We are going to give an overview of the Kotlin programming language today, including its history, motivation for being created, interoperability with Java and some other languages, and extensive language features. We hope you can leave the workshop today with a decent understanding of how expressive and powerful Kotlin can be to create clean and maintainable software. As well, if you have an existing project or software implemented using Java, such as an Android application, we encourage you to consider the option of migrating your codebase to Kotlin, as there will be obvious benefits in doing so.

What is Kotlin? Kotlin is a multi-paradigm statically-typed language that runs on the Java Virtual Machine. It was first released in 2011 by Jetbrains, the creator of the popular Intellij IDEA IDE, and its first stable release was in in June of this year. Though a relatively new language, Kotlin has garnered plenty of support from tech giants, including Google, who announced first-class support for Kotlin as a language for Android development at their 2017 Google I/O event.

In addition, many notable companies have introduced Kotlin into their codebase, such as Amazon Web Services, Pinterest, Prezi, Coursera, Netflix, Uber, Expedia, and Trello. Corda, a distributed ledger platform for the world's largest financial institutions has over 90% Kotlin in its codebase.

Why invent Kotlin? Firstly, there was a need for a more modern language for the JVM. Secondly, many people felt that Android development using Java could me a lot more elegant and concise. Lastly, Jetbrains hoped that Kotlin could drive Intellij IDEA sales. Development lead Andrey Breslav has said that Kotlin is designed to be an industrial strength Object-Oriented language, and a "better language" than Java, but still be fully interoperable with Java code, allowing companies to make a gradual migration from Java to Kotlin.

What is Kotlin's biggest selling point? Without a doubt it has to be Kotlin's ability to be written into a Java code-base one class at a time. This and many other interop features allow Kotlin to be integrated into a project slowly and risk-free. Kotlin is able to interact with Java classes, Objects, and methods directly within the same package or from other libraries without any hassle at all. This feature works both ways though; A Java class is able to access any Kotlin method or class and use it within Java code. Beyond this, Kotlin can take advantage of Java's diverse eco-system even including Java annotations (which may not be your favourite part of Java, but are indeed used everywhere).

Here's a high-level summary of Kotlin's language features. In general, we are aiming to discuss features that Kotlin has taken from Java and modified, or features that Kotlin leverages that are not present in Java at all. As we mentioned previously, Kotlin runs on the JVM so it can be compiled once and run on any machine with a JVM. It also supports garbage collection so there's no need to clean up your memory manually like in C/C++. The language can be used for OOP programming, procedural, and functional programming. One important difference between Java and Kotlin is that Kotlin has more functional aspects and is not as strictly Object-Oriented. Like we mentioned previously, Kotlin can utilize the existing rich ecosystem of Java fully because of its interoperability. It can use any Java library or source code. It can be used for pain-free Android dev, a replacement for Java in any project, and web dev (Kotlin code can be transpiled to JavaScript).

As listed on the Kotlin website, here are some features in Kotlin but not in Java.

1. Null references are controlled by the system type (in Java, an int cannot store a null value)
2. No raw types (everything is an object)
3. Lambda expressions and inline functions
4. Extension functions
5. Null-safety
6. Smart-casts

7. Type inference
8. Data classes

The first language feature we will look at closely is type inference. When declaring a variable most of the time, the data type of the variable will not have to be explicitly stated because the compiler can often infer the type based on the context (show code). In the grand scheme of things, this is not a huge benefit, but it does shorten code as it removes the explicit declaration of data types. Another important benefit is smart casting. This is best explained through an example (show code).

A NullPointerException is the bane of existence of most Java programmers. In short, it is an error that is thrown when an attempt is made to access the fields or methods of an object that is null. And obviously, this happens when we are oblivious to the fact that an object is null (or could be null). This is when nullables, or optionals, come in. Kotlin introduced null safety so programmers can be aware of when a variable can potentially have a null value. Every data type has a corresponding nullable version of itself (show code). If a value is not declared as a nullable, then it can never be set to store a null value. A nullable variable can contain null. You may access the contents of a nullable variable by writing two exclamation marks (show code). You may also access fields and properties of a nullable variable using safe call operators, or optional chaining (show code). Nullables are important because they make the Kotlin language more expressive.

