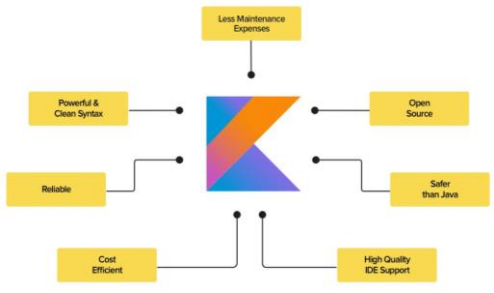
Application Delivery Fundamentals : Kotlin

Kotlin 1.7.20



Kotlin

High performance. Delivered.

consulting | technology | outsourcing

Kotlin Goals

Kotlin

- Get started with Kotlin
- Kotlin Multiplatform
- Kotlin for server side
- Kotlin for Android
- Kotlin for JavaScript
- Kotlin Native
- Kotlin for data science
- Kotlin for competitive programming

Kotlin Goals

Kotlin

- What's new in Kotlin 1.7.20
- What's new in Kotlin 1.7.0
- Basic syntax
- Idioms
- Coding conventions
- Basic types
- Type checks and casts
- Conditions and loops
- Returns and jumps

Kotlin Goals

Kotlin

- Exceptions
- Classes
- Inheritance
- Properties
- Interfaces
- Visibility Modifiers
- Extensions

Kotlin Goals

Kotlin

- Data Classes
- Sealed Classes
- Generics
- Nested Inner Classes
- Enum Classes
- Inline Classes
- Object Expressions and Declarations
- Delegation

Kotlin Goals

Kotlin

- Functions
- High Order Functions and Lambdas
- Inline Functions
- Operator Overloading
- Type Safety Builders
- Using Builders and Builder Type Inference
- Null Safety
- Asynchronous Programming

Kotlin Goals

Kotlin

- Coroutines
- Annotations
- De structuring Declarations
- Reflection
- Full Stack Web Application
- Kotlin JVM
- JAVA Comparison
- Calling Java from Kotlin

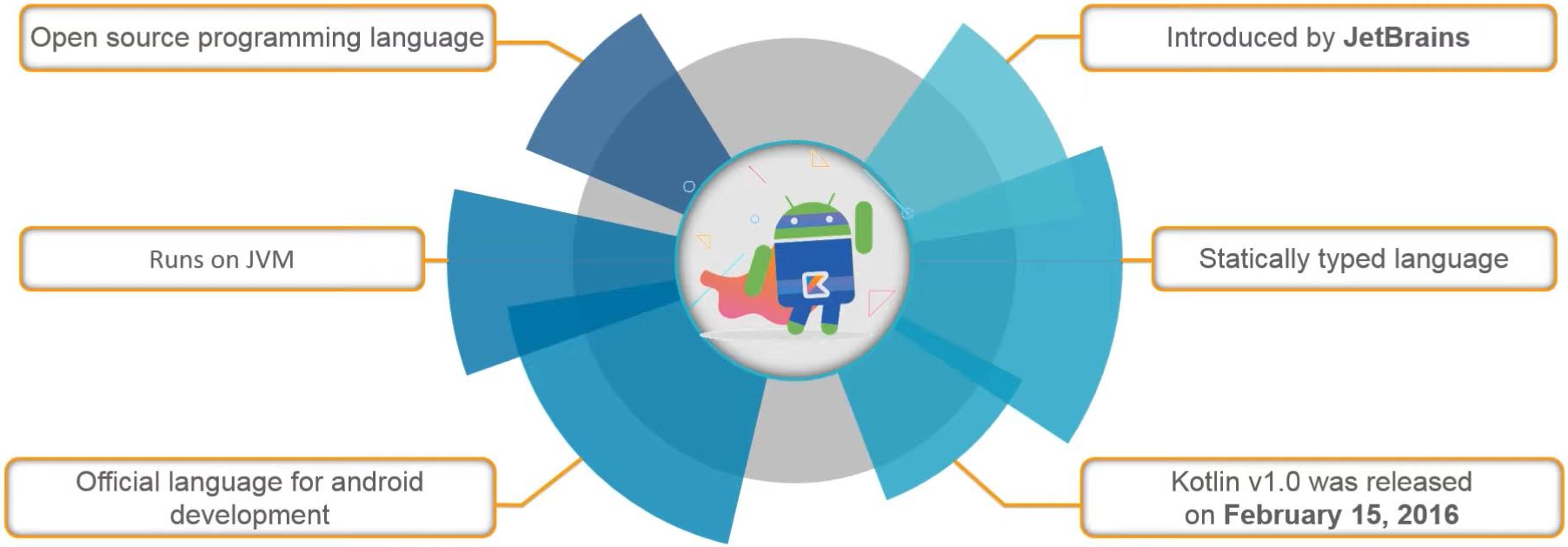
Kotlin Goals

Kotlin

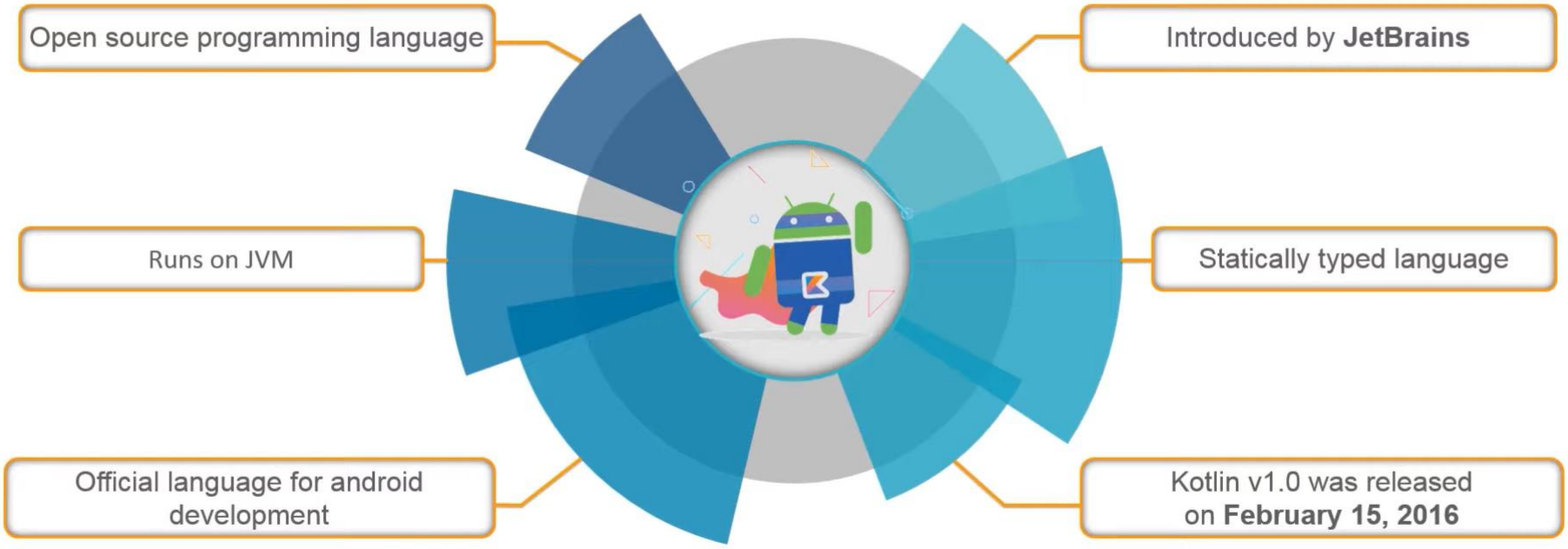
- Coroutines
- Annotations
- Destructuring Declarations
- Reflection
- Full Stack Web Application
- Kotlin JVM
- JAVA Comparison
- Calling Java from Kotlin
- Create a RESTful web service with a database using Spring Boot

What's Kotlin

- Kotlin is a modern but already mature programming language aimed to make developers happier.
- It's concise, safe, interoperable with Java and other languages.
- It provides many ways to reuse code between multiple platforms for productive programming.



What's Kotlin



Kotlin's main developer is St. Petersburg-based JetBrains, who just happen to be in the business of making a suite of the world's finest and most-used IDEs:

What's Kotlin

Kotlin

What’s Kotlin

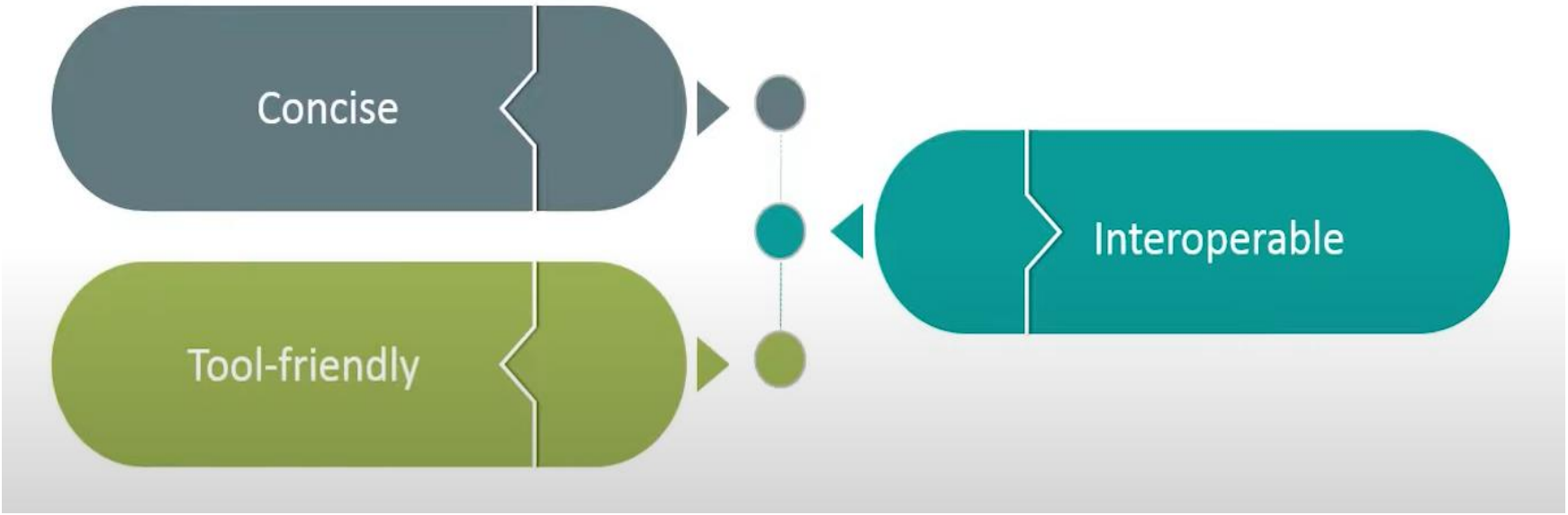
Kotlin

Statically typed

Develop Android applications

What's Kotlin

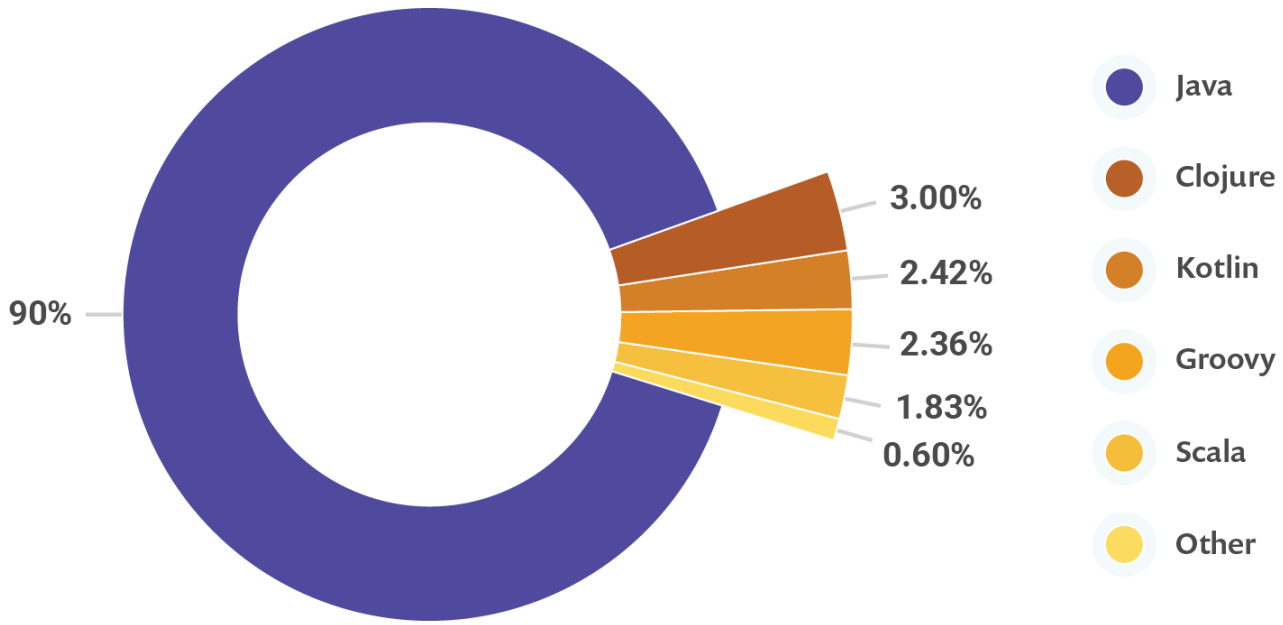
- Kotlin has impressed programmers with its friendly
- syntax, conciseness, flexibility and power



