Kotlin

BTI7311 (Computer Science Seminar), Autumn Term 2015/16, Topic No. 21

Table of Contents

Written by
Strub, Thomas Reto (strut1@bfh.ch)
Touw, Marc (touwm1@bfh.ch)

Evaluated by
Haenni, Prof Dr Rolf (hnr1@bfh.ch)

Handed in
17 January 2016

Presented
by Strub, Thomas Reto (strut1)
21 January 2016

1 Introduction

Developed by JetBrains since 2010 and publically unveiled one year later, Kotlin is yet another object-oriented programming language targeting the Java Virtual Machine (JVM). Ever since the public presentation its developers had to defend themselves for creating a new language instead of supporting the ones already existing [2].
JetBrains is a company specialised in creating development tools for the Java, .NET, C/C++ and web environments. They have years of experience with both the Java language and platform because their products are mostly written in Java. And they repeatedly mentioned the many flaws of the Java language cause them severe problems. Because of these problems, JetBrains evaluated several options to increase the company's own productivity. They argue that creating a new language was the most feasible one. They do not try to hide that – as a profit-oriented company – they started this project in their very own interest. Nevertheless, they expect the global developer community to profit from this new language equally. Kotlin is designed as a modern language to be actively used in productive environments.
Kotlin is statically typed, easy to learn and fully compatible with Java. This means developers will be able to gradually shift from Java to Kotlin without facing significant compatibility problems. JetBrains themselves are using Kotlin for new developments and are simultaneously shifting their codebase from Java to Kotlin. After all this is why they started the Kotlin project in the first place: to be able to use a modern programming language which is easy to read, requires less code to be written and has more expression power. They stated that, in November 2015, they already had quarter a million lines of code in production – only in projects other than Kotlin itself.
In 2011, the Kotlin developers stated they knew it would take years to get a new language to really take off. And finally, in November 2015, JetBrains released version 1.0 Beta [3]. Both the syntax and standard library were deemed mostly finished and no further major changes were scheduled. The release of a fully finalised productive version 1.0 was expected to take place soon afterwards.
It is the official company policy not to force customers to use the new language but rather to use it in the company itself and also offer the customers to benefit from is conciseness. All the company's tools for developing in Java, Scala and other Java platform-targeting languages are to be continued alongside the new tools for developing in Kotlin. In the language reference [1], JetBrains writes developers who are happy with Scala will probably not want to change to Kotlin but also that is not just a Scala clone but rather an independent language and partially targeting developers not interested in or fully convinced by Scala.
Unlike other JVM-executed languages, Kotlin officially is 100% compatible with Java as-is. Even existing Java frameworks such as JPA will continue to work with Kotlin applications. Many annotations exist to influence the way the Kotlin compiler compiles the source code to Java bytecode. Although these annotations have no influence on the way the program works when using Kotlin only, they can be used to change default class names, checked exception specifications, make members static and control other features not used in Kotlin.

Readers of this paper are expected to be familiar with the Java programming language and platform (both the Standard Edition and parts of the Enterprise Edition). Although Kotlin is discussed mainly from a Java perspective, additional experience with C# (or Visual Basic .NET), Scala, Groovy, Ceylon, or Swift may help catching the essence of the Kotlin programming paradigms.
Most code examples in this paper are taken from one continuous application (a simple address book) which is available on GitHub under the following link: https://github.com/StrubT/CSSemKotlin.

2 Basic Syntax

The following chapter will outline the basic features and their syntax so readers will subsequently be able to identify the main idea behind a given Kotlin source code [1].

2.1 Control Flow and Special Concepts

The two most striking differences between the Java and Kotlin syntaxes are that semicolons are usually not needed (yet allowed) in Kotlin and that the type declarations come after the name, separated by a colon.

2.1.1 Global Functions

In contrast to Java, functions can be declared globally in Kotlin. They are declared using the keyword `fun` followed by the generic parameter list with names and types in pointy brackets, the function name, the parameter list with names and types in round brackets, and the return type. As in Java the function body is surrounded by brackets and the value is returned using the keyword `return`. The Java `void` keyword is replaced with the `Unit` type and value in Kotlin, however, it too is optional.

```
fun <E : PersistentEntity> merge(entity: E): E {
 return entityManager.merge(entity)!!
}
```

Unlike Java, the curly brackets can be omitted if they are not needed, for example a single-expression function like above can be shortened to one line as follows.

```
fun <E : PersistentEntity> merge(entity: E) = entityManager.merge(entity)!!
```

