Type-Safe Builders

We'll cover the following

- HTML builder
- An XML builder

A key benefit of using a statically typed language is verifying the soundness of code at compile time. Anytime we deviate from the syntax permitted by the language, the compiler will let us know in no uncertain terms. This prevents a variety of errors from slipping to runtime and thus saves time. When working with DSLs, however, we're inventing the syntax that's permitted. The compiler doesn't have enough details to discern if a particular property access or a method call is legitimate when used within the DSL. This is where type-safe builders come in. Using a special annotation, you can instruct the compiler to keep an eye on the scope of properties and methods. Let's explore that with a built-in example in Kotlin and then by creating our own custom builder.

HTML builder

My wife says that I'm really good at typing…backspaces. If your typing skills are like mine, you'll appreciate finding errors sooner than later. Fail fast is a virtue, and compile-time failures can save hours of runtime debugging. We enjoy good compiler support for code we write in languages like Java and Kotlin, but what about creating HTML?

Looking at the built-in type-safe HTML builder in Kotlin is a good way to learn how type safety can be provided with DSLs in general and when working with HTML in particular. Play with the HTML builder at the online try-Kotlin site.

You may create an HTML content like this:

```
<html>
  <h1>Methods' Behavior<h1>
  <p>This is a sample</p>
```

```
</html>
```

Unless you were highly observant when reading the above HTML content, you may not have noticed that there's an error in there. More complex HTML content makes finding mistakes extremely hard. Asking someone to debug, just to find a silly mistake that caused the browser to display content poorly, is time consuming, costly for the organization, frustrating, embarrassing, and a disgrace to humanity.

An HTML builder can help in two ways. First, it can help us to write code instead of plain text that'll generate the HTML content. Second, it can verify, before generation, if the syntax is sound.

Visit the site specific for the Kotlin HTML builder example and replace the entire `main()` method with the following code:

```
fun main() {
  val result =
    html {
      h1 { +"Methods' Behavior" }
      p { "This is a sample" }
    }
  println(result)
}
```

Click on the `Run` button on the top right. The site reports two errors: it complains that `h1` and `p` are unresolved references. The reason is that we placed these tags at the wrong level, outside of the body tag. Let's fix that with the following code:

```
fun main() {
  val result =
    html {
      body {
        h1 { +"Methods' Behavior" }
        p { "This is a sample" }
      }
    }
  println(result)
}
```

Again, replace the entire `main()` method on the site with the corrected code and click on the `Run` button. This time the code will execute successfully and produce an HTML output.

You saw type safety in action right there. Study the code example at that site to see how it was created, and then move forward. Also take a look at Kotlinx.html, an alternative HTML builder that provides greater flexibility and error handling alternative HTML builder that provides greater flexibility and error handling.

An XML builder

Inspired by the HTML builder, let's build our own XML builder now.

First, let's look at the data for which we want to create an XML representation.

```
// xmlbuilder.kts
val langsAndAuthors =
  mapOf("JavaScript" to "Eich", "Java" to "Gosling", "Ruby" to "Matz")
```

The `Map` contains a few language names and their authors. From this data we'll create the XML representation, with names of the languages as attributes and author names as text content. Let the fun begin.

Just like HTML, XML is a hierarchical structure of elements. We'll start with a root. At each level there may be zero or more attributes and multiple child elements. Let's define some vocabulary for our DSL: `xml` to get the DSL in motion, `element` to define an element, and attributes can go within the element declaration. We may use `text` to represent a text content for an element. With these we can create a sample of the DSL, like so:

```
val xmlString = xml {
  root("languages") {
    langsAndAuthors.forEach { name, author ->
      element("language", "name" to name) {
        element("author") { text(author) }
      }
    }
  }
}
println(xmlString)
```

xmlbuilder.kts

Now we need to build the classes and methods that will process this and similar pieces of code that use our DSL. The first line in the DSL is `xml {...}`—that looks easy to implement. We can design `xml` as a function that takes a lambda as its parameter. From within that function we can return an object of an `XMLBuilder()` that will take over the building of the XML document. In other words, `xml()` will serve as a bootstrap function. Here's the short and succinct `xml()` function:

```
// xmlbuilder.kts
```

```
fun xml(block: XMLBuilder.() -> Node): Node = XMLBuilder().run(block)
```

The function creates an instance of a yet-to-be-implemented `XMLBuilder` class. The parameter to the `xml()` function is a `block`—a lambda—with a receiver of type `XMLBuilder`. Within the `xml()` function we invoke the block with the receiver as instance of `XMLBuilder` that we just created. That tells us that any methods or functions called within the block passed to the `xml()` function will run in the context of this receiver. The reason we used `run()` is that it executes the given block of code in the context of the receiver and returns the result of the block—we saw this in Fluency with Any Object. Since the lambda parameter specifies the return type to be an instance of a `Node` class, the result of calling the `xml()` function is an instance of `Node`. We'll soon create both the `XMLBuilder` class and the `Node` class.

