There has been a huge growing interest for using the newly launched Kotlin programming language in scientific and engineering applications. While Java which is one of the most widely used programming languages and also the official language of Android development, there are various reasons why Java might not always be the best option for various projects. This is just because Kotlin truly offers several features, which other traditional languages (C, C++, Java and FORTRAN) lack. Also, the historic poor performance of Java stops it from being widely used in scientific applications. Kotlin 1.2 was a major new release and a big step on the road towards enabling the use of Kotlin across all components of a modern application and multiplatform project. This study explores the major improvements and feature scopes over the year of the newly released enhanced version of Kotlin.

**KEYWORDS:** Kotlin, Programming language, Multiplatform Projects, Java, JavaScript, Android.

**INTRODUCTION:**
Kotlin is new software development language running on Java Virtual Machine, Android or browser. It is statically typed language and it can be compiled to Java. What is a universal programming language? From a unsophisticated view, the prospect could be that one language is used for all categories of programming. While that may be far-fetched in today’s complex multifaceted world, the expectation could be accustomed to single language becoming the dominant programming language. Most certainly, it is the solitary, most important language to master. [2][3]

It's true that Java was one of the top programming languages, but time flies by real soon. In the up-to-the-minute modern world, no coder can imagine coding without the integration or support of lambda functions & streams, and no programmer demands to bother with plugins to accomplish the same. The Java JVM on Android doesn't even have support for lambdas, method references, streams, try-with-resources (minSdk ≥ 19). Programmers still have to use the javax.time APIs from the ancient Java 6/7 worlds. One of the prime flaws in Java is the way it knobs the "null," leading to the feared NullPointerException (NPE), widely known as The Billion Dollar Mistake. [4][5]

This paper explores and adds significant knowledge to the field of Kotlin Programming by conducting an extensive literature review of the research done in the past and attempts to understand the changes included in the latest released Kotlin 1.2.30 as well as attempts to understand the scope in development work for multiplatform projects.

**Hindrances of Kotlin 1.0:**
IntelliJ plugin for Kotlin's initial release was extremely buggy and very far behind Java. That time, Kotlin seemed to have promise a better Java, deprived of compromises. Well, it did achieve that at the language level. However, a big part of using Java was JetBrains’ own brilliant Java tooling. Kotlin’s collection api had no correspondent to Java’s parallelStream(). This was profoundly missed by developers. It was unable to subclass the Data classes in initial release of Kotlin 1.0. Here the Kotlin plugin was having a lot of effortful work to do in demand to catch up with what JetBrains has built over 15 years for Java. The editor recurrently stopped doing the syntax highlighting, code completion etc. The only temporary solution when this occurred was to start over the entire IDE. The Call Hierarchy view recurrently was unable to show all calls to a method if that method was used aathwart both Java and Kotlin. This was a super serious bug. The Kotlin Grade integration was almost flaky in the past. Other than this, programmers were unable to see inferred variable type, Refactoring restrictions were present, No Postfix Completion, No Duplicate Detection support was present. [1]

Kotlin 1.1:
Kotlin's new version 1.1 was then released introducing a number of new language features most markedly Coroutines and improved support for its JavaScript target. All language features were supported, and there were many new tools for integration with the front-end development environment. Even though it was still considered experimental, one of the key new features in Kotlin 1.1 the Coroutines, which were available through the use of three higher-level constructs: await, await, and yield. But being experimental, Coroutines were opt-in only and there was an anxiety among programmers that the Coroutines API may change in forthcoming releases. Other significant new features added to Kotlin 1.1 were, Type aliases (which allowed programmers to define an alternative name for a type), The :: operator (to get a member reference to a method of a specific object), Data classes could now be extended, Destructuring support in lambdas was also included. [6][8]

**Multiplatform Projects:**
In Kotlin’ major release 1.1, support for multiplatform projects was included. A multiplatform project allows an individual to build multiple tiers of his/her application – backend, frontend and Android app – from the same codebase. Such a project contains both common modules, which contain platform-independent code, as well as platform-specific modules, which contain code for a specific platform (JVM or JS) and can use platform-specific libraries. To call platform-specific code from a common module, an individual can specify expected declarations – declarations for which all platform-specific modules need to provide actual implementations.