2.1.2 Global and Local Variables

Variables too can be declared either locally or globally. They can either be declared read-only using the `val` keyword or variable using `var`. Read-only variables can be assigned a value only once, similar to final variables in Java. If a variable is declared and immediately assigned a value, the type can be omitted and will be inferred by the compiler.

```
val addressBook: AddressBook = ServiceLoader.load(/*...*/).single()
val addressBook = ServiceLoader.load(AddressBook::class.java).single()
```

2.1.3 Decisions

Clearly Kotlin supports basic `if`-`else if`-`else`-decisions. But unlike Java, these decisions can also be used as expressions and thus be evaluated. Therefore the ternary if (`?:`) is not implemented.

```
return if (n == 0) 1 else n * factorial(n - 1)
```

This also works with more complicated examples.

```
return if (n in 0..1) 1
else if (n > 0) n * factorial(n - 1)
else throw IllegalArgumentException("Cannot calculate factorial of negative
numbers.")
```

Kotlin provides a more powerful replacement for the Java `switch`, called `when`. It, too, is an expression. Unlike Java, it supports not only value comparison, but also range and type checks and of course, an `else` case used if no other condition was met. It can also be used without an initial object in round brackets in which case it behaves much like an `if`-`else if`-`else` expression.

```
when (x) {
 0                -> print("x == 0")
 parseInt(string) -> print("string encodes x")
 "hello", "Hello" -> print("x says hello")
 in 1..10         -> print("x is in the range 1..10")
 !in validNumbers -> { print("x is invalid"); doSomething() }
 is Iterable<*>   -> print("x is iterable")
 else             -> throw IllegalArgumentException("unknown argument")
}
```

2.1.4 Loops

Kotlin supports the traditional `do`-loop

```
var fac = n
var res = fac--.toLong()
while (fac > 1)
 res *= fac--
```

as well as the `do`-`while`-loop.

```
var fac = n
var res = 1L
do res *= fac--
while (fac > 1)
```

It also supports a `for`-loop which is actually a `for`-(each-)`in`-loop looping over all elements in an array or collection.

```
for (element in this)                          //`this` and `other` are lists
 if (element in other) intersection.add(element)
```

However, a traditional `for`-loop can easily be written using a range. Kotlin also provides extensions to arrays and collections to loop over the indices or even index-value pairs as shown later on.

```
var res = 1L
for (fac in 2..n)                              //or: n downTo 2
 res *= fac
```

2.1.5 Null Safety

Java, like other languages such as C/C++ and C# supports the `null` pointer / reference (also known as *The Billion Dollar Mistake*). While they may be useful sometimes, these reference can become a big problem if they are not handled properly.
C# introduced certain mechanisms to deal with nullable references and values, but Kotlin went further and makes references non-nullable by default. To make a reference nullable, a question mark (`?`) must be appended to the type.

```
var city: City = null                          //ERROR: cannot assign null
var country: Country? = null                   //OK, variable is nullable
```

This way, the compiler knows which references may contain `null` references and which are safe to access. The compiler then forces the developer to handle the references accordingly. While non-nullable references can be accessed immediately, nullable references must be checked first.

```
if (country !== null) filter.valueFilters.put("country", country.abbreviation)
```

Of course, this approach would lead to too verbose code. Instead of the standard call operator (`.`) a safe call operator (`?.`) can be used instead. This operator only accesses the reference if it is not `null`, and returns `null` otherwise. To specify a default value to use instead of the `null` reference, the binary `?:` operator (also known as Elvis operator) may be used.

```
var abbreviation = country?.abbreviation ?: "CH"
```

Sometimes, however, the developer knows there can be no `null` reference. In this case, a third operator (`!!`) exists to access the reference in an unsafe way (or throw a `NullPointerException`).

```
var abbreviation = country!!.abbreviation
```

2.1.6 String Templates / String Interpolation

String interpolation already exist in other languages such as PHP and C#, but not in Java. Kotlin decided to implement what they call string templates, regular strings with placeholders containing expressions to evaluate at runtime and place into the string. The placeholders are started with a dollar sign and complex expressions have to be enclosed by curly brackets.

```
val country = switzerland
val state = "SO"
val code = "${country.abbreviation}-$state"  //results in "CH-SO"
```

2.2 Class and Member Syntax

For bigger applications and especially when working with existing (Java) libraries, not all functionality can be achieved with global functions and not all data can be stored in global variables.

