Many happy early returns

Hi, my name is Maciej, and I'm a Product Marketing Manager in the IntelliJ Scala Plugin team. I started working at JetBrains as a Scala developer and moved to my current position. This means that before I stopped writing code altogether, there was a short period when I learned the insides of IntelliJ IDEA, and—among other things—I saw methods that look like *this* and *this* literally next to each other.

```
  private static Integer getOffsetToPreserve(Editor editor) {
    if (editor == null) return -1;
    int offset = editor.getCaretModel().getOffset();
    if (offset == 0) return -1;
    return offset;
  }

  //todo append $END variable to templates?
  public static void moveCaretAfterNameIdentifier(PsiNameIdentifierOwner
createdElement) {
    final Project project = createdElement.getProject();
    final Editor editor =
FileEditorManager.getInstance(project).getSelectedTextEditor();
    if (editor != null) {
      final VirtualFile virtualFile =
createdElement.getContainingFile().getVirtualFile();
      if (virtualFile != null) {
        if (FileDocumentManager.getInstance().getDocument(virtualFile)
== editor.getDocument()) {
          final PsiElement nameIdentifier =
createdElement.getNameIdentifier();
          if (nameIdentifier != null) {

 editor.getCaretModel().moveToOffset(nameIdentifier.getTextRange().getEn
dOffset());
          }
        }
      }
    }
  }
```

This is not to say that there's anything wrong with these two pieces of code. I just find it curious that they are on two opposite ends of how to write Java code, yet they are next to each other in the IntelliJ IDEA codebase. In both cases, the logic is that we want to perform some operation. Still, it's only possible if certain

references are not null - or, in more general terms, if the arguments passed to the method fulfill certain requirements. If not, we should return early with some default value instead of performing the operation. In the first method, we use the `return` keyword. In the second case, we use many nested ifs.

So, today, we will discuss early returns—not in Java, where our options are pretty limited, but in Kotlin. Furthermore, I will try to spice it up by pushing your idiomatic Kotlin code a bit into the Scala direction to show you the alternatives to these two approaches.

// 1

The code example refers to the PSI model in the IntelliJ codebase, but it's just a mockup. We have code snippets. We will use the `convert` method for each code snippet to generate a `PSIElement`. The conversion takes time, so we don't want to call it more often than absolutely necessary. Then, when a code snippet is converted to a PSI element, we will use the `validate` method. Many code snippets are valid; others are invalid. Validation also takes time. So, we have code snippets, PSI elements, a conversion method, and a validation method. Now, I will write another method to iterate over all our code snippets in a loop and early return, in the imperative style, the first valid element. If no element is valid, the method will return `null`.

```
    fun findFirstValidElement(snippets: List<CodeSnippet>): PSIElement?
{
        for (snippet in snippets) {
            val psiElement = convert(snippet)
            if (validate(psiElement)) return psiElement
        }
        return null
    }
```

// 2

... But, you may say, okay, we clearly see why making nested if-else clauses leads to unreadable code, and besides, it doesn't let us jump out of a loop, but what's wrong with the return keyword? Well, imagine what will happen if this method grows and it's no longer just a few lines of easy-to-read code but... something like this [example]. One good idea is to split such a complicated method into a few smaller ones. But if we do that, then, look, this return here, for example, will no longer let

us jump out of the main method, only from the inner one. We will still need to check the value returned from it, decide if we want to return early once again and do it. In short, we can split that big method only to some extent - certain logic will have to stay in it. The method will stay complicated.

```
fun findFirstValidElementComplicated(snippets: List<CodeSnippet>):
PSIElement? {
    for (snippet in snippets) {
        val psiElement = nullableConvert(snippet)
        if (psiElement != null) {
            try {
                if (unsafeValidate(psiElement)) return psiElement
            } catch (e: Exception) {
                // ...
            }
            if (otherValidate(psiElement)) return psiElement
            if (psiElement is PSIVariable &&
variableValidate(psiElement))
                return psiElement
        }
    }
    return null
}

fun safe(validate: () -> Boolean): Boolean =
    try {
        validate()
    } catch (e: Exception) {
        false
    }

fun findFirstValidElementComplicated(snippets: List<CodeSnippet>):
PSIElement? {
    for (snippet in snippets) {
        val psiElement = nullableConvert(snippet)
        if (psiElement != null) {
            if (safe { unsafeValidate(psiElement) }) return psiElement
            if (otherValidate(psiElement)) return psiElement
            if (psiElement is PSIVariable &&
variableValidate(psiElement))
                return psiElement
        }
```

```
    }
    return null
}
```