JVM Family

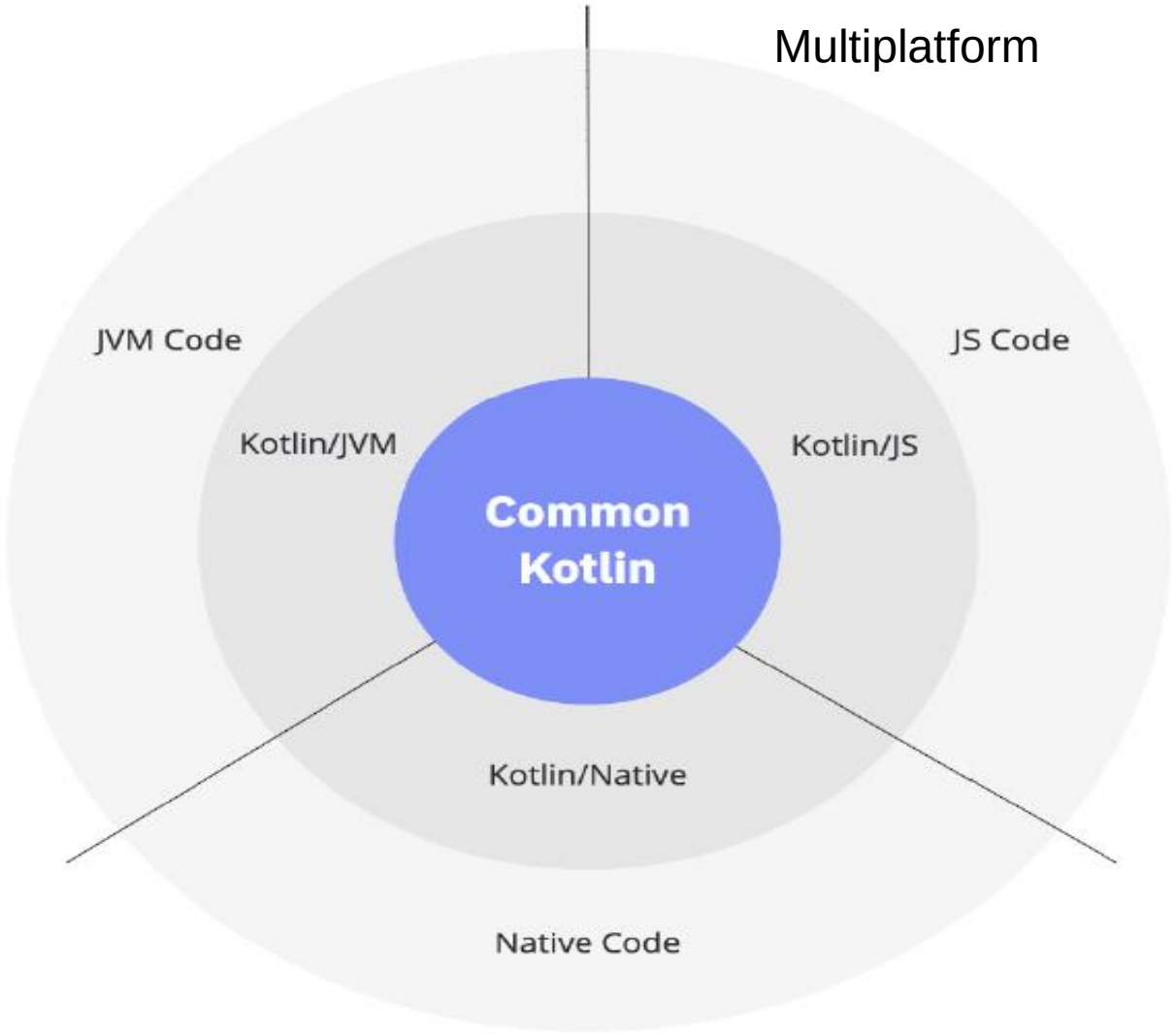
Kotlin

Kotlin Stands at 3rd Position



What’s Kotlin

Kotlin



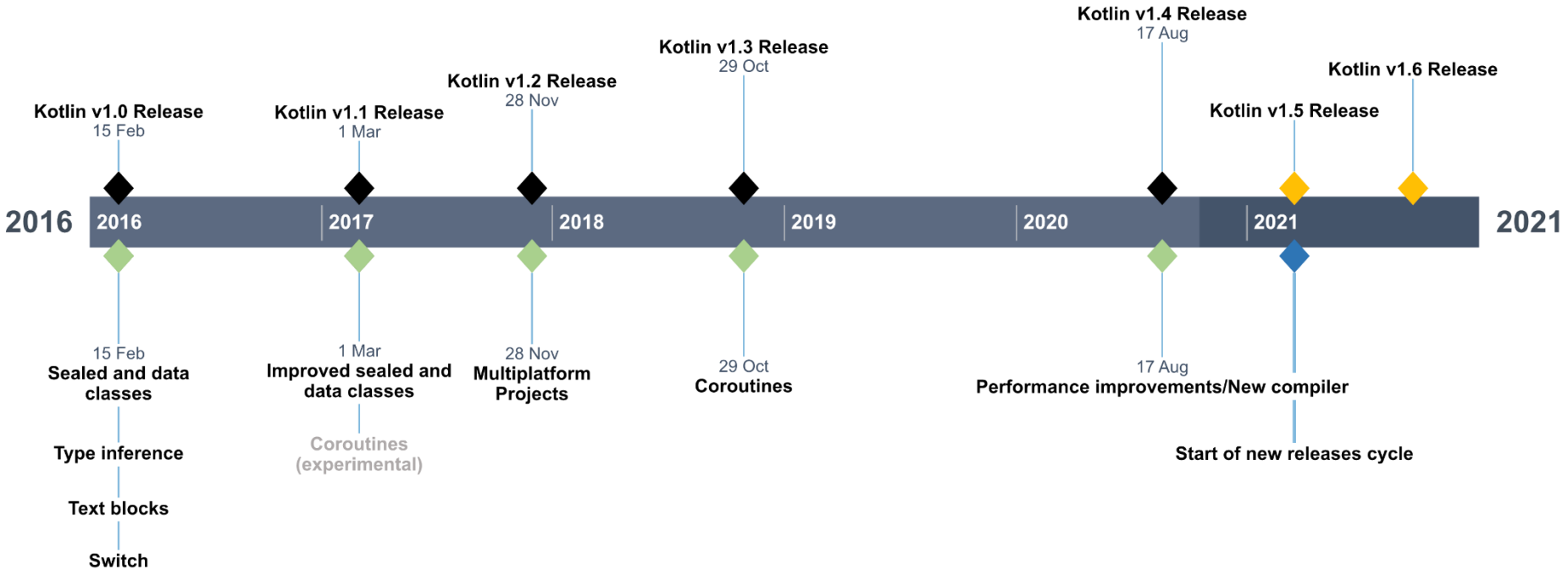
Kotlin History

Open source

Official language for
Android development

Kotlin History

Kotlin Release cycle



Kotlin History

1.7.20 Released: **September 29, 2022** Release on GitHub ↗	An incremental release with new language features, the support for several compiler plugins in the Kotlin K2 compiler, the new Kotlin/Native memory manager enabled by default, and the support for Gradle 7.1. Learn more in: • What's new in Kotlin 1.7.20 • What's new in Kotlin YouTube video ↗ • Compatibility guide for Kotlin 1.7.20 Learn more about Kotlin 1.7.20 ↗.
1.7.10 Released: **July 7, 2022** Release on GitHub ↗	A bug fix release for Kotlin 1.7.0. Learn more about Kotlin 1.7.0 ↗. ⓘ For Android Studio Dolphin (213) and Android Studio Electric Eel (221), the Kotlin plugin 1.7.10 will be delivered with upcoming Android Studios updates.
1.7.0 Released: **June 9, 2022** Release on GitHub ↗	A feature release with Kotlin K2 compiler in Alpha for JVM, stabilized language features, performance improvements, and evolutionary changes such as stabilizing experimental APIs. Learn more in: • What's new in Kotlin 1.7.0 • What's new in Kotlin YouTube video ↗ • Compatibility guide for Kotlin 1.7.0

Kotlin History

1.8.20 Released: **April 3, 2023** Release on GitHub ↗	A feature release with Kotlin K2 compiler updates, AutoCloseable interface and Base64 encoding in stdlib, new JVM incremental compilation enabled by default, new Kotlin/Wasm compiler backend. Learn more in: • What's new in Kotlin 1.8.20 • What's new in Kotlin YouTube video ↗
1.8.10 Released: **February 2, 2023** Release on GitHub ↗	A bug fix release for Kotlin 1.8.0. Learn more about Kotlin 1.8.0 ↗. ⓘ For Android Studio Electric Eel and Flamingo, the Kotlin plugin 1.8.10 will be delivered with upcoming Android Studios updates.
1.8.0 Released: **December 28, 2022** Release on GitHub ↗	A feature release with improved kotlin-reflect performance, new recursively copy or delete directory content experimental functions for JVM, improved Objective-C/Swift interoperability. Learn more in: • What's new in Kotlin 1.8.0 • Compatibility guide for Kotlin 1.8.0
1.7.21 Released: **November 9, 2022** Release on GitHub ↗	A bug fix release for Kotlin 1.7.20. Learn more about Kotlin 1.7.20 in What's new in Kotlin 1.7.20. ⓘ For Android Studio Dolphin, Electric Eel, and Flamingo, the Kotlin plugin 1.7.21 will be delivered with upcoming Android Studios updates.

Kotlin History

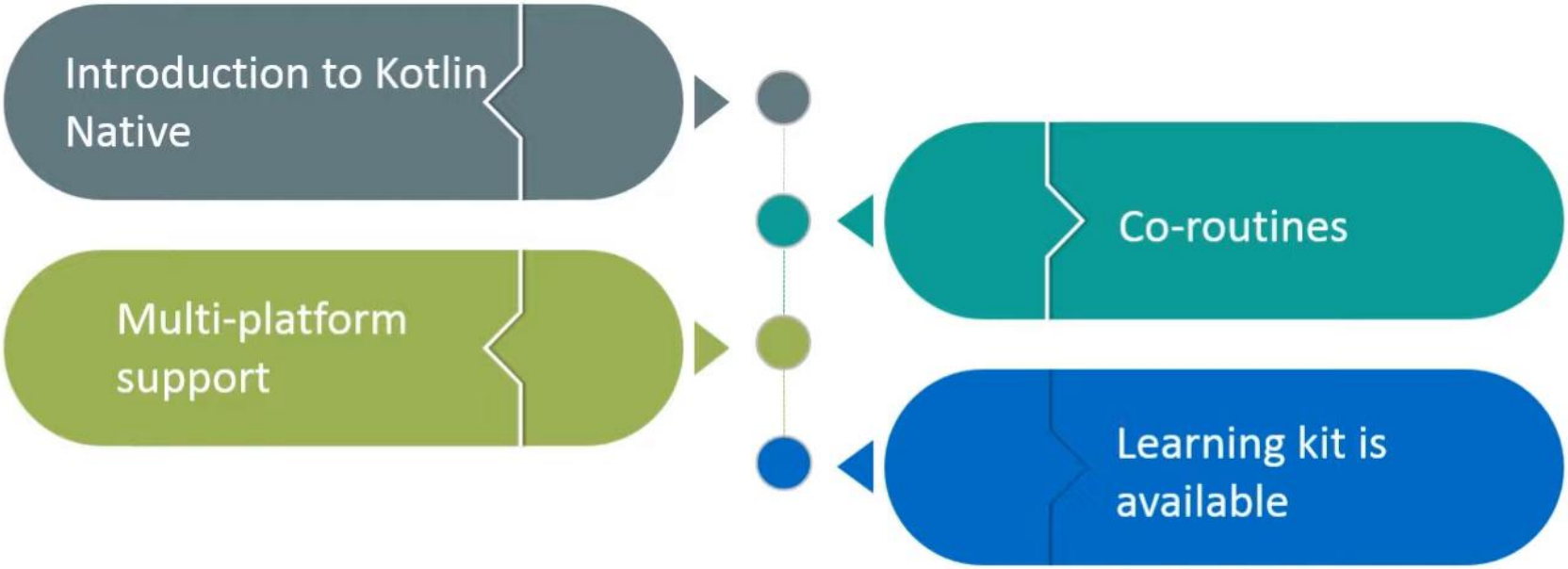
Kotlin

1.9.10 Released: **August 23, 2023** Release on GitHub ↗	A bug fix release for Kotlin 1.9.0. Learn more about Kotlin 1.9.0 in What's new in Kotlin 1.9.0. ⓘ For Android Studio Giraffe and Hedgehog, the Kotlin plugin 1.9.10 will be delivered with upcoming Android Studios updates.
1.9.0 Released: **July 6, 2023** Release on GitHub ↗	A feature release with Kotlin K2 compiler updates, new enum class values function, new operator for open-ended ranges, preview of Gradle configuration cache in Kotlin Multiplatform, changes to Android target support in Kotlin Multiplatform, preview of custom memory allocator in Kotlin/Native. Learn more in: • What's new in Kotlin 1.9.0 • What's new in Kotlin YouTube video ↗
1.8.22 Released: **June 8, 2023** Release on GitHub ↗	A bug fix release for Kotlin 1.8.20. Learn more about Kotlin 1.8.20 in What's new in Kotlin 1.8.20.
1.8.21 Released: **April 25, 2023** Release on GitHub ↗	A bug fix release for Kotlin 1.8.20. Learn more about Kotlin 1.8.20 in What's new in Kotlin 1.8.20. ⓘ For Android Studio Flamingo and Giraffe, the Kotlin plugin 1.8.21 will be delivered with upcoming Android Studios updates.