2.2.1 Classes and Singleton Objects

Just like Java, Kotlin supports classes with member functions and an object state. A basic class definition looks very similar to Java. The curly braces of an empty class can be omitted, so the following statement already creates an empty class.

```
class FXApplication
```

While the singleton pattern is generally frown upon in object-oriented languages, Kotlin does embrace it and even provides a specialised syntax for it. The following code will create an underlying inaccessible class with a single instance. In certain use cases, it is even possible to drop the object name and create an anonymous object.

```
object FXResources {

 val fxml = //...
}
```

This pattern is especially useful because Kotlin does not support static members. An object can be used instead to group all the properties and functions that would be declared static in Java.

2.2.2 Packages

A package declaration in Kotlin looks just the same as in Java. There is, however, one important difference in the way Kotlin handles packages: as C# and its namespaces, Kotlin packages do not need to be aligned with the directory structure.

```
package ch.bfh.cssem.kotlin.app.kotlin
```

Packages are used to group not just classes, but all top-level declarations such as classes, objects, properties and functions.

2.2.3 Member Functions

In Java, there are only methods: functions belonging to a class or an instance. As mentioned before, Kotlin does support global functions. But it also supports member functions which basically are the same thing as methods: functions that are members of a class or an object. These are declared in the very same way global functions are declared, so no example is provided.

2.2.4 Properties

Java uses fields (instance variables) to store state. Java programming guidelines suggest developers to make non-final fields accessible only within the class and provide getters and setters for public access. Kotlin – just like C# – groups the (optional) field, the getter and the setter together forming what is called a property. Such a property can then be used just like a field when accessing and assigning values while actually the getter and setter functions are called.
The following class contains a read-write property for the first name (field, getter and setter), a read-only property for the last name (field and getter only) and another read-write property for the identifier with a private field, a public getter and an internal setter.

```
class Person {

 var id: Int = 0
  get
  internal set

 var firstName: String = "n/a"
 val lastName: String = "n/a"
}
```

3 Advanced Features

Because Kotlin introduces a number of features previously unknown in the Java environment, this chapter will introduce some interesting new concepts.

3.1 Advanced Syntax

Kotlin combines the syntax of several popular languages and offers a wide range of features to choose from. Some of these syntactical features might seem odd to Java developers, but the following chapter will try to show how helpful they can be.

3.1.1 Type Casts and Checks

To perform type casts, Kotlin provides a pair of operators as well. The `as` operator performs an unsafe cast throwing an exception if the object cannot be cast to the desired type. The `as?` Operator performs a safe cast trying to cast but returning `null` if the object cannot be cast. The following example demonstrates how this `null` value can be handled by throwing a meaningful exception instead.

```
return entity as? E ?: throw IllegalArgumentException("Foreign implementation")
```

However, most of the time, explicit casts are unnecessary in Kotlin because smart casts can be used instead. Similar to how nullable values can be ensured not to be nullable using an `if` expression and safely accessed thereafter, the compiler recognises type checks and automatically treats the arbitrary object to be of the checked type. Type checks are performed using the operator `is` or its negated form `!is`.

```
if (filter is Country)                          //filter is automatically cast
 searchFilter.valueFilters.put("country", filter.abbreviation)
```

These smart casts can also be used inline without the introduction of a new scope. The following example checks a variable for `null` and in the same expression accesses it safely as a non-nullable variable.

```
if (it !== null && it.length > 0)           //automatically cast non-nullable
                                              and accessed safely
```

3.1.2 Destructuring

Kotlin does not support tuples, but it does support destructuring. Any type with member functions called `componentN()` with the capital N being an incremental number starting with one can be destructured into as many variables. This means that the result of `component1()` will be assigned to the first variable, `component2()` to the second, and so on. That way, maps can be iterated over directly instead of iterating over the keys and then fetching the value corresponding to the key.

```
for ((key, flt) in filter.valueFilters)      //loop over Map<String, String>
```

Similarly, both the index and the element value of a sequence can be iterated over.
Kotlin already provides generic classes for pairs and triples but data classes as described in a later chapter ought to be used because it is easier to understand what elements they encompass.

3.1.3 Visibility Modifiers

Kotlin supports four visibility modifiers: `private`, `protected`, `internal` and `public` with the latter being the default visibility in the absence of an explicit modifier. These visibility modifiers can be applied to classes, objects, interfaces, constructors, functions, properties and their setters.
When applied to a top-level declaration, `private` elements are visible only within the same file, `protected` is unavailable, `internal` makes the element visible within the same compilation module and `public` means that the element is visible everywhere.
When applied to a member declaration (one that declares a class member), `private` elements are visible only within the same class, `protected` elements are visible within the same class and any of its subclasses, `internal` and `public` retain their meaning with the exception that the class itself must be visible too wherever one of its members is to be accessed.