Note that multiplatform projects are presently an experimental feature; it means that the feature is ready for use, but the developers may need to change the design in the subsequent release (and if they do, they’ll surely provide migration tools for existing code) allowing programmers to reuse code between target platforms supported by Kotlin – JVM, JavaScript and (in the future) Native. In a multiplatform project, programmers have three kinds of modules:
A common module contains code that is not specific to any platform, as well as declarations without implementation of platform-dependent APIs. [7]

A platform module contains implementations of platform-dependent declarations in the common module for a specific platform, as well as other platform-dependent code.

A regular module targets a specific platform and can either be a dependency of platform modules or depend on platform modules.

While compiling a multiplatform project for a specific platform, the code for both the common and platform-specific parts is generated. A key feature of the multiplatform project support is the possibility to express dependencies of common code on platform-specific parts through expected and actual declarations. An expected declaration specifies an API (class, interface, annotation, top-level declaration etc.) An actual declaration is either a platform-dependent implementation of the API or a typealias referring to an existing implementation of the API in an external library. Here’s an example:

In common code:

```kotlin
// expected API:
expect fun hello(world: String): String

fun greet() {
  // usage of the expected API:
  val greeting = hello("multi-platform world")

  println(greeting)
}
```

In JVM platform code:

```kotlin
actual fun hello(world: String): String =
    "Hello, $world, on the JVM platform!"
```

In a cross-platform serialization codebase:

```kotlin
expect class URL(spec: String) {
    open fun getPath(): String
    open fun getHost(): String
}
```

Enhancement in Kotlin 1.2.x:

While, in Kotlin 1.1, IntelliJ officially released the JavaScript target, permitting programmers to compile Kotlin code to JS and to run it in the browser, in Kotlin 1.2, they’ve added the possibility to reuse code between the JVM and JavaScript. Now programmers can write the business logic of their application_once, and reuse it across all tiers of their application—the backend, the browser frontend and the Android mobile app. They’ve also operated on libraries to service programmers to reuse more of the code, such as a cross-platform serialization library. The release of Kotlin 1.2.30, was along with new bugfix and tooling improvements.

### IntelliJ IDEA plugin enhancements:

This release brought various enhancements in the IntelliJ IDEA Kotlin plugin, such as performance enhancements, bug fixes, and new inspections and intentions. Intentions for converting the scoping function calls:

```kotlin
val person = employee.apply { 
    log("First name: "+ first)
    log("Last name: "+ last)
  }
```

### Pasting Kotlin code into a package:

The IntelliJ plugin now allows pasting Kotlin code into a package item in the Project View, creating a new Kotlin file for the code:

![Image](image-url)

Other changes in the IDE plugin:

- Data flow analysis (‘Analyze Data Flow …’) support for mixed Kotlin and Java codebases.

- An option to create a run configuration for a Node CLI application from a main function in Kotlin/JS projects.

- Enhancements in the Rename/Move refactoring, such as warnings on possible conflicts introduced by renaming.

- Changes in the compiler:

  - The Kotlin 1.2.30 update fixes several known issues in the Kotlin compiler and includes performance improvements.

  - The compiler is now able to optimize a tail call made in a suspending function to another Unit-returning suspending function, resulting into more efficient compiled code.

### Compilation Performance:

Over the course of development of 1.2, they’ve put a lot of effort in making the compilation process faster. The official developers have already reached approximately 25% improvement over Kotlin 1.1, and see significant potential for further improvements, for the releases in 1.2.x updates. The graph below shows the difference in compilation times for two large JetBrains projects built with Kotlin:
CONCLUSIONS:
In the noteworthy facets of this JVM language, one can express of its ease of pro-
ject setup and its excellent Java interoperability. Taking the whole shebang into
account, a good Android developer, got to give Kotlin a try. Its automated syntax
converter that detects Java code as soon as you paste it in your current file, is a
boon since it converts most of the Java code impeccably. JetBrains has made a
great job on tools and is incessantly extemporizing. To get started with it, it is
therefore important to start introducing Kotlin with real and specific needs.
Accompanied by full Java compatibility and good IDE support it is envisioned to
expand code readability, which will give an easier way to encompass Android
SDK classes and accelerate development.

REFERENCES:
available at https://blog.dripstat.com/kotlin-in-production-the-good-the-bad-and-the-
Belgrade, Serbia, pp. 6-7.
kotlin Programming in Android. International Journal of Innovative Trends in Engi-
neering & Research, Vol-2, Issue 9, p.4-10.
JavaScript Support, available at https://www.infoq.com/news/2017/03/kotlin-1-1-