Besides, suppose that you're not familiar with this code, that you're already tired after a long day, and the last thing you need to do today before going home is to look for a bug or to make a minor modification; then it will be easy for you to miss that little `return` hiding somewhere there. I know you may think, "I will be careful", but it would be even better if the code were written so that you didn't have to be careful.

// 3

First, let's go along the line of the least resistance. Baby steps. Every standard collection in Kotlin defines the find method. `find` returns the first element of the collection satisfying a predicate — or it will return null if no element fulfills the requirements. As the predicate, we have to use both conversion and validation. We get a code snippet, convert it, and check if it's valid. If it is, then we stop iterating our snippets and return them. But what we return is a code snippet. So, we use the question mark operator and the let method to convert this found code snippet again into a PSI element.

```
fun findFirstValidElement(snippets: List<CodeSnippet>): PSIElement? =
    snippets
        .find { validate(convert(it)) }
        ?.let { convert(it) } // complex conversion done twice!
```

// 4

As I mentioned before, conversion is costly, so we can't accept doing it twice. But can it be done better? Yes, of course.

As you may know, on top of `find`, there is also the `first` method that works similarly, but it throws an exception if no valid element is found. And there is `findOrNull`, which works precisely like `find`. And there's `firstNotNullOf`, which takes a mapping function, applies it to every element, and returns the first mapped element that is not null. If no such element is found, `firstNotNullOf` throws an exception. So that's still not ideal, but we're getting close. Fortunately, there is... you guessed it... firstNotNullOfOrNull, which takes a mapping function, returns the

first mapped element that is not null - or null otherwise.

I took the liberty of writing an alias for `firstNotNullOfOrNull`. I called it `collectFirst`, which is the name of a method in Scala that does roughly the same thing.

```
inline fun <T, R: Any> Iterable<T>.collectFirst(transform: (T) -> R?):
R? =
    firstNotNullOfOrNull(transform)

fun findFirstValidElement(snippets: List<CodeSnippet>): PSIElement? =
    snippets.collectFirst {
        convert(it).takeIf { validate(it) }
    }
```

// 5

Just look at it. It's an expression; it's three lines of code - not counting that closing brace - and you can just read it and immediately understand what it does. But of course, this is a happy path, and just a moment ago, I criticized how `return` keywords become a nuisance when the code gets complex. That complicated version of the imperative method had a nullable conversion, a validation that could throw an exception, and alternative validations where we look for an element valid in one way of many. Let's try to write the same thing in this new, functional style:

```
fun findFirstValidTwoElements(snippets: List<CodeSnippet>): PSIElement?
=
    snippets.collectFirst {
        nullableConversion(it)?.takeIf {
            safe { unsafeValidation(it) } ||
            otherValidation(it) ||
            (it is PSIVariable && variableValidation(it))
        }
    }
```

// 6

Not so bad, is it? The question mark operator takes care of nullability, and various validations are connected with "or". And that's it.

But if you don't mind writing code that is more verbose yet easier to read, especially when it comes to that last line where we check the element's type, since Kotlin 2.1, we have a new feature - the when clause with guards.

```
fun findFirstValidManyElements(snippets: List<CodeSnippet>): PSIElement?
=
    snippets.collectFirst {
        when (val el = nullableConvert(it)) {
            null                                    -> null
            is PSIVariable if variableValidate(el) -> el
            is PSILiteral  if literalValidate(el)  -> el
            else           if validate(el)         -> el
            else                                    -> null
        }
    }
```

You can see how it is similar to placing ifs and returns. The order of lines here is from the most specific to the least specific. First, we deal with the possibility that `el` is null. Then, we check if it's a variable or a literal, and so if it's suitable for one of the special validations for variables and literals, and only if not, then do we try the generic validation.
I like it. It reads well, and it's easy to expand with more logic. Of course, I'm biased —this is quite similar to how I would write this code in Scala.