3.1.4 Exception Handling

Kotlin takes the usual approach to exception handling. It supports the throwing of exceptions using the keyword `throw`.

```
throw IllegalArgumentException("Cannot pass different implementation.")
```

The traditional `try-catch-finally` construct is supported as well. Multiple `catch` blocks are supported, the `catch` or `finally` blocks can also be omitted in Kotlin, but there must be at least one defined or the construct would be meaningless.

```
try {
 transaction.begin()
 val result = operation()
 transaction.commit()

 return result

} catch (ex: Exception) {
 transaction.rollback()
 throw PersistenceException("Could not successfully complete transaction.", ex)

} finally {
 println("This message will always be printed.")
}
```

Like many other constructs, `try-catch-finally` is implemented as an expression in Kotlin. So something like the `TryParse` as known from C# could be implemented as follows returning `null` instead of throwing an exception in case of a failure.

```
fun String.tryToInt(): Int? {

 return try {
  toInt()

 } catch(ex: NumberFormatException) {
  null
 }
}
```

As mentioned above, either a `catch` or a `finally` block is required. This means the `try`-with-resources as known from Java 7 is not available in Kotlin. The Kotlin library includes an extension function `use`, available on any instance of a class implementing the interface `Closeable` instead. This has a positive side effect: it is much easier to read.

```
primaryStage.scene = Scene(
                FXResources.fxml.openStream().use { loader.load<Parent>(it) })
```

Internally, it does just what a `try`-with-resources statement does: it executes the statement inside a `try-catch-finally` block, closes the instance regardless of the outcome and either returns the result or re-throws the exception.
In contrast to Java, Kotlin does not distinguish between checked and unchecked exceptions. Functions do not declare the exceptions they might throw. Sometimes, in Java, it is impossible to actually cause a method to throw a certain exception, but the exception nevertheless has to be handled for the application to compile. This can be very annoying and leads developers to leave the `catch` blocks empty which is very dangerous and thus Kotlin tries to prevent this practice.

3.2 Advanced Function and Property Features

While all function features as known from Java can be applied to Kotlin, Kotlin also supports some additional features that are described in the following chapters.

3.2.1 Default, Named and Variable Arguments

Kotlin supports default values to be specified for parameters. This feature can significantly reduce the number of function overloads necessary.

```
fun filterPeople(city: City? = null, state: State? = null,
                 country: Country? = null) {
 //...
}
```

Because when using this approach, most of the time, some arguments can be omitted, it is convenient to specify only the arguments to supply and to skip the others.
While other languages such as PHP do support default arguments but not allow the developer to skip certain arguments when invoking a function, Kotlin does supports named arguments when calling a function. This way, the developer can use the names to specify the arguments to provide in arbitrary order while using the default values for the remaining ones.

```
filterPeople(state = row.item)
```

Another useful feature during invocation is the support for variable numbers of arguments so the developer can specify the last parameter to take an arbitrary number of values separated by commas as if they were different arguments but then to combine them into an array.

```
fun main(vararg args: String) {
 //...
}
```

Java also supports this behaviour, but does not support the opposite, to supply an array with values to incorporate into the resulting array. Noteworthy is that, unlike in non-type safe PHP, an array's values cannot be *spread* across different parameters. The spread operator is the asterisk `*`.

```
Application.launch(FXApplication::class.java, *args)
```

3.2.2 Higher-order Functions

Kotlin supports higher-order functions, this means that functions can be passed as arguments to other functions. This is also supported in Java as of version 8, but Kotlin can also be compiled compatible to versions 6 or 7.
Unlike Java, Kotlin supports function types. This means that the parameter function's in- and output types can be declared directly in the higher-order function's parameter definition by specifying the parameter type enclosed by round brackets, the arrow operator `->`, and the return type.

```
public fun <K, V> forEach(map: Map<K, V>, action: (Map.Entry<K, V>) -> Unit) {
 for (element in map) action(element)
}
```

Kotlin also supports Java's approach of using *functional interfaces* (also known as *single abstract method*, SAM); but only when calling Java methods, so the following example does not work.

```
public fun <K, V> forEach(map: Map<K, V>, action: Consumer<Map.Entry<K, V>>) {
 //...
}
```

