Extension Methods in Scala and Kotlin

By Brandon Raboy

Section 1 - Thesis:

In the 1940s, the first electrically-powered computer was created. Ten years later, the inventions of the first programming languages followed [1]. Along with the introduction of programming languages came interesting features like extension methods, which are methods added to objects after the original objects are compiled [2]. By tracing the history of programming languages, and more specifically Scala and Kotlin, it is easier to understand how extension methods work, how they are important, and how they compare between the two languages.

Section 2 - History of Programming Languages:

The earliest computers had limited speed and memory which required programmers to write high-level assembly language programs, proving that programming in this way required a great deal of effort. With this realization, the first functioning programming languages designed to communicate instructions with a computer were developed in the 1950s. A report created by a group of American and European computer scientists, the *ALGOL 60 Report*, discussed three major programming language innovations. The first innovation was nested block structures, which were code sequences and declarations that could be grouped into blocks without having to explicitly name procedures. The second innovation was lexical scoping, in which a block could have its own private variables, procedures, and functions, which

were segregated from the other parts of the code. Finally, the third innovation, a mathematical notation called the Backus-Naur form (BNF), outlined how each language contained some variant of the BNF notation in the context-free portions of their syntax. Another report, *ALGOL 68*, also produced some key ideas. Anonymous routines and a recursive typing system with higher-order functions allowed for more orthogonality in syntax and semantics [1]. Full language syntax and semantics also became formally defined using Van Wijngaarden grammar, a two-level grammar that converts an infinite context-free grammar into a finite number of rules [1, 3]. During the 1960s and the 1970s, fundamental paradigms were established that paved the way for future programming languages [1]. Paradigms are used for classifying programming languages based on their features [4]. In the 1980s, imperative languages, or languages in which the programmer instructs the machine to change its state, were strengthened with the introduction of modules and performance improvement [1, 4]. Modules separated functionalities of programs which helped large-scale systems [1, 17]. Concurrently, the RISC movement postulated that hardware should be designed for computers rather than human assembly programmers, which sparked the development of processer speed improvements. Finally, in the 2000s, Scala and Kotlin were developed [1].

Section 3 - Scala:

Scala is a strong, statically-typed programming language that supports object-oriented programming and functional programming [5]. Weakly typed languages allow for implicit conversions between unrelated types while strongly typed languages

do not allow this. Statically-typed languages execute type-checking, the process of verifying and enforcing constraints on types of values at compile-time, while dynamically-typed languages execute type-checking at runtime [6]. Compile-time is when the code is converted to machine code while runtime is when the program is running [7]. Object-oriented programming consists of grouping instructions with the part of the state they operate in while functional programming involves declaring the desired result as the value of a series of functional applications [4]. Scala was designed to be concise and a remedy for Java's implementation of generics, forced object-oriented programming, unsigned number handling, implementation of floating-point arithmetic, and security vulnerabilities [5, 18].

Some key features of Scala include syntactic flexibility, a unified type system, for-expressions, functional tendencies, object-oriented extensions, an expressive type system, and type enrichment. Some examples of syntactic flexibility in Scala include semicolons being unnecessary, any method can be used as an infix operator, methods `apply` and `update` have syntactic short forms, distinguishing between no-parentheses and empty-parentheses methods, methods ending in a colon expect the argument on the left side and the receiver on the right side, class body variables can be implemented as separate getter and setter methods, the use of curly braces instead of parentheses is allowed in method calls, and for-expressions can accommodate any type that defines methods like `map`, `flatMap`, and `filter`.

In Scala's unified type system, all types inherit from a top-level class `Any`, whose children are `AnyVal` and `AnyRef`, which means that boxing and unboxing are transparent to the user [5]. Boxing is the process of implicitly converting value types to

object types. Unboxing is the process of explicitly converting object types to value types [8].