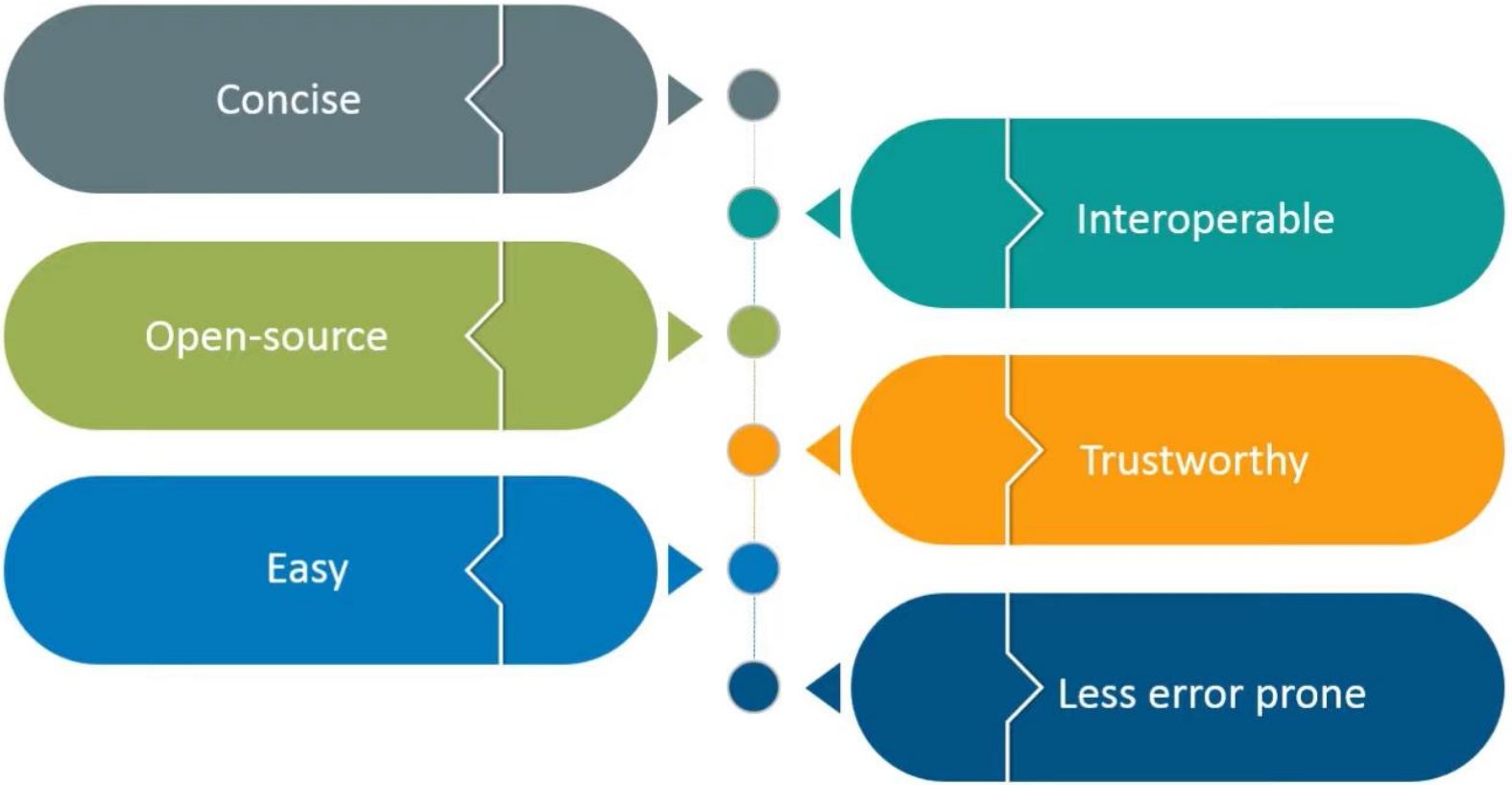
Kotlin History

1.9.21 Released: **November 23, 2023** Release on GitHub ↗	A bug fix release for Kotlin 1.9.20. Learn more about Kotlin 1.9.20 in What's new in Kotlin 1.9.20.
1.9.20 Released: **November 1, 2023** Release on GitHub ↗	A feature release with Kotlin K2 compiler in Beta and Stable Kotlin Multiplatform. Learn more in: • What's new in Kotlin 1.9.20,
1.9.10 Released: **August 23, 2023** Release on GitHub ↗	A bug fix release for Kotlin 1.9.0. Learn more about Kotlin 1.9.0 in What's new in Kotlin 1.9.0. For Android Studio Giraffe and Hedgehog, the Kotlin plugin 1.9.10 will be delivered with upcoming Android Studios updates.

Kotlin features

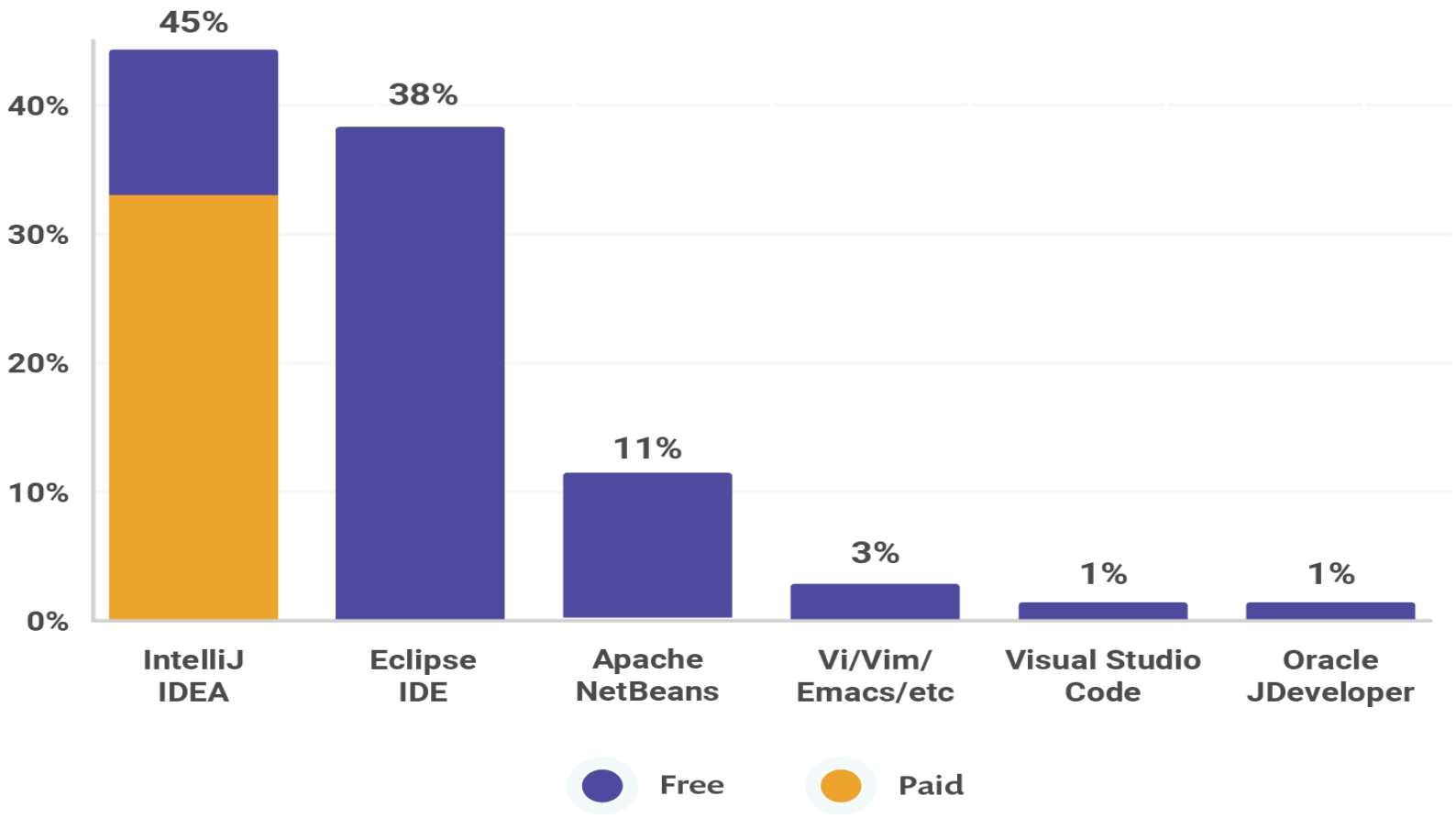


Kotlin features



IDEs

Kotlin


45%
38%
11%
3%
1%
1%
40%
30%
20%
10%
0%
IntelliJ IDEA
Eclipse IDE
Apache NetBeans
Vi/Vim/ Emacs/etc
Visual Studio Code
Oracle JDeveloper
Free
Paid

Intellij Ultimate

Kotlin

A code editor

The code editor offers code completion to help you write Kotlin code, and formatting and color highlighting to make your code easier to read. It also gives you hints for improving your code.

Build tools

You can compile and run your code using quick and easy shortcuts.

Kotlin REPL

You have easy access to the Kotlin REPL, which lets you try out code snippets outside your main code.

Version control

IntelliJ IDEA interfaces with major version control systems such as Git, SVN, CVS and more

Java vs Kotlin

Kotlin

Feature	Java	Kotlin
Check Exceptions	Available	Unavailable
Code Conciseness	Not very concise	Better than Java
Coroutines	Unavailable	Available
Data classes	Required to write a lot of boilerplate code	Requires adding only the data keyword in the class definition
Extension Functions	Unavailable	Available
Higher-Order Functions and Lambdas	Higher-order functions are implemented using Callables. Lambdas expressions are introduced in the Java 8	Prebuilt features
Implicit Widening Conversions	Unavailable	Available
Inline Functions	Unavailable	Available
Native Support for Delegation	Unavailable	Available
Non-private Fields	Available	Unavailable

Java vs Kotlin

Kotlin

NullPointerExceptions	Available	Unavailable
Primitive Types	Variables of a primitive type aren't objects	Variables of a primitive type are objects
Smart Casts	Unavailable	Available
Static Members	Available	Unavailable
Support for Constructors	No secondary constructors. Although, can have multiple constructors (constructor overloading)	Can have one or more secondary constructors
Ternary Operator	Available	Unavailable
Wildcard Types	Available	Unavailable, has declaration-site variance and type projects as an alternative

Why does compilation take so long?

Kotlin

- There are generally three big reasons for long compilation times:
  - Codebase size: compiling 1 MLOC usually takes longer than 1 KLOC.
  - How much your toolchain is optimized, this includes the compiler itself and any build tools you are using.
  - How smart your compiler is: whether it figures many things out without bothering the user with ceremony or constantly requires hints and boilerplate code.

Where Kotlin Stands

Kotlin

- Kotlin is designed to be used in an industrial setting where projects live long, grow big and involve a lot of people.
- So, we want static type safety to catch bugs early and get precise tooling (completion, refactorings and find usages in the IDE, precise code navigation and such).
- Then, we also want clean readable code without unneeded noise or ceremony.
- Among other things, this means that we don’t want types all over the code.
- That’s why we have smart type inference and overload resolution algorithms that support lambdas and extensions function types.
- We have smart casts (flow-based typing), and so on.
- The Kotlin compiler figures out a lot of stuff on its own to keep the code clean and type-safe at the same time.

Can One Be Smart and Fast

Kotlin

- To make a smart compiler run fast we certainly need to optimize every bit of the toolchain.
- This is something industry constantly working on.
- Kotlin team is working on a new-generation Kotlin compiler that will run much faster than the current one

Can One Be Smart and Fast

Kotlin

- There are two general approaches to reducing the amount of code to recompile:
- Compile avoidance — only recompile affected modules,
- Incremental compilation — only recompile affected files.

Dark Secrets of Kotlin Compiler

Kotlin

- Compile avoidance
  - Tracking ABI changes
  - Pros and cons of compile avoidance
- Incremental compilation
  - Compiling the dirty files
  - Widening the dirty file set
  - Incremental compilation across module boundaries

Compile Avoidance

Kotlin

- The core idea of compile avoidance is:
  - Find “dirty” (=changed) files
  - Recompile the modules these files belong to (use the results of the previous compilation of other modules as binary dependencies)
  - Determine which other modules may be affected by the changes
  - Recompile those as well, check their ABIs too
  - Repeat until all affected modules are recompiled

Tracking ABI changes

Kotlin

- ABI stands for Application Binary Interface, and it’s kind of the same as API but for the binaries.
- Essentially, the ABI is the only part of the binary that dependent modules care about.
- A Kotlin binary (be it a JVM class file or a KLib) contains declarations and bodies.
- Other modules can reference declarations, but not all declarations.
- So, private classes and members, for example, are not part of the ABI.