The first line of code within the lambda passed to the `xml()` function is a call to a `root()` function. Since the lambda runs in the context of the instance of `XMLBuilder` created within the `xml()` function, the `XMLBuilder` should have this method. Whatever the `root()` method returns, the `xml()` function will return. From the details we know so far, that has to be an instance of `Node`. Let's define the `XMLBuilder` class with the `root()` method:

```
// xmlbuilder.kts
class XMLBuilder {
  fun root(rootElementName: String, block: Node.() -> Unit): Node =
    Node(rootElementName).apply(block)
}
```

The `root()` method takes two parameters: `rootElementName` of type `String` to receive the name of the root element—`languages` in this example—and `block` for a lambda that will run in the context of a `Node` instance as receiver. The lambda won't return anything, as indicated by `Unit`. The `root()` method creates an instance of `Node`, passing the `rootElementName` as an argument to the constructor, then runs the given block, the lambda, in the context of the `Node` instance just created. The reason we use `apply()` here instead of `run()` is we don't care to return anything from the block passed to `apply()`, we merely want to run the lambda in the context of the `Node` instance and return that `Node` instance.

Within the block passed to the `root()` method we iterate over the `Map` of languages and authors, `langsAndAuthors`, and for each name and author we create a nested element. This element, also an instance of `Node`, will reside within the `Node` created by the `root()` function—that is, the receiver to the block. To achieve this behavior we can make `element()` a method of `Node`. This method will create a child `Node` and insert it into the current `Node` as its child. Each instance of `Node` needs to keep a collection of attributes, a collection of children nodes, and a text value.

In short, the `Node` should have the three properties just mentioned and two methods: `element()` and `text()`. Oh, we also need a method to create a `String` representation of the `Node`, but with proper indentation. Here's the code for the `Node`:

```
class Node(val name: String) {
  var attributes: Map<String, String> = mutableMapOf()
  var children: List<Node> = listOf()
  var textValue: String = ""

  fun text(value: String) { textValue = value }

  fun element(childName: String,
    vararg attributeValues: Pair<String, String>,
    block: Node.() -> Unit):Node {

    val child = Node(childName)
    attributeValues.forEach { child.attributes += it }
    children += child
    return child.apply(block)
  }

  fun toString(indentation: Int):String {
    val attributesValues = if (attributes.isEmpty()) "" else
      attributes.map { "${it.key}='${it.value}'" }.joinToString(" ", " ")

    val DEPTH = 2
    val indent = " ".repeat(indentation)

    return if (!textValue.isEmpty())
         "$indent<$name$attributesValues>$textValue</$name>"
       else
          """$indent<$name$attributesValues>
          |${children.joinToString("\n") { it.toString(indentation + DEPTH) }}
          |$indent</$name>""".trimMargin()
  }

  override fun toString() = toString(0)
}
```

xmlbuilder.kts

Take some time to walk through the call to the `xml()` function and how each line

Take some time to walk through the call to the `xml()` function and how each line will result in calls to create an instance of `XMLBuilder` first, and then instances of `Node`.

Let's run the code to see the output, the result of our XML building DSL code processed by the `xml()` function, and the `Node class`:

```
<languages>
  <language name='JavaScript'>
    <author>Eich</author>
  </language>
  <language name='Java'>
    <author>Gosling</author>
  </language>
  <language name='Ruby'>
    <author>Matz</author>
  </language>
</languages>
```

The most exciting feature of Kotlin used in this design is the lambdas with receiver, both in the `xml()` function and in the `element()` method. This allows the lambdas to execute in the context of an instance of `Node`, thus enabling the methods of `Node` to be called without using `this`. notation in the DSL.

In the next lesson, we'll explore how we can limit access with scope control.