Let’s give a brief overview on some common Kotlin data structures. Kotlin distinguishes between mutable and immutable data structures in hopes of allowing easier debugging and better API designs. For example, a List is read only whereas a MutableList allows the user to actually change the contents of a list. A list is similar to an ArrayList in Java, which is essentially an array that can grow and resize as needed. In Kotlin, there is no dedicated syntax for creating collections objects; you must resort to using functions in the standard library. Another popular data structure is a hashtable. In Kotlin, we use a Map or MutableMap to store key/value pairs.

In Java 8, the introduction of lambda expressions brought along with it new ways to design and write code. Kotlin obviously supports lambda expressions too, but they are integrated differently and are much more elegant and easy to use (show code comparing Kotlin lambda vs. Java lambda). Lambdas in Kotlin can be passed directly into functions or returned from them. There is a special syntax for defining the type of a lambda (show code). Here is an example of how lambdas, in the context of higher order functions, can significantly reduce code length (show code). Higher order functions should replace traditional for loops and such whenever possible as they are highly optimized.

There is a feature in Kotlin that is not present in Java, but has variations of it appearing in more modern languages like JavaScript, C#, or Swift: extension functions. Extension functions are similar to how prototypes are used in JavaScript. In Swift, extensions are heavily used. Extension functions allow the user to add functions and computed properties to a class without actually modifying the source code inside the class. They are used often when you want to enhance, modify, or extend an existing API. Extensions allow for greater improvements in code design and readability. Here’s a simple example of how you can define an extension function and how it can be of use to you (show code).

We are going to give a brief summary of functions in Kotlin (show code). What’s neat is, unlike Java, you can set the value of a variable equal to a function. There is a special syntax for the type of a function (show code). A consequence of this capability is that functions can be passed into other functions as parameters or returned from them. A concept known as variable capture is worthy of noting. Capturing happens when a block of code like a closure or lambda is able to access or modify variables in its outer scope. Here’s the catch: in Java, the captured variables are read only, whereas in Kotlin, they are mutable. Something neat: there is a problem that asks the user to create a function that can only run once without the use of variables in the outer scope. This problem can be solved easily using Kotlin (show code).

Let’s focus on function parameters. In Kotlin, parameters of functions can take on default values specified if no values are provided to the functions when they are called. Thus, a function with default parameters in Kotlin can, in a way, take on the role of multiple overloaded functions in Java. As well, Kotlin supports named parameters. This feature greatly improves code readability, as you can specify the name of parameters explicitly when you pass in arguments to a function (show code). Sometimes it can be difficult to understand how an argument will be utilized inside a function, so the naming of parameters can give a lot of insight into the meanings behind each variable. This feature is, again, something that Kotlin leverages that Java does not. Like I mentioned previously, if a function contains a single expression, you can remove the braces surrounding a function and place an equal sign between the function signature and body directly. The return type of the function is inferred, so it does not need to be explicitly stated. Lastly, Kotlin supports variable-length parameters. A variable-length parameter is declared with the vararg keyword and can accept a variable number of arguments of a common type. These arguments can be accessed by using the variable-length parameter as an array (show code).

Kotlin allows you to take advantage of many different programing paradigms. One of those that it allows you to use is the well-known Object Oriented Programming design. Now obviously any new language should support this, since it provides many benefits that are nice to have, but Kotlin added some of its own unique features while implementing OOP. Kotlin supports classes, which can be open, default, or sealed (different inheritance allowances essentially). Unlike Java though, Kotlin is relatively unaware of static anything. Instead it has “Objects” which are essentially one instance classes. They allow you to access their fields and methods via a global instance that Kotlin takes care of. As well, any class can have a companion object for functions that would normally be static.