Once such a higher-order function has been declared, a *lambda expression* can be used to define the function argument's body. A lambda expression in Kotlin looks quite unusual with the parameter declaration placed inside the curly braces using the arrow operator `->` to separate it from the body possibly taking multiple lines. The evaluated result of the last expression is automatically returned.
Another important syntactical speciality worth mentioning is that the lambda expression may be written outside the round brackets if it is the last parameter. This allows for C# *LINQ*-style queries.

```
forEach(queryParameters) { param -> query.setParameter(param.key, param.value)}
```

If there is only one parameter, the parameter declaration can be omitted and a single parameter called `it` will be generated automatically.

```
forEach(queryParameters) { query.setParameter(it.key, it.value) }
```

Alternatively, an anonymous function may be used in place of a lambda expression, especially if a return type needs to be explicitly specified. The parameter types may be omitted.

```
forEach(queryParameters, fun(param): Unit = query.setParameter(/*...*/))
```

Kotlin has one additional advantage over Java when it comes to higher-order functions: the captured variables (variables from outside the lambda expression's scope) may be modified whereas in Java, such variables must be either `final` or *effectively final*.

3.2.3 Inline Functions and Reified Type Parameters

Kotlin also supports inline functions whose body is copied to wherever the function is called while the original function is still kept for compatibility.
This is particularly useful for higher-order functions because the lambda expressions do not have to be encapsulated into an object but can be inserted directly into the function body. This removes the memory and runtime penalties involved in encapsulating the lambda and capturing its closure.

```
inline fun <E : PersistentEntity> findByQuery(name: String): List<E> {
 //...
}
```

Another interesting advantage when using inline functions are reified type parameters. Java erases generic parameters at compilation time which means that a generic function does not know the actual type argument at runtime.
Instead of trying to overcome these limitations using reflection, functions can be declared inline and the type parameters reified which means that because the function body is copied to the caller location, the actual type argument is known and can be used as fully featured type.

```
inline fun <reified E : PersistentEntity> findByQuery(name: String): List<E> {
 return entityManager.createNamedQuery(name, E::class.java).resultList
}                                  //::class would be unavailable without reified
```

3.2.4 Backing Fields

Properties' getter and setter functions can be omitted and the compiler will generate a backing field and default functions. For properties only accessing from and assigning values to other properties, no backing field is generated.
But sometimes, the getter or setters need to be customised while still using a backing field. In these cases, the backing field can be accessed using the `field` keyword. It is possible to customise either the getter or the setter or both. The following example only customises the getter.

```
var state: ApiState? = null
 get() = field!!
 set
```

3.2.5 Extension Functions and Properties

Like C#, Kotlin supports extension functions. In Kotlin, extension functions have to define the *receiver type* to be extended before the function name separated by a dot. The extension function can then be called on any object of the receiver type. Inside the function, `this` references the receiver object; however, the receiver type may be defined nullable in which case `this` may exceptionally be `null`.

```
fun String.makeFilter() = "%${this.replace(Regex("[^a-z0-9]+"), "%")}%"
```

Because properties can be regarded as getter (and setter) functions, Kotlin – unlike C# – also supports extension properties. These properties, however, cannot use a backing field as classes cannot be retroactively extended with additional field.

```
val String.filter: String
 get() = "%${this.replace(Regex("[^a-z0-9]+"), "%")}%"
```

3.2.6 Operator Overloading

Operator overloading is yet another feature missing from Java but available in C++, C# and others. While operators like `+`, `==`, `>`, `[...]` and the like are only defined for predefined types in Java, C# – and Kotlin – allow the developer to define these operators for user-defined types as well. The functions to be called when using these operators need to have the `operator` modifier and match the predefined signature (name, number of parameters and their types as well as the return type) of an overloadable operator.
Operator overloading is used excessively in certain language such as C++, but Kotlin decided to reasonably limit the number of overloadable operators.
A detailed list of the overloadable operators and their corresponding function signatures is available in the Kotlin documentation; however, the following list provides a rough overview:

- Unary operators: `+`, `-`, `!`, `++`, `--` (both pre- and postfix)
- Binary operators: `+`, `-`, `*`, `/`, `%`, `..` (range), `in`, `!in` (collection inclusion)
- Assignment operators: `+=`, `-=`, `*=`, `/=`, `%=` (with fallback to binary operators)
- Indexer operator: `[...]` (with an arbitrary number of parameters, excluding 0)
- Call / invocation operator: `(...)` (with an arbitrary number of parameters, including 0)
- Comparison operators: `==`, `!=`, `>`, `<`, `>=`, `<=`