Tracking ABI changes

Kotlin

- A straightforward way to detect a change in the ABI is
- Store the ABI from the previous compilation in some form (you might want to store hashes for efficiency),
- After compiling a module, compare the result with the stored ABI:
  - If it’s the same, we are done;
  - If it’s changed, recompile dependent modules

Incremental compilation

Kotlin

- Incremental compilation tries to approximate the set of affected files by going in multiple rounds, here's the outline of how it's done:
  - Find "dirty" (=changed) files
  - Recompile them (use the results of the previous compilation as binary dependencies instead of compiling other source files)
  - Check if the ABI corresponding to these files has changed
    - If not, we are done!
    - If yes, find files affected by the changes, add them to the dirty files set, recompile
    - Repeat until the ABI stabilizes (this is called a "fixpoint")

Compiling the dirty files

Kotlin

- The compiler knows how to use a subset of the previous compilation results to skip compiling non-dirty files.
- Just load the symbols defined in them to produce the binaries for the dirty files.
- This is not something a compiler would necessarily be able to do if not for incrementality:
- Producing one big binary from a module instead of a small binary per source file is not so common outside of the JVM world.
- And it’s not a feature of the Kotlin language, it’s an implementation detail of the incremental compiler.

Compiling the dirty files

Kotlin

Consider this example:

File: dirty.kt

```
// rename this to be 'fun foo(i: Int)'
fun changeMe(i: Int) = if (i == 1) 0 else bar().length
```

File: clean.kt

```
fun foo(a: Any) = ""
fun bar() =  foo(1)
```

Compiling the dirty files

Kotlin

- Let’s assume that the user renamed the function changeMe to foo.
- Note that, although clean.kt is not changed, the body of bar() will change upon recompilation:
- It will now be calling foo(Int) from dirty.kt, not foo(Any) from clean.kt, and its return type will change too.
- This means that we have to recompile both dirty.kt and clean.kt.
- How can the incremental compiler find this out?

Compiling the dirty files

Kotlin

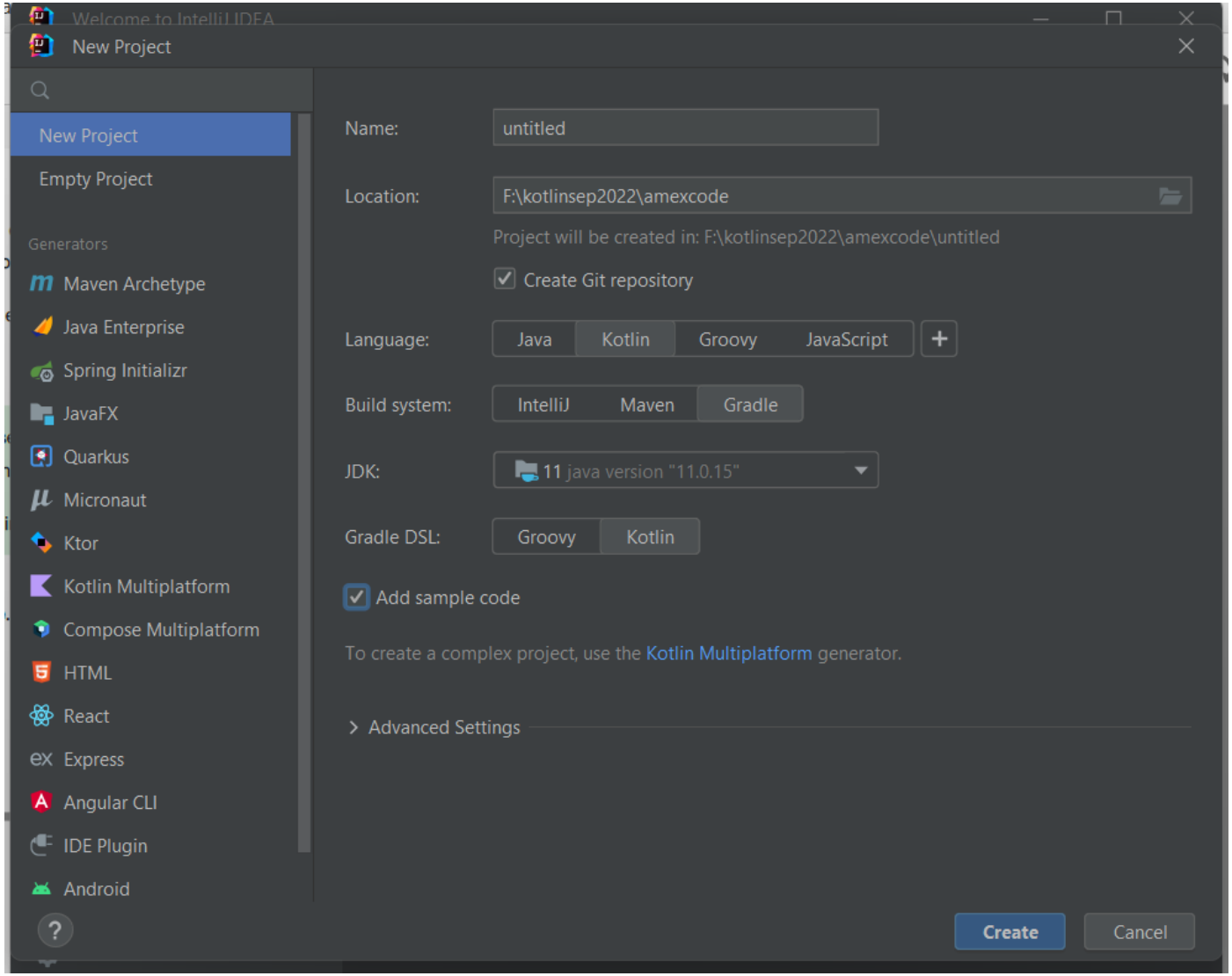
- We start by recompiling the changed file: dirty.kt.
- Then we see that something in the ABI has changed:
  - there’s no function changeMe any more,
  - there’s function foo that takes an Int and returns an Int.

Incremental compilation

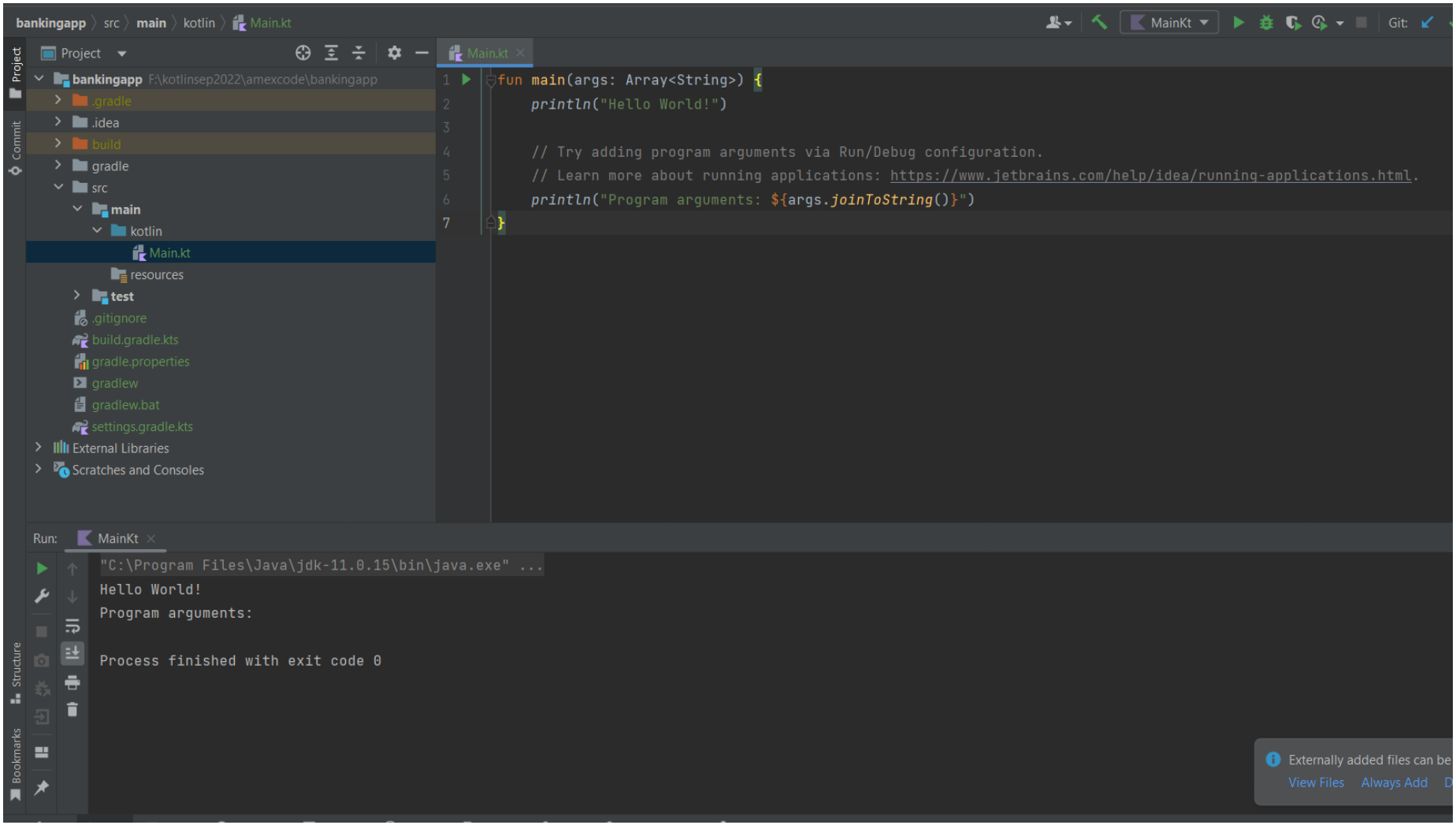
Kotlin

- Incremental compilation is more granular than compile avoidance:
- It works on individual files rather than modules.
- It does not care about module sizes nor does it recompile the whole project when an ABI of a “popular” module is changed insignificantly.
- In general, this approach does not restrict the user as much and leads to shorter time-to-test.

First Application



First Application

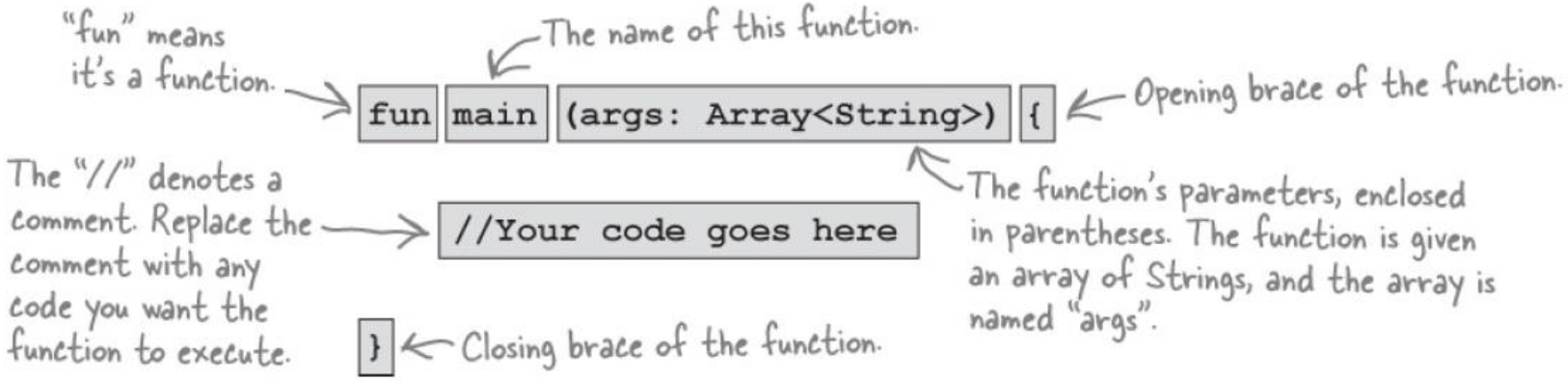


First Application

```
import org.jetbrains.kotlin.gradle.tasks.KotlinCompile

plugins { this: PluginDependenciesSpecScope
    kotlin("jvm") version "1.6.21"
    application
}

group = "com.amex"
version = "1.0-SNAPSHOT"

repositories { this: RepositoryHandler
    mavenCentral()
}

dependencies { this: DependencyHandlerScope
    testImplementation(kotlin("test"))
}

tasks.test { this: Test!
    useJUnitPlatform()
}

tasks.withType<KotlinCompile> { this: KotlinCompile
    kotlinOptions.jvmTarget = "1.8"
}

application { this: JavaApplication
    mainClass.set("MainKt")
}
```

First Application

The `main` function looks like this:



First Application

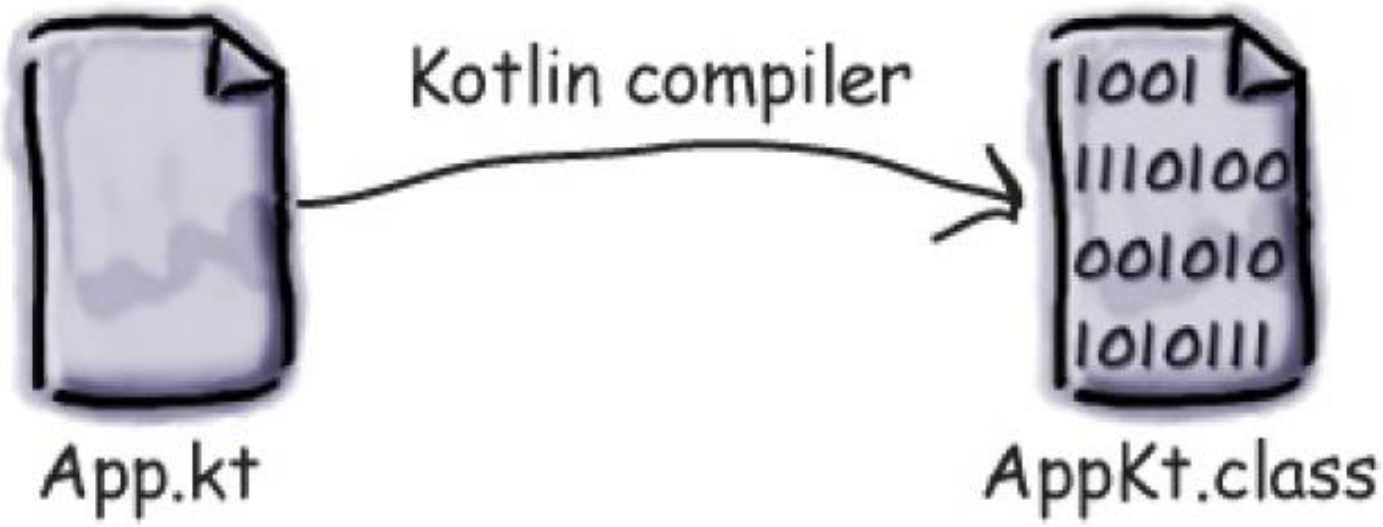
Kotlin

PARAMETERLESS MAIN FUNCTIONS

From Kotlin1.3, however, you can omit `main`'s parameters so that the function looks like this:

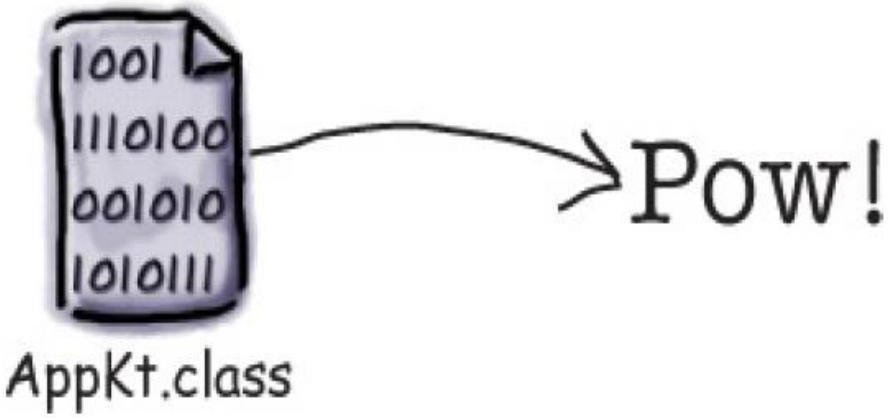
```
fun main() {

    //Your code goes here

}
```

The IDE compiles your Kotlin source code into JVM bytecode



The IDE starts the JVM and runs AppKt.class

The JVM translates the *AppKt.class* bytecode into something the underlying platform understands, then runs it. This displays the `String` "Pow!" in the IDE's output window.



Kotlin Multiplatform

Kotlin

- Support for multiplatform programming is one of Kotlin's key benefits.
- It reduces time spent writing and maintaining the same code for different platforms while retaining the flexibility and benefits of native programming.

Kotlin Multiplatform use cases

Kotlin

- Android and iOS applications
  - Sharing code between mobile platforms is one of the major Kotlin Multiplatform use cases.
  - With Kotlin Multiplatform Mobile, we can build cross-platform mobile applications and share common code between Android and iOS, such as business logic, connectivity, and more.

Kotlin Multiplatform use cases

Kotlin

- Kotlin Multiplatform is also useful for library authors.
- We can create a multiplatform library with common code and its platform-specific implementations for JVM, JS, and Native platforms.
- Once published, a multiplatform library can be used in other cross-platform projects as a dependency.

Kotlin Multiplatform use cases

Kotlin

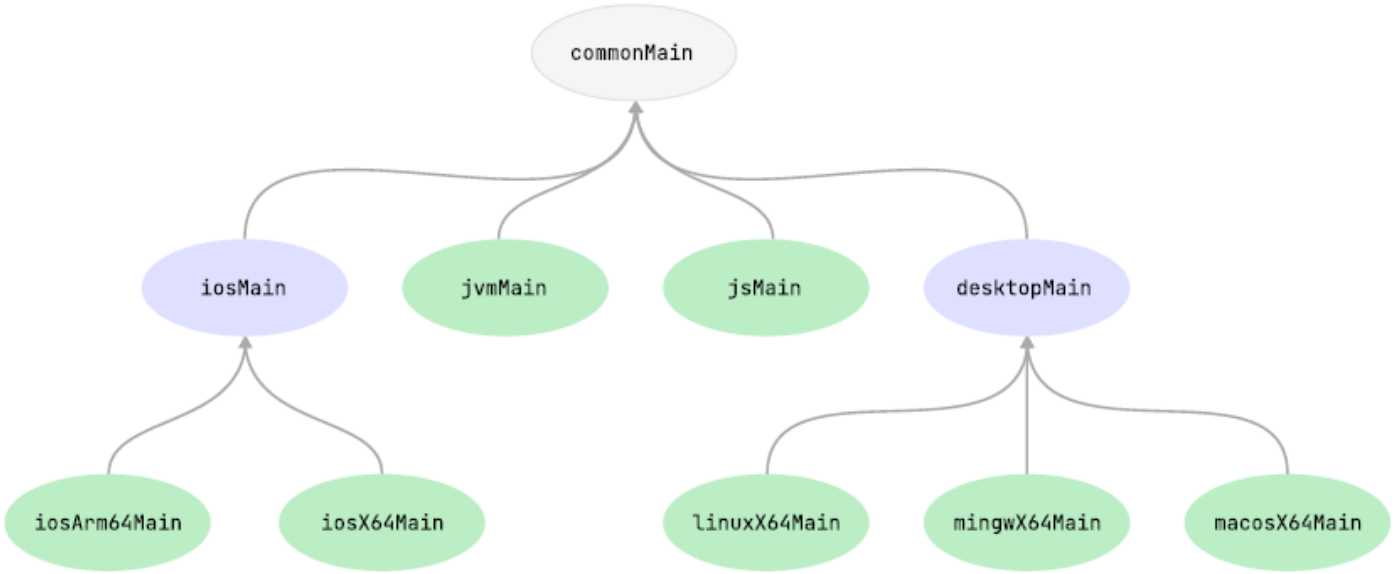
- Common code for mobile and web applications.
- One more popular case for using Kotlin Multiplatform is sharing the same code across Android, iOS, and web apps.
- It reduces the amount of business logic coded by frontend developers.
- It helps implement products more efficiently, decreasing the coding and testing efforts.

Kotlin Multiplatform use cases

Kotlin

- With Kotlin Multiplatform, spend less time on writing and maintaining the same code for different platforms – just share it using the mechanisms Kotlin provides:
- Share code among all platforms used in your project.
- Use it for sharing the common business logic that applies to all platforms.
- Share code among some platforms included in our project but not all.
- Do this when you can reuse much of the code in similar platforms.

Kotlin Multiplatform use cases



Kotlin Multiplatform use cases

Kotlin

- Netflix: The popular OTT platform Netflix has rebuilt the current UI player in the Netflix Android application using Kotlin.
- Autodesk: Autodesk is a multinational software corporation.
- Kotlin works well with iOS and android without any Java native interface, which is a major reason for the company to opt for Kotlin multiplatform.

Kotlin Multiplatform use cases

Kotlin

- Vmware: Vmware is an American cloud computing and virtualization technology company.
- The Vmware development team wanted a cross-platform strategy for mobile applications, then the Kotlin multiplatform was selected, and now it is the “default, go-to framework of choice” for VMware.
- Trello: Trello is a project management application. The android application of Trello is built using Kotlin. Trello uses Kotlin because of its interoperability with Java.

Kotlin Multiplatform use cases

Kotlin

- Philips: Philips is a very popular multinational corporation, and surely you have heard about this company.
- In Philips, Kotlin Multiplatform proved to be faster at implementing new features and to get more interaction in the team between Android and iOS developers.

Kotlin Multiplatform use cases

Kotlin

- Ice Rock: Ice Rock, an outsourcing company that provides mobile and backend solutions for various industries and market segments, uses Kotlin multiplatform mobile to create applications for its clients.

Kotlin for Server Side

Kotlin

- Kotlin is a great fit for developing server-side applications.
- It allows you to write concise and expressive code while maintaining full compatibility with existing Java-based technology stacks, all with a smooth learning curve:
- Expressiveness: Kotlin's innovative language features, such as its support for type-safe builders and delegated properties, help build powerful and easy-to-use abstractions.
- Scalability: Kotlin's support for coroutines helps build server-side applications that scale to massive numbers of clients with modest hardware requirements.

Kotlin for Server Side

Kotlin

- Interoperability: Kotlin is fully compatible with all Java-based frameworks.
- So, we can use our familiar technology stack while reaping the benefits of a more modern language.
- Migration: Kotlin supports gradual migration of large codebases from Java to Kotlin.
- We can start writing new code in Kotlin while keeping older parts of your system in Java.
- Tooling: In addition to great IDE support in general, Kotlin offers framework-specific tooling (for example, for Spring) in the plugin for IntelliJ IDEA Ultimate.

Kotlin for Server Side

Kotlin

- Learning Curve: For a Java developer, getting started with Kotlin is very easy.
- The automated Java-to-Kotlin converter included in the Kotlin plugin helps with the first steps.
- Kotlin Koans can guide you through the key features of the language with a series of interactive exercises.

Frameworks for server-side development with Kotlin

Kotlin

- **Spring** makes use of Kotlin's language features to offer more concise APIs, starting with version 5.0. The online project generator allows you to quickly generate a new project in Kotlin.
- **Vert.x**, a framework for building reactive Web applications on the JVM, offers dedicated support for Kotlin, including full documentation.
- **Ktor** is a framework built by JetBrains for creating Web applications in Kotlin, making use of coroutines for high scalability and offering an easy-to-use and idiomatic API.
- **kotlinx.html** is a DSL that can be used to build HTML in Web applications. It serves as an alternative to traditional templating systems such as JSP and FreeMarker.

Frameworks for server-side development with Kotlin

- Micronaut is a modern JVM-based full-stack framework for building modular, easily testable microservices and serverless applications.
- It comes with a lot of useful built-in features.
- http4k is the functional toolkit with a tiny footprint for Kotlin HTTP applications, written in pure Kotlin. The library is based on the "Your Server as a Function" paper from Twitter and represents modeling both HTTP servers and clients as simple Kotlin functions that can be composed together.
- Javalin is a very lightweight web framework for Kotlin and Java which supports WebSockets, HTTP2, and async requests.

Kotlin for Android

Kotlin

- Less code combined with greater readability. Spend less time writing our code and working to understand the code of others.
- Mature language and environment. Since its creation in 2011, Kotlin has developed continuously, not only as a language but as a whole ecosystem with robust tooling.
- Now it's seamlessly integrated in Android Studio and is actively used by many companies for developing Android applications.
- Kotlin support in Android Jetpack and other libraries.
- KTX extensions add Kotlin language features, such as coroutines, extension functions, lambdas, and named parameters, to existing Android libraries.
- Interoperability with Java. You can use Kotlin along with the Java programming language in your applications without needing to migrate all your code to Kotlin.

Kotlin for Android

Kotlin

- Support for multiplatform development.
- We can use Kotlin for developing not only Android but also iOS, backend, and web applications.
- Enjoy the benefits of sharing the common code among the platforms.
- Code safety.
- Less code and better readability lead to fewer errors.
- The Kotlin compiler detects these remaining errors, making the code safe.

Kotlin for Android

Kotlin

- Easy learning.
- Kotlin is very easy to learn, especially for Java developers.
- Big community. Kotlin has great support and many contributions from the community, which is growing all over the world.
- According to Google, over 60% of the top 1000 apps on the Play Store use Kotlin.
- Many startups and Fortune 500 companies have already developed Android applications using Kotlin

Kotlin for JavaScript

Kotlin

- Kotlin/JS provides the ability to transpile your Kotlin code, the Kotlin standard library, and any compatible dependencies to JavaScript.
- The current implementation of Kotlin/JS targets ES5.

Use cases for Kotlin/JS

Kotlin

- Write frontend web applications using Kotlin/JS
  - Kotlin/JS allows us to leverage powerful browser and web APIs in a type-safe fashion.
  - Create, modify, and interact with the elements in the Document Object Model (DOM), use Kotlin code to control the rendering of canvas or WebGL components, and enjoy access to many more features that modern browsers support.
  - Write full, type-safe React applications with Kotlin/JS using the kotlin-wrappers provided by JetBrains, which provide convenient abstractions and deep integrations for React and other popular JavaScript frameworks.