// 7

There is one more approach to this issue that I would like to show you. A lazy collection, or a sequence as it is called in Kotlin, is a collection that evaluates its elements only when we call for their values. If, for example, I have:

```
val squared = listOf(1, 2, 3).map { it * 2 }
print(squared[1])
```

This is a regular collection, so Kotlin will first create a list with three elements, then eagerly create a second list with squared elements, and then display the second element of that new list. It doesn't display squared[0] and squared[2], but they are created anyway.

But if this is a lazy collection:

```
val squared = sequenceOf(1, 2, 3).map { it * 2 }
println(squared.elementAt(1))
```

then something different happens. Kotlin creates a sequence instead of a list, which has an overloaded `map` method. The new map method does not create a new sequence of squared numbers, but a list of lambdas it * 2. Only when we display squared.elementAt(1), Kotlin calls a lambda under that index and displays the result. It's not perfect: the value for the element under the zero index will be calculated too because the sequence always progresses from the start to the given element, and the calculated values are not memorized, so if we call it squared.elementAt(1) for the second time, Kotlin will again call the lambdas.

But at least the elements after the one we are interested in are not calculated. And we can take advantage of that.

```
fun findFirstValidElement(snippets: List<CodeSnippet>): PSIElement? =
    snippets
        .asSequence()
        .map { convert(it, verbose = true) }
        .firstOrNull { validate(it, verbose = true) }
```

or for something more complicated:

```
fun findFirstValidElementUnsafe(snippets: List<CodeSnippet>):
PSIElement? =
    snippets
        .asSequence()
        .mapNotNull { nullableConvert(it, verbose = true) }
        .firstOrNull {
            safe { unsafeValidate(it, verbose = true) } ||
            otherValidate(it, verbose = true)
        }
```

// 8

And since we have a few more minutes... So far, we have discussed situations when a conversion fails, a validation fails, and a validation returns false. But every time, we ignore why that happened. That's the problem with relying on null to convey that something didn't work out—it's as if we only had one bit for this information.

Of course, the Result class in Kotlin is used chiefly when the failure is solely because of an exception being thrown. We have other cases where there is a failure but no exception. Besides, I would avoid handling exceptions at all.

Instead, let's use the ArrowKt library and the `Either` class there. `Either` is a structure that can hold *either* one of two data types. We call them `Left` and `Right`; by convention, we use `Right` to pass valid data, while `Left` is for errors.

First, let's wrap up our convert and validate methods:

```
enum class PSIError {
    FailedConversion, FailedValidation, InvalidElement
}

fun eitherConversion(codeSnippet: CodeSnippet): Either<PSIError,
PSIElement> =
    nullableConvert(codeSnippet)
        ?.let { Either.Right(it) }
        ?: Either.Left(PSIError.FailedConversion)

fun eitherValidation(psiElement: PSIElement): Either<PSIError, Boolean>
=
    Either
        .catch { unsafeValidate(psiElement) }
        .mapLeft { PSIError.FailedValidation }
```

As you can see, instead of nulls and exceptions, we now use an enum called PSIError with three possible values. The eitherConvert method calls nullableConvet, and if it returns null, we turn it into Left of PSIError.FailedConversion. If the conversion is successful and we get a PSIElement, we pass it as Right of PSIElement.

```
fun convertAndValidate(snippet: CodeSnippet): Either<PSIError,
PSIElement> {
    val el = eitherConvert(snippet)
    val validated = el.flatMap { eitherValidate(it) }
    return when (validated) {
        is Either.Left                    -> validated
        is Either.Right if validated.value -> el
        else                              ->
Either.Left(PSIError.InvalidElement)
    }
}

fun findFirstValidElement(snippets: List<CodeSnippet>): PSIElement? =
    snippets.collectFirst { convertAndValidate(it).getOrNull() }
```