A Java developer might ask where operator overloading is useful. For example, the indexer operator is overloaded in Kotlin for lists and maps to allow access to specific elements / values in the same way as for arrays.

```
var valueFilters: MutableMap<String, String>
//...
valueFilters["country"]
```

The comparison operators are overloaded in Kotlin for classes implementing `Comparable<T>` so as not to have to call `compareTo` and checking the result all the time. Two classes extended with several operators are the standard Java classes `BigInteger` and `BigDecimal`. Instead of using the methods `add`, `subtract` and so on, the operators `+`, `-`, etc. can be used just like with integers and floating-point numbers.

3.2.7 Infix Notation

If a developer wants to emulate additional custom operators, they can use the infix notation which allows functions to be called without the usual call operator (`.`) or the round brackets.

```
infix fun <T> List<T>.intersectList(other: List<T>): List<T> {
 //...
}
```

A traditional function call looks as follows.

```
partialResults.reduce { a, b -> a.intersectList(b) }
```

The same call in infix notation looks like this:

```
partialResults.reduce { a, b -> a intersectList b }
```

3.2.8 Referential and Structural Equality

As mentioned above, the operator `==` is overloadable and therefore its mode of operation is modifiable. This makes it similar to C#, where this operator checks for structural equality (it calls the `Equals` method), which defaults to referential equality if not overloaded.
In Java, however, `==` always checks for referential equality (that the references point to the same object). To achieve the same, Kotlin provides a second set of non-overloadable equality operators: `===` and `!==` checking for referential equality.
When checking whether a value is `null`, referential equality ought to be checked because structural equality has a special treatment for `null` values.

3.3 Advanced Class, Interface and Singleton Object Features

When it comes to classes, interfaces, and singleton objects, Java and Kotlin differ significantly. The following chapters describes the features Kotlin provides and how to use them.

3.3.1 Constructors and Instantiation

Kotlin supports two types of constructors: primary and secondary constructors. Primary constructors are defined after the class name in the form of a parameter list. By prepending primary parameters with `val` or `var`, corresponding member properties are automatically generated and initialised.

```
class Person(var name: String, var emailAddress: String? = null)
```

While multiple constructors can often be merged using default arguments, secondary constructors may be needed occasionally. They have their own parameter list and body and, unlike Java or C#, must mandatorily call the primary constructor.

```
class Person(var name: String, var emailAddress: String? = null) {

 protected constructor() : this("") {
  //necessary for persistence framework
 }
}
```

Unlike secondary constructors, the primary constructor does not have a body. If an initialisation logic has to be executed, an initialisation block may be used. Because every secondary constructor must eventually call the primary constructor, it is no problem that the initialisation block is run after every initialisation no matter what constructor was called.

```
class Person(var name: String, var emailAddress: String? = null) {

 init {
  name = name.trim()
 }
}
```

To instantiate a class, the desired constructor can be called just like a function, Kotlin does not have the usual `new` keyword.

```
var strut1 = Person(" Thomas Reto Strub   ", "strut1@bfh.ch")
```

3.3.2 Overriding Members

In Java, classes and all of their members are extendable and overridable by default. In Kotlin – and C# – members must be explicitly marked to be overridable by adding the `open` modifier (`virtual` in C#). Similarly, Kotlin classes, too, are closed by default.

```
open class FXPerson(val implementation: Person) : Person by implementation {

 open val address: String?
  get() = //generic format
}

class US(implementation: Person) : FXPerson(implementation) {

 override val address: String?
  get() = //US-specific format
}
```

3.3.3 Enum Classes

Because enumerations are in fact fully-featured classes in Java, Kotlin calls them enum classes. They are classes with the `enum` modifier but otherwise equivalent to Java enumerations in both functionality and syntax.

```
enum class PersistenceStatus { VOLATILE, PERSISTENT }
```

Kotlin also allows enumerations to provide constructors and abstract members for elements to use.