Use cases for Kotlin/JS

Kotlin

- kotlin-wrappers also provides support for a select number of adjacent technologies, like react-redux, react-router, and styled-components.
- Interoperability with the JavaScript ecosystem means that you can also use third-party React components and component libraries.
- Use the Kotlin/JS frameworks, which take full advantage of Kotlin concepts and its expressive power and conciseness.

Kotlin Native

Kotlin

- Kotlin/Native is primarily designed to allow compilation for platforms on which virtual machines are not desirable or possible, such as embedded devices or iOS.
- It is ideal for situations when a developer needs to produce a self-contained program that does not require an additional runtime or virtual machine.

Kotlin Native

- Kotlin/Native supports the following platforms:
  - macOS
  - iOS, tvOS, watchOS
  - Linux
  - Windows (MinGW)
  - Android NDK

Interoperability

Kotlin

- Kotlin/Native supports two-way interoperability with native programming languages for different operating systems.
- The compiler creates:
  - an executable for many platforms
  - a static library or dynamic library with C headers for C/C++ projects
  - an Apple framework for Swift and Objective-C projects

Interoperability

Kotlin

- Kotlin/Native supports interoperability to use existing libraries directly from Kotlin/Native:
  - static or dynamic C libraries
  - C, Swift, and Objective-C frameworks

Kotlin for data science

Kotlin

- From building data pipelines to productionizing machine learning models, Kotlin can be a great choice for working with data:
  - Kotlin is concise, readable, and easy to learn.
  - Static typing and null safety help create reliable, maintainable code that is easy to troubleshoot.
  - Being a JVM language, Kotlin gives you great performance and an ability to leverage an entire ecosystem of tried-and-true Java libraries.

Kotlin for data science

Kotlin

- Interactive editors
  - Notebooks such as Jupyter Notebook, Datalore, and Apache Zeppelin provide convenient tools for data visualization and exploratory research.
  - Kotlin integrates with these tools to help you explore data, share your findings with colleagues, or build up your data science and machine learning skills.
    - Jupyter
    - Kotlin
    - kernel

Kotlin for data science

Kotlin

- The Jupyter Notebook is an open-source web application that allows you to create and share documents (aka "notebooks") that can contain code, visualizations, and Markdown text.
- Kotlin-jupyter is an open-source project that brings Kotlin support to Jupyter Notebook.

Kotlin for data science

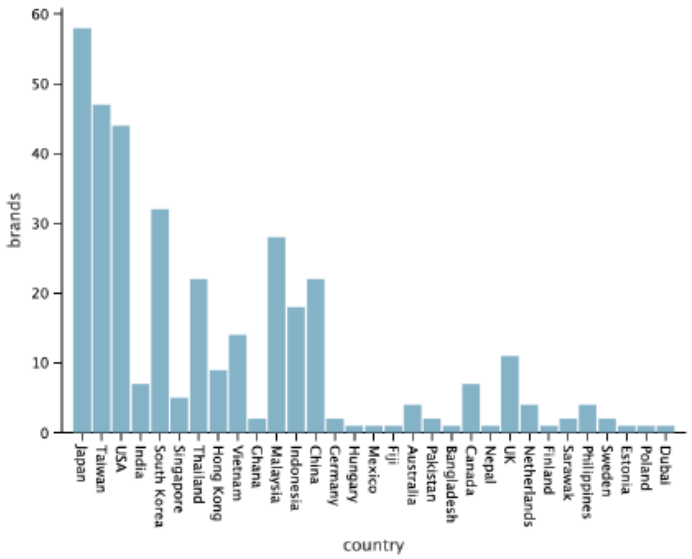
Kotlin

In [1]:
```
%use lets-plot, krangl
```

In [2]:
```
val df = DataFrame.readCSV("ramen-ratings.csv")
val processedDF = df.filter({ it["Stars"].isMatching <String>{ !startsWith("Un")  }})
    .addColumn("StarsAsDouble") { it["Stars"].map <String> { it.toDouble()}}
```

In [3]:
```
val distinctBrandsPerCountry = processedDF.groupBy("Country").distinct("Brand").groupBy("Country").count()
val (xs, ys) = distinctBrandsPerCountry.rows.map { row -> (row["Country"] as String) to (row["n"] as Int) }.unzip()
val p = lets_plot(mapOf("country" to xs, "brands" to ys))

val layer = geom_bar (stat=Stat.identity, fill = "#78B3CA") {
    x = "country"
    y = "brands"
}

p + layer
```

Out[3]:



Kotlin Lib For Data Science

Kotlin

- Multik: multidimensional arrays in Kotlin.
- KotlinDL is a high-level Deep Learning API written in Kotlin and inspired by Keras.
- Kotlin DataFrame is a library for structured data processing.
- Kotlin for Apache Spark adds a missing layer of compatibility between Kotlin and Apache Spark.
- kotlin-statistics is a library providing extension functions for exploratory and production statistics.

Kotlin Lib For Data Science

Kotlin

- kmath is an experimental library that was intially inspired by NumPy but evolved to more flexible abstractions.
- krangl is a library inspired by R's dplyr and Python's pandas.
- lets-plot is a plotting library for statistical data written in Kotlin.
- kravis is another library for the visualization of tabular data inspired by R's ggplot.
- londogard-nlp-toolkit is a library that provides utilities when working with natural language processing such as word/subword/sentence embeddings, word-frequencies, stopwords, stemming, and much more.

Kotlin Lib For Data Science

Kotlin

- Java libraries
- Since Kotlin provides first-class interop with Java, you can also use Java libraries for data science in your Kotlin code. Here are some
- examples of such libraries:
  - DeepLearning4J - a deep learning library for Java
  - ND4J - an efficient matrix math library for JVM
  - Dex - a Java-based data visualization tool
  - Smile - a comprehensive machine learning, natural language processing, linear algebra, graph, interpolation, and visualization system.
  - Besides Java API, Smile also provides a functional Kotlin API along with Scala and Clojure API.

Kotlin Lib For Data Science

Kotlin

- Apache Commons Math - a general math, statistics, and machine learning library for Java
- NM Dev - a Java mathematical library that covers all of classical mathematics.
- OptaPlanner - a solver utility for optimization planning problems
- Charts - a scientific JavaFX charting library in development
- Apache OpenNLP - a machine learning based toolkit for the processing of natural language text
- CoreNLP - a natural language processing toolkit
- Apache Mahout - a distributed framework for regression, clustering and recommendation
- Weka - a collection of machine learning algorithms for data mining tasks
- Tablesaw - a Java dataframe. It includes a visualization library based on Plot.ly

Kotlin for competitive programming

Kotlin

- Competitive programming is a mind sport where contestants write programs to solve precisely specified algorithmic problems within strict constraints.
- Problems can range from simple ones that can be solved by any software developer.
- It requires little code to get a correct solution, to complex ones that require knowledge of special algorithms, data structures, and a lot of practice.

Kotlin for competitive programming

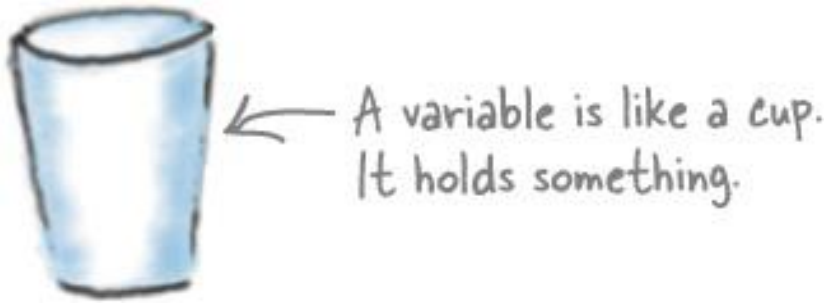
Kotlin

- Even though Kotlin not designed for competitive programming, Kotlin incidentally fits well in this domain.
- It reduces the typical amount of boilerplate that a programmer needs to write and read while working with the code almost to the level offered by dynamically-typed scripting languages, while having tooling and performance of a statically-typed language.

Basic types and variables: Being a Variable

Variable

- When we think of a variable in Kotlin, think of a cup.
- Cups come in many different shapes and sizes—big cups, small cups, the giant disposable cups that popcorn comes in at the movies.
- But they all have one thing in common: a cup holds something.



What happens when you declare a variable

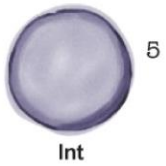
Kotlin

- To create a variable, the compiler needs to know its name, type and whether it can be reused.
- The value is transformed into an object...
  - When we declare a variable using code like:
  - var x = 5
- The value we are assigning to the variable is used to create a new object.
- In this example, you're assigning the number 5 to a new variable named x.
- The compiler knows that 5 is an integer, and so the code creates a new Int object with a value of 5:

What happens when you declare a variable

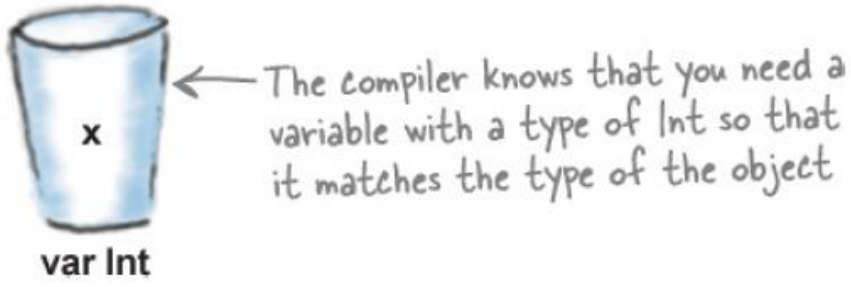
Kotlin

- The compiler really cares about a variable's type so that it can prevent dangerous operations that might lead to bugs.
- It won't let us assign the String "Fish" to an integer variable, for example, because it knows that it's inappropriate to perform mathematical operations on a String.
- For this type-safety to work, the compiler needs to know the type of the variable.
- And the compiler can infer the variable's type from the value that's assigned to it.



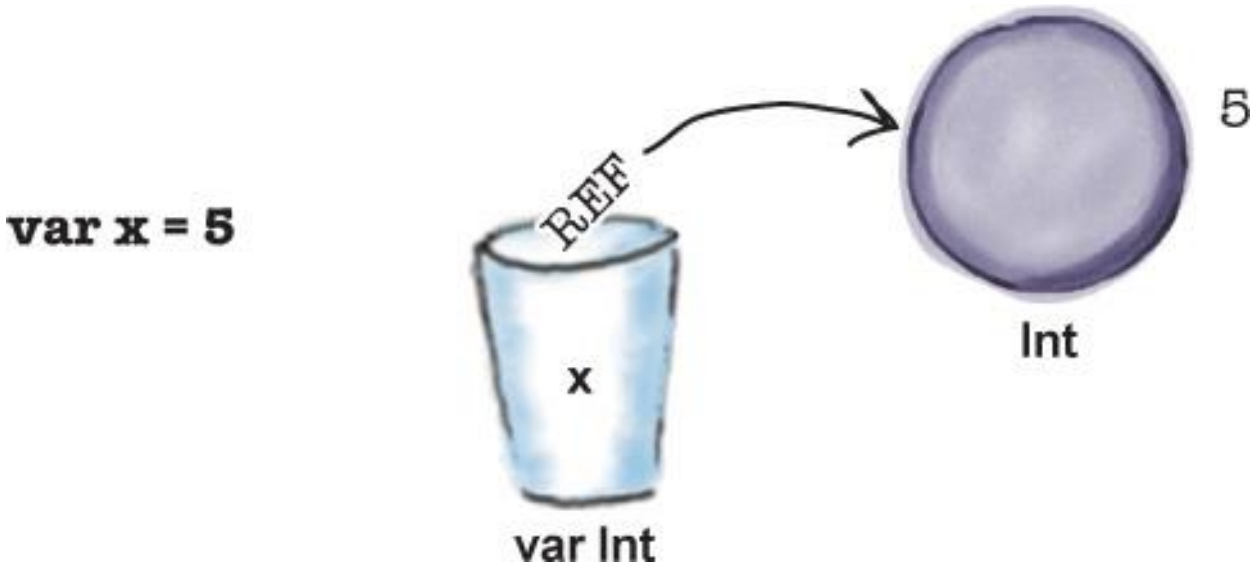
What happens when you declare a variable

- the compiler infers the variable's type from that of the object
- The compiler then uses the type of the object for the type of the variable.
- In the above example, the object's type is Int, so the variable's type is Int as well.
- The variable stays this type forever.