For looping through an iterator, Scala has `for`-expressions, which are similar to list comprehensions in Haskell and Python. These expressions use the keyword `yield` which allows a new collection, a class used to represent a group of similar-typed items, to be produced by iterating over an existing collection [5, 16]. The compiler translates these expressions into `map`, `flatMap`, and `filter` calls. If yield is not used, the code is converted to a `foreach` loop. Iterating over a map will return a set of key-value tuples, a set of compounded values in a fixed order structure [5, 15]. Pattern matching, the act of checking a given sequence of tokens for the presence of some pattern, enables the tuples to be decomposed into separate key and value variables [5, 9]. Because Scala provides several capabilities in functional programming, Scala programs can be written in a functional style. For example, Scala cannot distinguish between statements and expressions, meaning that all statements are expressions that evaluate to a value [5]. Scala also has type inference, in which types of expressions are automatically detected [5, 10]. Also, anonymous functions exist, which are functions that are not bound to identifiers [5, 11]. These functions automatically capture any variables that are available in the enclosing function and those variables will be modifiable after the function returns. Scala also distinguishes between mutable and immutable variables by using the `var` keyword for mutables and the `val` keyword for immutables. Immutable variables are more commonly used because they are data structures that always return an updated copy of an object instead of replacing the existing one. Scala can also lazily evaluate expressions, meaning that the program will evaluate the

expression the first time the variable is referenced. The keywords `lazy`, `Stream`, and `view` are used in lazy evaluation. Also, since every value is an object in Scala, they are described by classes and traits. Traits are similar to regular, abstract classes, but lack class parameters. The `super` operator can be used to chain traits using composition and inheritance. A technique called type enrichment, allows new methods to be used as if they were added to existing types. This is accomplished by declaring an implicit conversion on the type that the method is added to. Then, the new type wraps the old type with the method [5].

Section 4 - Kotlin

Kotlin is a statically-typed programming language with type inference. It was developed to surpass Scala's slow compile times and to compile as quickly as the Java programming language. Since Kotlin was designed to be fully interoperable with Java, its design gives a similar feel with different features. Some examples include semicolons being optional as a statement terminator, variable declarations and parameters requiring the data type to be after the variable name, variables in Kotlin being immutable by using the `val` keyword or mutable by using the `var` keyword, classes being final by default, class members being public by default, and supporting procedural programming with the use of functions. Kotlin's programming style allows users to define static objects and functions at the top level of the package without using the class level. Syntactically, Kotlin has important distinctions between other languages. To enter a Kotlin program, the main function, `func main()`, is used. Other syntax features include arguments being unpacked using the spread operator, destructuring declarations, nested functions,

in which local functions can be declared inside of other functions, deriving a new class from another requires the base class to have the `open` keyword, abstract classes being open by default, primary and secondary constructors, sealed classes, data classes, null safety, and lambdas [12].

Section 5 - Extension Methods and Conclusion:

Extension methods are features of Scala and Kotlin. Extension methods benefit users in that they centralize common behavior, better loose coupling, enable fluent application interfaces, increase productivity, increase performance, and alleviate the need for a common base class. Extension methods allow features to be implemented once without the need for inheritance. Also, type construction and conversion can be implemented as extension methods. Fewer constraints are placed on class hierarchies to construct fluent interfaces, which improves the productivity and performance of the program. With extension methods, there is no need to derive from a base class because implementation is available for all instantiations of the generic type [2].

In Scala, extensions are defined using the `extension` keyword and the method can be called using `str.toSnakeCase` [13]:

```
          object StringExtensions {
               extension (str: String) {
                    def toSnakeCase = {
                    str.replaceAll("([A-Z])", "_" +
"$1").toLowerCase
                    }
               }
}
```

Multiple extension methods can be defined in the same class [13]:

```
          object StringExtensions {
               extension (str: String) {
                    def toSnakeCase = {
               str.replaceAll("([A-Z])", "_" + "$1").toLowerCase
                    }
                    def isNumber = {
          str.matches("[0-9]+")
                    }
               }
}
```

It is also possible to use generic extensions without restriction [13]:

```
     object GenericExtensions {
          extension [T](list: List[T]) {
               def getSecond = if(list.isEmpty) None else
list.tail.headOption
          }
}
```

Extension methods can also have type constraints [13]:

```
     extension [T: Numeric](a: T) {
          def add(b:T): T = {
               val numeric = summon[Numeric[T]]
               numeric.plus(a, b)
          }
}
```

In Kotlin, extension methods are written using the receiver class as part of the function name and the receiver can be accessed using keyword `this` [14]:

```
fun String.escapeForXml() : String {
  return this
    .replace("&", "&amp;")
    .replace("<", "&lt;")
    .replace(">", "&gt;")
}
```

It is also possible to write generic extension methods for multiple types [14]:

```
fun <T> T.concatAsString(b: T) : String {
    return this.toString() + b.toString()
}
```