3.3.4 Data Classes

A feature completely lacking in Java are data classes. Data classes offer a number of features out of the box without any boilerplate code, however they also come with a restriction. To turn a regular class into a data class, it can be modified with the `data` keyword.
A data class automatically overrides the functions `toString`, `hashCode` and `equals` based on the class' properties.

```
data class Person(var name: String, var emailAddress: String? = null)
```

Furthermore, a `copy` function is generated to create a second instance with the properties' values copied from the original instance except where explicitly overridden.

```
strut1 = strut1.copy(eMailAddress = "ths@strubt.ch")
```

Last but not least, component functions for destructuring are automatically generated as well.

```
val (name, emailAddr) = strut1                  //name and e-mail destructured
```

The mentioned restriction is that data classes cannot extend a superclass nor be extended by subclasses; they may, however, implement interfaces.

3.3.5 Inheritance

Just like Java and C#, Kotlin allows classes to extend only one class but allows them to implement as many interfaces as they want. To define the superclass and implemented interfaces, a syntax similar to C# is used. A colon is added after the optional primary constructor followed by the names of the superclass and implemented interfaces.

```
class FXWindow : Controller, Initializable, Serializable {
 //Controller: superclass, Initializable & Serializable: interfaces
}
```

The difference to C# is that, if the superclass has one or more constructors, one of them must be called directly in the inheritance declaration.

```
class FXApplication : Application() {
 //...
}
```

3.3.6 Object Expressions

When singleton objects were introduced earlier, object declarations were used. An object declaration is the declaration of a named singleton object inside a scope. An object expression is the singleton instantiation of an anonymous class. Although technically optional, it is most useful to use an object expression to anonymously extend a superclass or implement interfaces or both.

```
row.onMouseClicked = object : EventHandler<MouseEvent> {
 override fun handle(event: MouseEvent) { /*...*/ }
}
```

Kotlin does not support anonymous classes. But object expressions that can often be used instead. Because anonymous classes are almost exclusively instantiated only once, it is idiomatic to replace them with singleton objects in Kotlin.

3.3.7 Companion Objects

While Kotlin does not support static members, it supports both package-level objects and so-called companion objects. These are singleton objects declared inside a class.

```
class SearchFilter {

 companion object Factory {

  fun makeFilter(string: String): SearchFilter {
   //...
  }
 }
}
```

As there can be only one instance of a singleton object (hence the name), the companion object's members can be seen as being similar to static members. And because the object's name can be omitted both during declaration and access, they can also be called as if they were static members of the enclosing class.

```
SearchFilter.Factory.makeFilter("[country=CH] Marc Touw")
SearchFilter.makeFilter("[country=CH] Marc Touw")     //equivalent
```

3.3.8 Generics

Java uses a fairly complicated system for generic type parameters. It offers upper- and lower-bound constraints and wildcards to achieve various effects. Kotlin introduces a much simpler concept inspired by C#.
Kotlin only supports upper-bound constraints using the colon inheritance operator.

```
protected fun <E : PersistentEntity> cast(entity: Any) = entity as? E ?: //...
```

To implement scenarios where wildcards would be used in Java, Kotlin supports declaration-site variance and use-side type projections. A generic type can be declared as being either covariant or contravariant – or invariant if neither.
The simple explanation is as follows: a producer is covariant because `Producer<Base>` is a supertype of `Producer<Derived>`: the hierarchy is in the same direction (co: Latin for together, with) as the generic types' hierarchy; a consumer is contravariant because `Consumer<Base>` is a subtype of `Consumer<Derived>` which is the opposite direction (contra: Latin for opposite, against) compared to the direction of the generic types' hierarchy.
An exemplary consumer is a read-only collection. A collection producing arbitrary objects can safely be replaced by one only whose output is only strings. Hence, the keyword to declare a generic type parameter as covariant is `out`.

```
public interface List<out E> : Collection<E> {
 //...
}
```

On the opposite, a comparison method is a classic consumer. A comparator comparing only strings can safely be replaced by one comparing arbitrary objects. The corresponding keyword is `in`.

```
public interface Comparable<in T> {
 public operator fun compareTo(other: T): Int
}
```

The reason why all generic types in Java are invariant (neither co- nor contravariant) is most classes are neither mere consumers nor pure producers. A classic bad design are arrays in Java (and C#) for they are covariant. But a `String[]` cannot always be used in place of an `Object[]`; the following example will cause an `ArrayStoreException`.

```
String[] strings = { "Thomas", "Marc" };     //Java code
Object[] objects = strings;
objects[0] = 5;                               //ERROR: Integer in String array
```