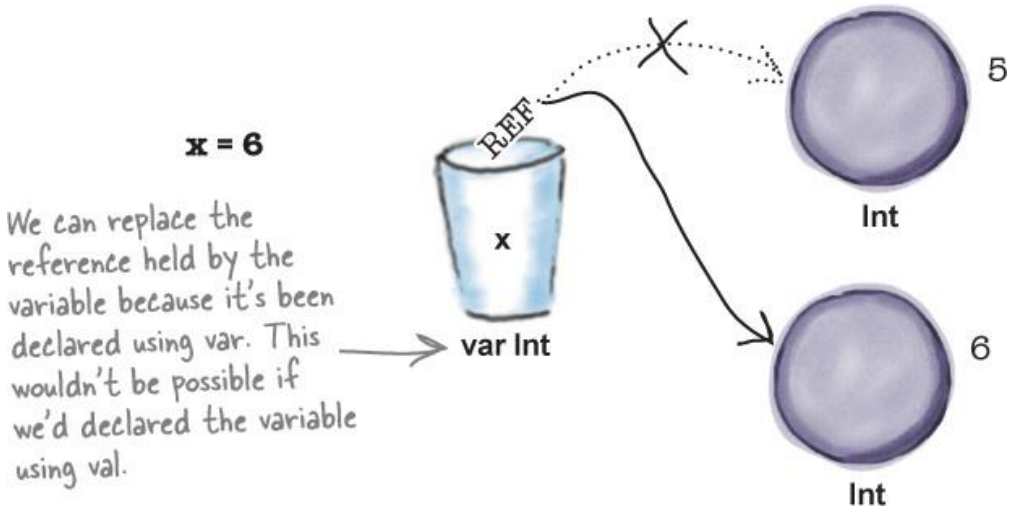
The variable holds a reference to the object

- When an object is assigned to a variable, the object itself doesn't go into the variable.
- A reference to the object goes into the variable instead.

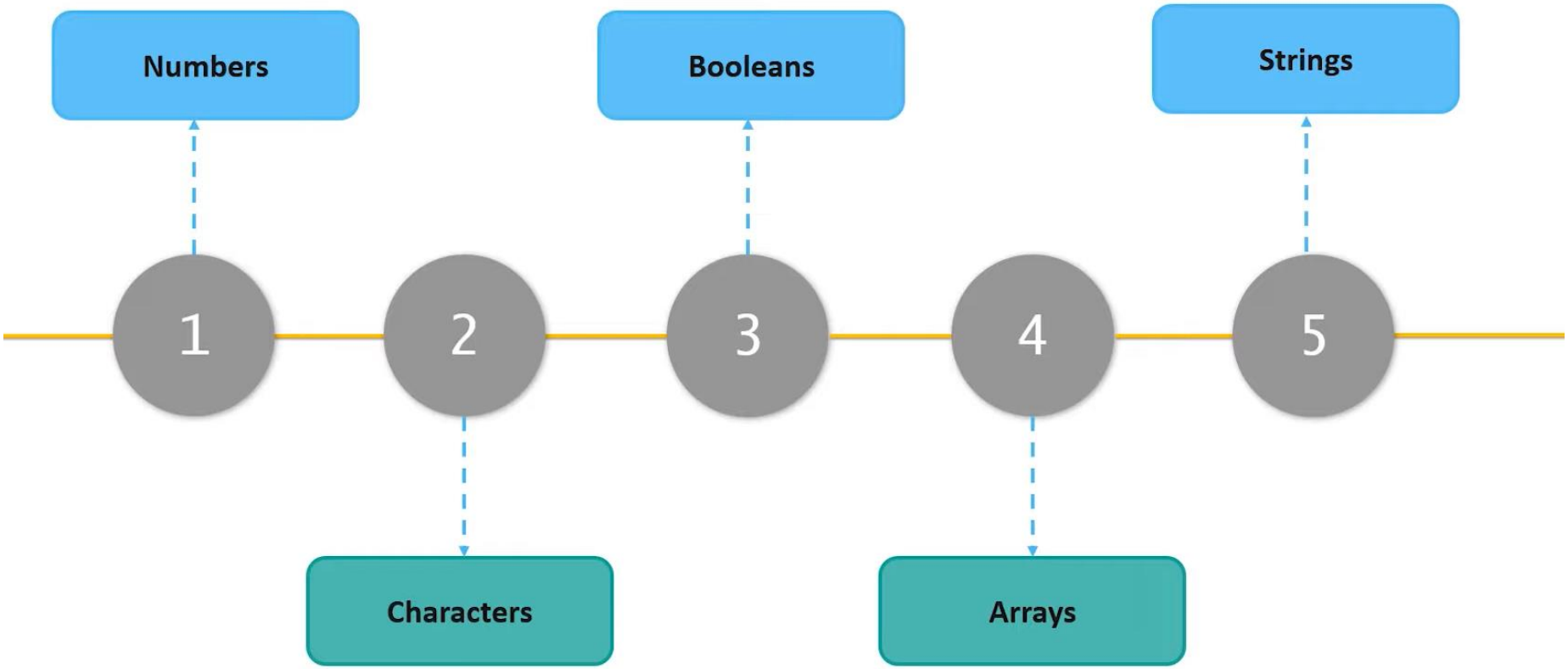


val vs. var revisited

- If we declare the variable using val, the reference to the object stays in the variable forever and can't be replaced.
- But if we use the var keyword instead, we can assign another value to the variable.
  - x = 6
- to assign a value of 6 to x, this creates a new Int object with a value of 6, and puts a reference to it into x. This replaces the original reference:



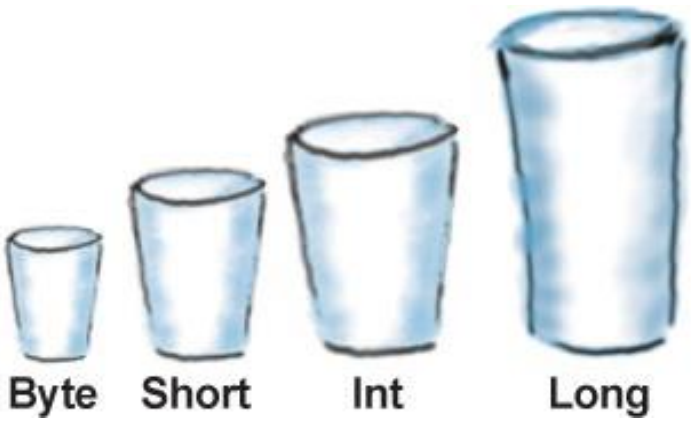
Data Types



Data Types

Kotlin

- Numbers
  - Integer
  - Floating
- Characters
- Boolean
- Arrays
- Strings



Byte	8 bits	-128 to 127
Short	16 bits	-32768 to 32767
Int	32 bits	-2147483648 to 2147483647
Long	64 bits	-huge to (huge - 1)

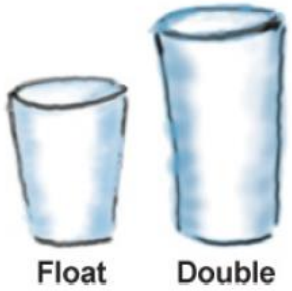
Byte 8 bit
Short 16 bit
Int 32 bit
Long 64 bit
Float 32 bit
Double 64 bit

Data Types

Kotlin

- Numbers
  - Integer
  - Floating
- Characters
- Boolean
- Arrays
- Strings

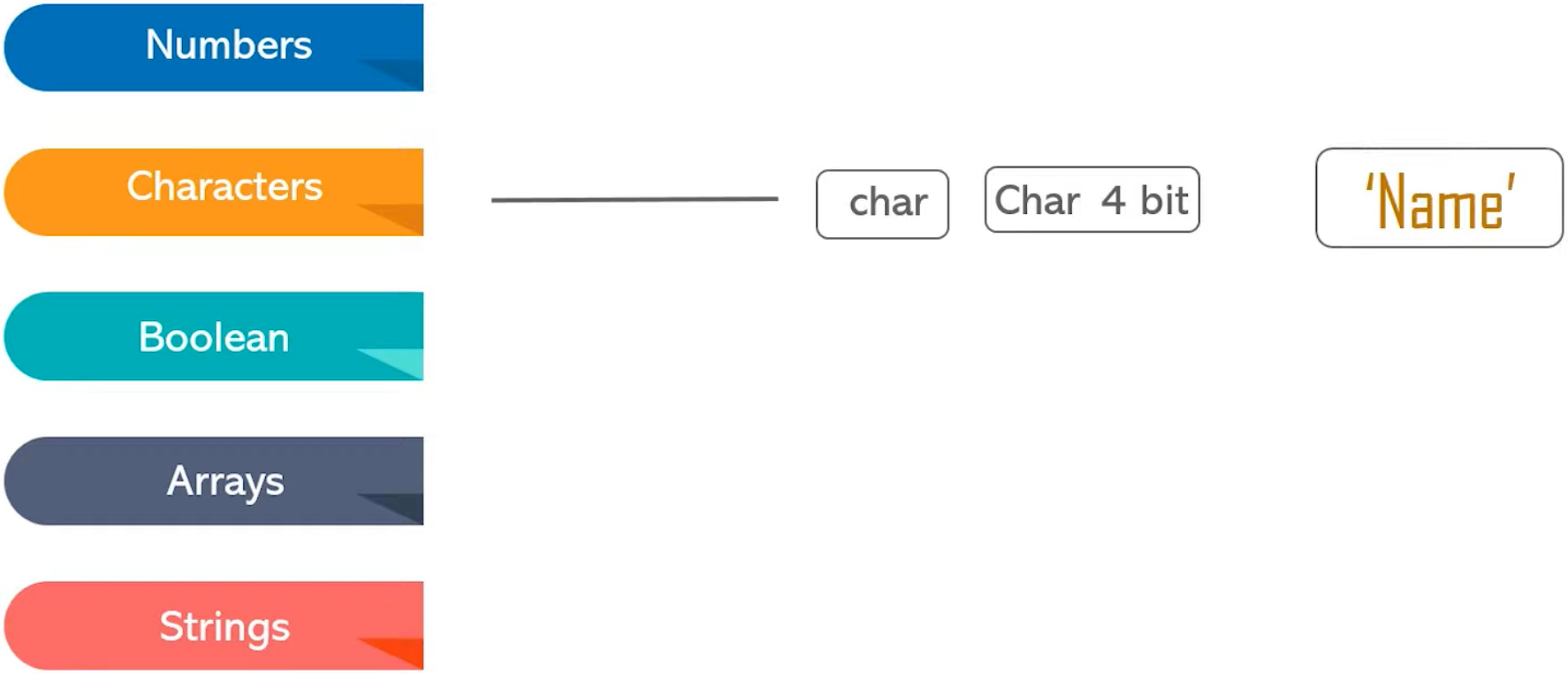
Byte 8 bit
Short 16 bit
Int 32 bit
Long 64 bit
Float 32 bit
Double 64 bit



We will create an object and variable of type Double. If we add an “F” or “f” to the end of the number, a Float will get created instead:
var x = 123.5F

