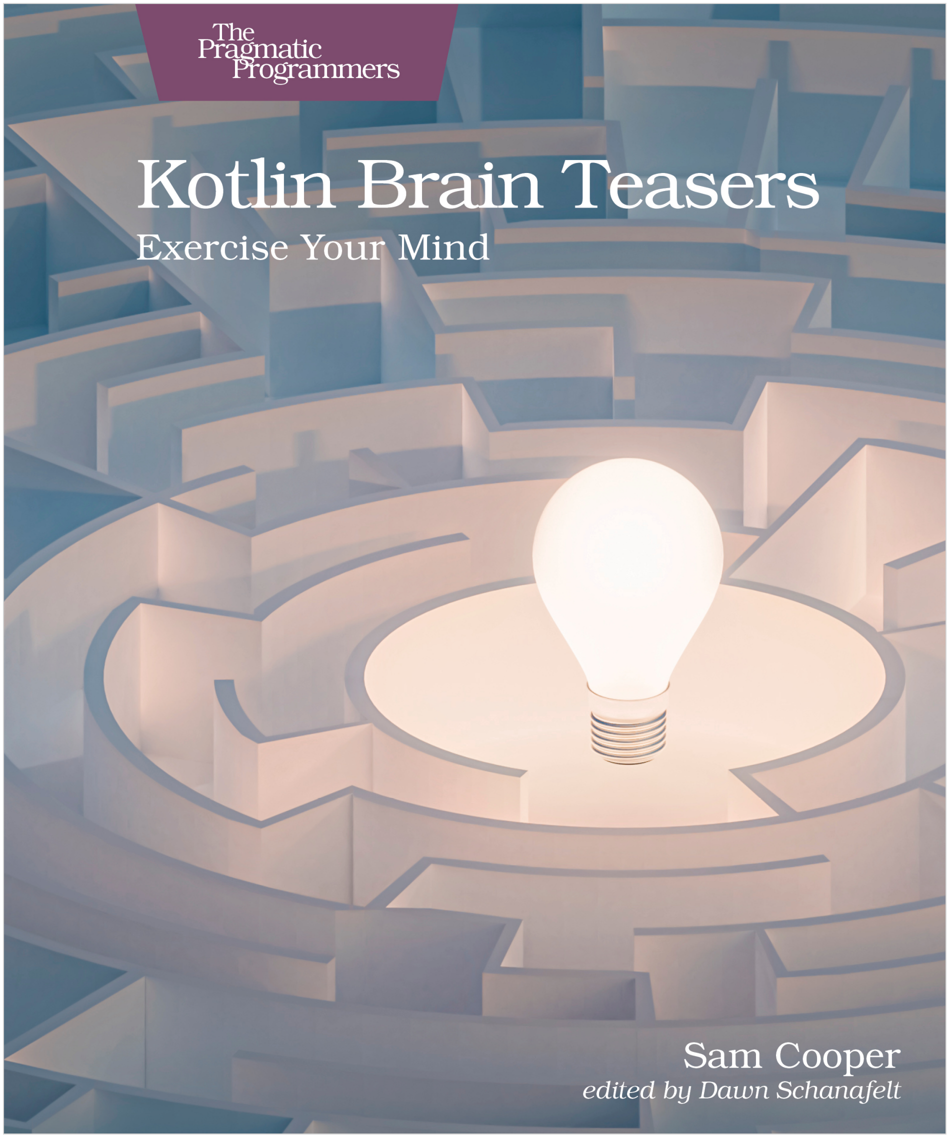
The Pragmatic Programmers
Kotlin Brain Teasers
Exercise Your Mind
Sam Cooper
edited by Dawn Schanafelt

Puzzle 17

A Class With No Name?

src/commonMain/kotlin/com/example/puzzles/syntax/declaration/ClassNames.kt

```
enum class

private enum class Visibility { Visible, Hidden }

fun main() {
  println("Nothing to see here…")
}
```

Guess the Output

Try to guess the output (or error) before moving to the next page.

The program displays the following output:

```
Nothing to see here…
```

Discussion

This puzzle's first line of code is an unfinished `enum class` declaration that's missing its name and body. So why doesn't the compiler reject it?

A class declaration doesn't end at the first line break—it can span multiple lines, continuing until the compiler determines that it's syntactically complete. The formatting makes it look like `private` is the start of a new declaration, but in fact, it's the end of the first one, and acts as our missing class name.

A class doesn't require a body, and doesn't need to end with a final line break. That means `enum class private` is a complete and valid declaration that can be immediately followed by another class.

Out Of Context

If you've written code in other programming languages, you might be used to the idea that modifier keywords like `private` are reserved syntax, and can't be used to name your own classes and variables.

But in Kotlin, many keywords can be reused for other purposes. For example, it's fine to have a class or property named *private*:

```
private class private {
  private val private = "Shh!" // ✓ This is allowed
}
```

This works because the compiler knows what it's expecting to see in each position. Before the `class` or `val` keywords, it knows that `private` must be acting as an access modifier keyword. But after `class` or `val`, access modifiers aren't allowed, so Kotlin can deduce that `private` must be filling the role of an ordinary name instead.

Naming Convention

Kotlin class names almost always start with a capital letter, but that's just a convention, not a rule enforced by the compiler.

This trick doesn't work for all keywords. Some terms, such as *class* or *return*, are permanently reserved, and can never be used to name a class or property.

These are called *hard keywords*, and you can find a full list of them in this puzzle's *Further Reading* section.

End Of The Line

Just like keywords, line breaks can also mean different things in different contexts. When a statement or declaration is unfinished and requires more code to make it syntactically valid, the compiler will keep looking for that required code on the next line.

That's what's happening in our puzzle program. On its own, `enum class` isn't a syntatically complete class declaration. The next thing the compiler needs to see is a class name, and it'll remain in that expectant state until it finds one—no matter how much whitespace it encounters on the way.

Since the context calls for a name, not an access modifier, `private` isn't a reserved keyword, and can be used as the name of the class. The code after `private` doesn't look like any of the other optional parts of a class, so the compiler accepts `enum class private` as a complete declaration.

And unlike statements inside a function, this top-level declaration doesn't require a line break or semicolon after it. Functions and classes are allowed to appear one after the other with only a single space between them:

```
class MyClass fun myFunction() {} class MyOtherClass // ✓ This is allowed
```

That's why our second `enum class` can appear immediately after the first. Don't do this in your own code, though! The auto-formatter in IntelliJ IDEA will add a blank line before each new class and function, and you should follow the same style if you don't want to confuse yourself and your team.

What can we learn from this puzzle?

- Although some keywords are permanently reserved, others, like `private`, can be freely used to name your own classes and properties.
- The same code can mean different things depending on what the compiler's execting to see next. Apply consistent formatting to avoid confusion.

Further Reading

Keywords and operators

https://kotlinlang.org/docs/keyword-reference.html

Kotlin language reference ‣ Grammar

https://kotlinlang.org/docs/reference/grammar.html