Infix extension methods allow methods to be called without using periods or brackets [14]:

```
infix fun Number.toPowerOf(exponent: Number): Double {
    return Math.pow(this.toDouble(), exponent.toDouble())
}
```

To call this method [14]:

```
3 toPowerOf 2 // 9
9 toPowerOf 0.5 // 3
```

Finally, writing operator methods allows the programmer to use use the shorthand method rather than calling the full method name [14]:

```
operator fun List<Int>.times(by: Int): List<Int> {
    return this.map { it * by }
}
```

To call this method [14]:

```
listOf(1, 2, 3) * 4 // [4, 8, 12]
```

By examining the origin of programming languages and how Scala and Kotlin came to be, it is understandable that features like extension methods would exist to improve the efficiency of implementing code.

References:

[1] Contributors to Wikimedia projects, “History of programming languages - Wikipedia,” Wikipedia, the free encyclopedia, Aug. 12, 2004. https://en.wikipedia.org/wiki/History_of_programming_languages.

[2] Contributors to Wikimedia projects, “Extension method - Wikipedia,” Wikipedia, the free encyclopedia, Jan. 06, 2007. https://en.wikipedia.org/wiki/Extension_method.

[3] Contributors to Wikimedia projects, “Van Wijngaarden grammar - Wikipedia,” Wikipedia, the free encyclopedia, Feb. 25, 2005. https://en.wikipedia.org/wiki/Van_Wijngaarden_grammar.

[4] Contributors to Wikimedia projects, “Programming paradigm - Wikipedia,” Wikipedia, the free encyclopedia, Mar. 01, 2003. https://en.wikipedia.org/wiki/Programming_paradigm.

[5] Contributors to Wikimedia projects, “Scala (programming language) - Wikipedia,” Wikipedia, the free encyclopedia, Mar. 07, 2004. https://en.wikipedia.org/wiki/Scala_(programming_language).

[6] “Statically v. dynamically v. strongly v. weakly typed languages,” Educative: Interactive Courses for Software Developers. https://www.educative.io/edpresso/statically-v-dynamically-v-strongly-v-weakly-typed-languages.

[7] S. Roy, “Runtime vs. Compile Time,” Baeldung, Oct. 13, 2021. https://www.baeldung.com/cs/runtime-vs-compile-time.

[8] BillWagner, “Boxing and Unboxing - C# Programming Guide | Microsoft Docs,” Developer tools, technical documentation and coding examples | Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/boxing-and-unboxing.

[9] Contributors to Wikimedia projects, “Pattern matching - Wikipedia,” Wikipedia, the free encyclopedia, Jul. 27, 2003. https://en.wikipedia.org/wiki/Pattern_matching.

[10] Contributors to Wikimedia projects, “Type inference - Wikipedia,” Wikipedia, the free encyclopedia, Jul. 27, 2003. https://en.wikipedia.org/wiki/Type_inference.

[11] Contributors to Wikimedia projects, “Anonymous function - Wikipedia,” Wikipedia, the free encyclopedia, Sep. 16, 2006. https://en.wikipedia.org/wiki/Anonymous_function.

[12] Contributors to Wikimedia projects, “Kotlin (programming language) - Wikipedia,” Wikipedia, the free encyclopedia, Feb. 02, 2014. https://en.wikipedia.org/wiki/Kotlin_(programming_language).

[13] baeldung, “Extension Methods in Scala 3,” Baeldung, 2021. https://www.baeldung.com/scala/extension-methods.

[14] baeldung, “Extension Methods in Kotlin,” Baeldung, Sep. 03, 2021. https://www.baeldung.com/kotlin/extension-methods.

[15] Contributors to Wikimedia projects, “Product type - Wikipedia,” Wikipedia, the free encyclopedia, Dec. 26, 2007. https://en.wikipedia.org/wiki/Product_type.

[16] Techopedia, “What is a Collection? - Definition from Techopedia,” Techopedia.com, Dec. 11, 2011. https://www.techopedia.com/definition/25317/collection.

[17] Contributors to Wikimedia projects, “Modular programming - Wikipedia,” Wikipedia, the free encyclopedia, Aug. 29, 2004. https://en.wikipedia.org/wiki/Modular_programming.

[18] Contributors to Wikimedia projects, “Criticism of Java - Wikipedia,” *Wikipedia, the free encyclopedia*, Jun. 11, 2006. https://en.wikipedia.org/wiki/Criticism_of_Java.