This is why arrays in Kotlin are designed as invariant generic classes and collections are read-only by default. When a generic class not only consumes but also produces values, it has to be invariant. But then, declaration-site variance is only useful in certain cases.
This is why Kotlin additionally supports use-site type projections. They are used to achieve the same effect on the use-site where an object can normally be reduced to either a consumer or producer.

```
fun <T> copy(from: Array<out T>, to: Array<in T>) {

for ((index, element) in from.withIndex().take(to.size))
  to.set(index, element)                      //safe: read- / write-only arrays
}
```

4 Conclusion

Kotlin definitely has the potential to become an industry-approved alternative to the Java programming language; not only is it backed by JetBrains, a company renowned for its various excellent development environments and supporting tools, it is also fully compatible with existing Java systems. Of course, no profit-oriented company will simply switch to a new language just because it is new. But Kotlin code is not only shorter than the corresponding Java code, it is also easier to read even for people not familiar with the language.

Even though parts of the Kotlin syntax (e.g. that the type comes after the name) is unfamiliar for Java developers, the syntax as a whole is much more uniform when compared to that of Java. Kotlin also supports a number of features borrowed from or inspired by other languages. While some of these features might seem odd or even cumbersome to experienced Java developers, no one is forced to use them. Kotlin allows for developers to follow their preferred coding style.

Another heavily advertised advantage of Kotlin is that it many features introduced in Java 7 and 8 while being compatible with Java 6. In some environments, using current versions of the Java runtime is not an option; this is particularly for Android development where it is impossible to update the Java version without updating the Android platform. While other libraries are available offering similar features, Kotlin is advertised as being much smaller and thus faster to load and execute. Kotlin is also unofficially known as *The Swift of Android* as it apparently has a similar syntax and is thought to be able to become the major language for Android programming.

Two other areas where Kotlin can be used are the compilation to JavaScript code and the use as a scripting language. Neither of these possibilities were explored for this paper, but it shows that Kotlin is indeed a very versatile language that can be used in a variety of ways.

This paper only gave a brief overview over the most important and common features of Kotlin, but the official language and library documentations are very thorough and definitely worth a glance before starting to use Kotlin so as to see for oneself if there are other interesting features that can be used in to simplify coding.

References

The official Kotlin reference was used as the primary reference and is not cited throughout the paper:

[1] Kotlin Reference, 17 November 2015, https://kotlinlang.org/docs/reference/
also available as a complete PDF document: https://kotlinlang.org/docs/kotlin-docs.pdf

Entries in JetBrains' official company and product blogs were used as secondary references:

[2] Jemerov, D., Why JetBrains needs Kotlin, 2 August 2011,
http://blog.jetbrains.com/kotlin/2011/08/why-jetbrains-needs-kotlin/

[3] Breslav, A., The Kotlin Language: 1.0 Beta is Here!, 2 November 2015,
http://blog.jetbrains.com/kotlin/2015/11/the-kotlin-language-1-0-beta-is-here/

Several slides from presentations held by Kotlin lead developer Andrey Breslav were used as well:

[4] Breslav, A., Who's More Functional: Kotlin, Groovy, Scala, or Java?, 5 October 2012,
http://www.slideshare.net/abreslav/whos-more-functional-kotlin-groovy-scala-or-java

[5] Breslav, A., Language Design Trade-offs, 14 June 2013,
http://www.slideshare.net/abreslav/trade-offs-22989326

[6] Breslav, A., Kotlin: Challenges in JVM language design, 12 September 2013,
http://www.slideshare.net/abreslav/challenges-in-jvm-language-designpptx

[7] Breslav, A., Introduction to Kotlin: Brief and clear, 4 April 2014,
http://www.slideshare.net/abreslav/introduction-to-kotlin-brief-and-clear

[8] Breslav, A., Flexible Types in Kotlin - JVMLS 2015, 12 August 2015,
http://www.slideshare.net/abreslav/flexible-types-in-kotlin-jvmls-2015

Some illustrating examples were taken from or inspired by the following Wikipedia pages:

[9] Kotlin (programming language), 17 November 2015,
https://en.wikipedia.org/wiki/Kotlin_(programming_language)

[10] C# (programming language), 17 November 2015,
https://en.wikipedia.org/wiki/C_Sharp_(programming_language)

[11] Comparison of C# and Java, 17 November 2015,
https://en.wikipedia.org/wiki/Comparison_of_C_Sharp_and_Java

[12] Scala (programming language), 17 November 2015,
https://en.wikipedia.org/wiki/Scala_(programming_language)

[13] Swift (programming language), 17 November 2015,
https://en.wikipedia.org/wiki/Swift_(programming_language)