Data Types

Kotlin



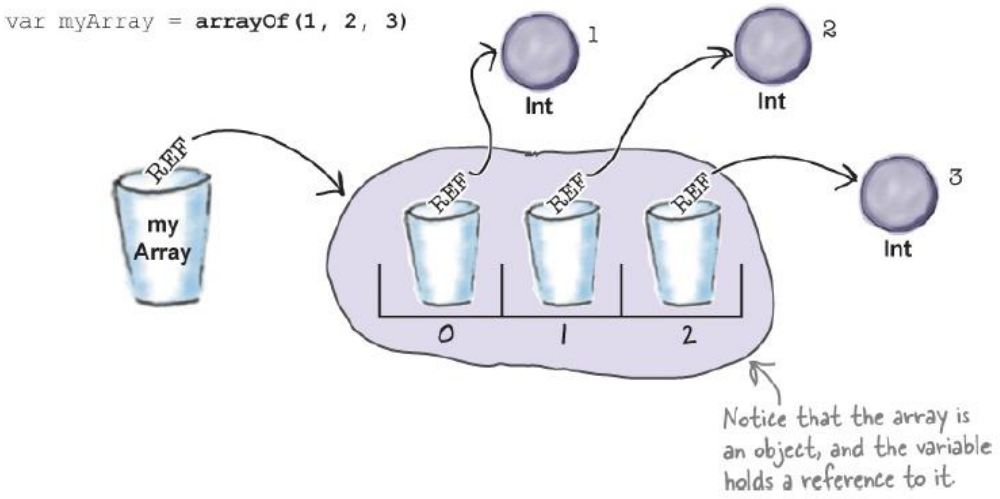
Data Types


Numbers
Characters
Boolean
Arrays
Strings
Boolean True or False
1 bit

Data Types

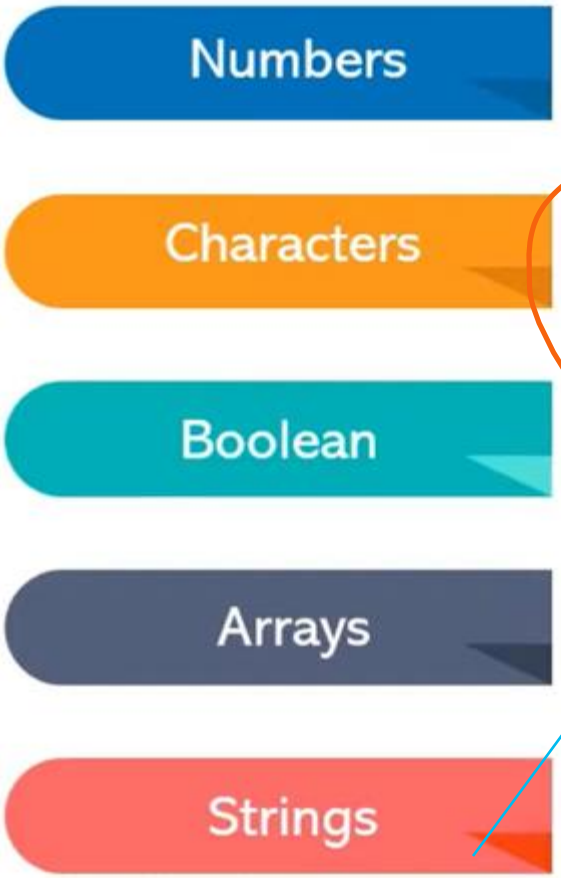
Kotlin

- Numbers
- Characters
- Boolean
- Arrays
- Strings



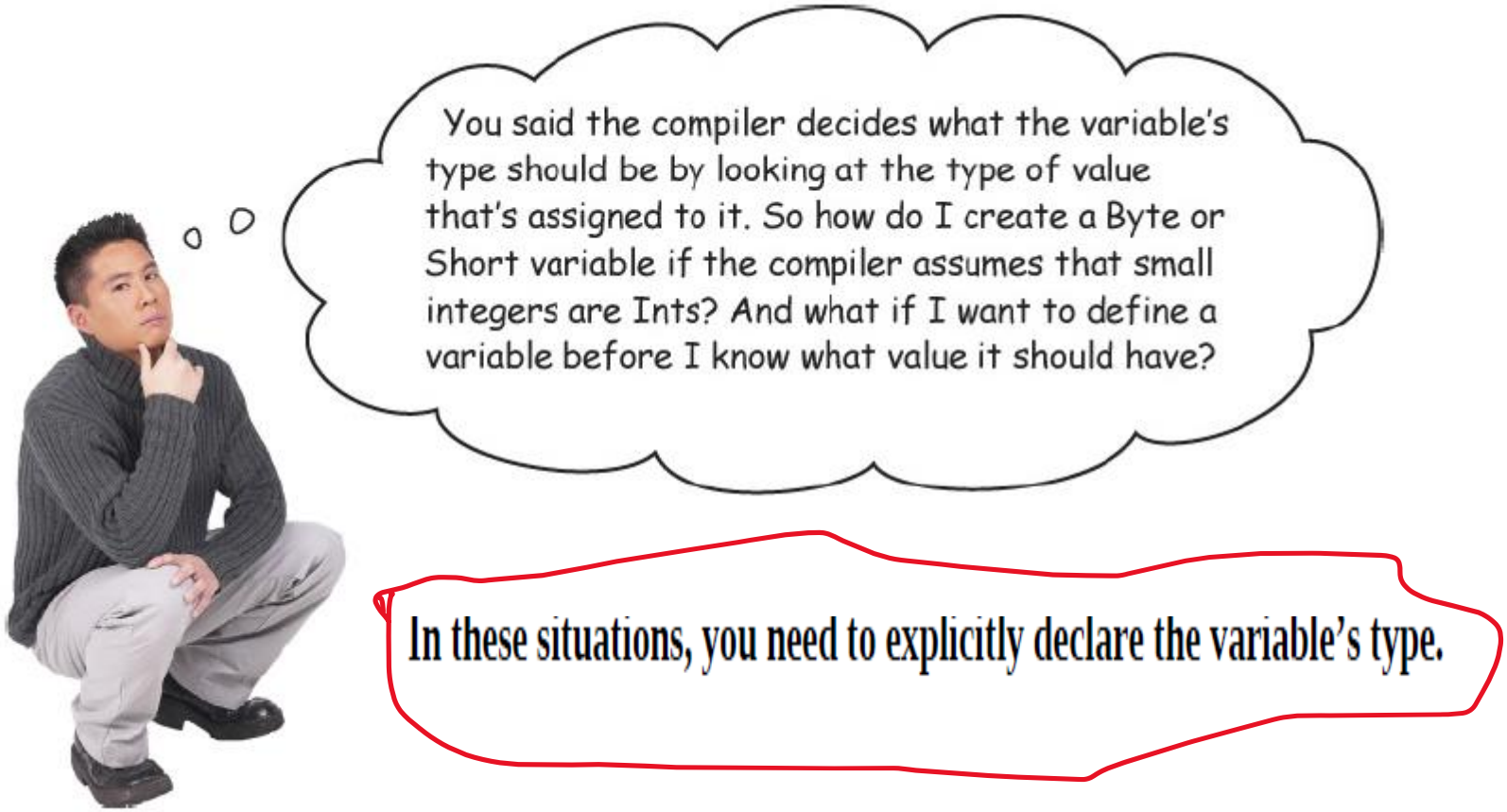
ArrayOf(),
intArrayof()

Data Types

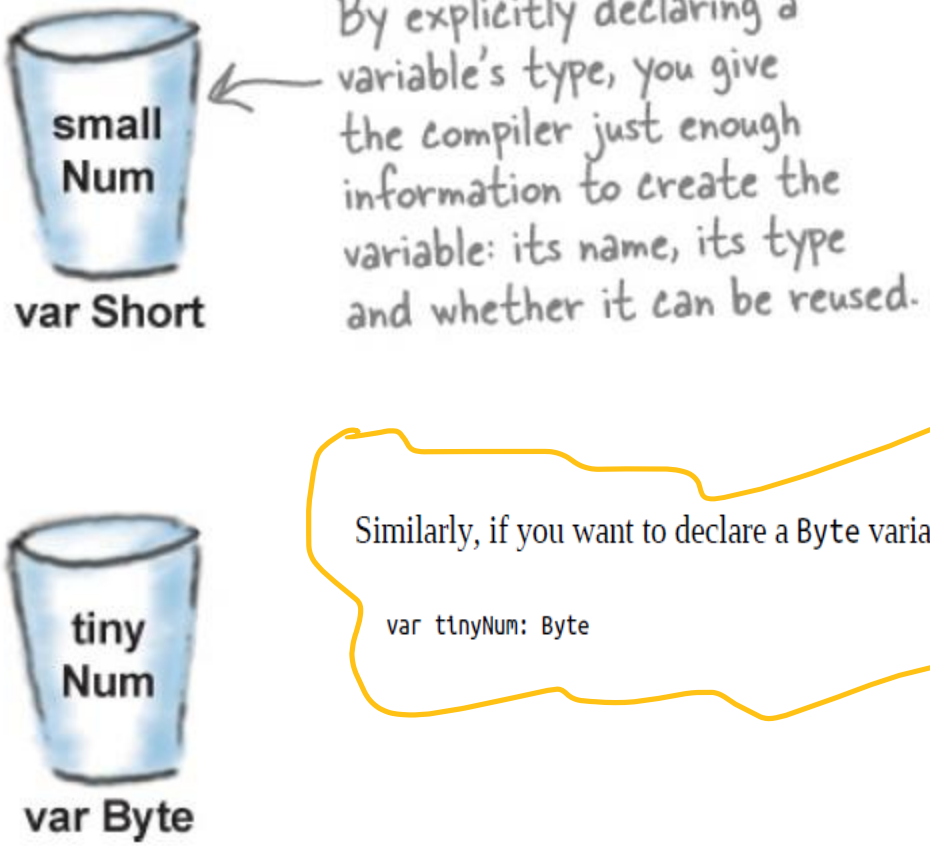


- **String variables are used to hold multiple characters strung together.**
- **We create a String variable by assigning the characters enclosed in double quotes:**
- **var name = "Fido"**

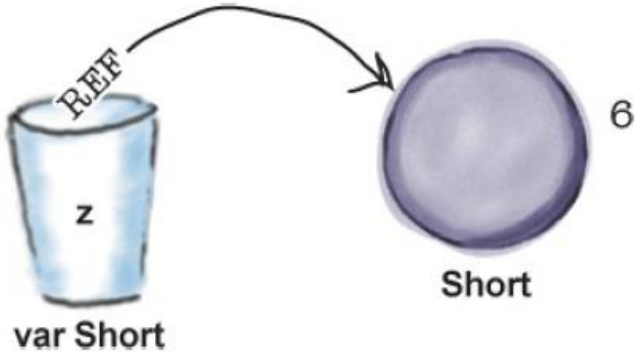
Data Types


You said the compiler decides what the variable's type should be by looking at the type of value that's assigned to it. So how do I create a Byte or Short variable if the compiler assumes that small integers are Ints? And what if I want to define a variable before I know what value it should have?
In these situations, you need to explicitly declare the variable's type.

How to explicitly declare a variable's type



How to explicitly declare a variable's type




```
var x: Int
var y = x + 6
value:
```

x hasn't been assigned a value, so the compiler gets upset.

How to explicitly declare a variable's type

By explicitly declaring a variable's type, you give the compiler just enough information to create the variable: its name, its type and whether it can be reused.

Similarly, if you want to declare a `Byte` variable, you use code like this:

```
var tinyNum: Byte
```

Use the right value for the variable's type

Kotlin

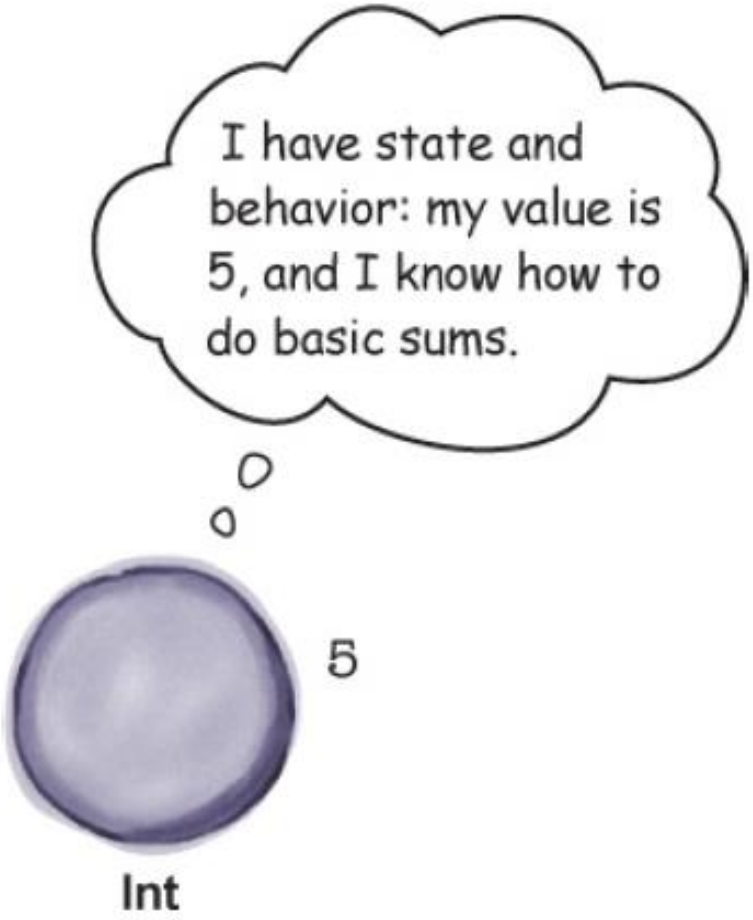
- The compiler really cares about a variable's type so that it can stop you from performing inappropriate operations that may lead to bugs in your code.
- As an example, if you try to assign a floating-point number such as 3.12 to an integer variable, the compiler will refuse to compile your code.
- The following code, for example, won't work:
  - var x: Int = 3.12
- The compiler realizes that 3.12 won't fit into an Int without some loss of precision (like, everything after the decimal point), so it refuses to compile the code

Use the right value for the variable's type

Kotlin

- In order to assign a literal value to a variable, we need to make sure that the value is compatible with the variable's type.
- This is particularly important when we want to assign the value of one variable to another.
- The Kotlin compiler will only let you assign a value to a variable if the value and variable are compatible

An object has state and behavior

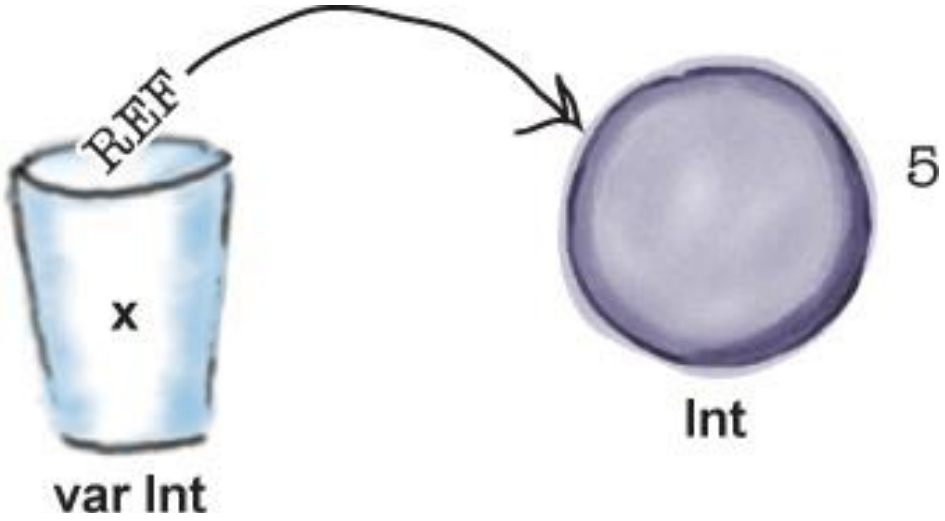


How to convert a numeric value to another type

```
var x = 5
var z: Long = x.toLong()
```

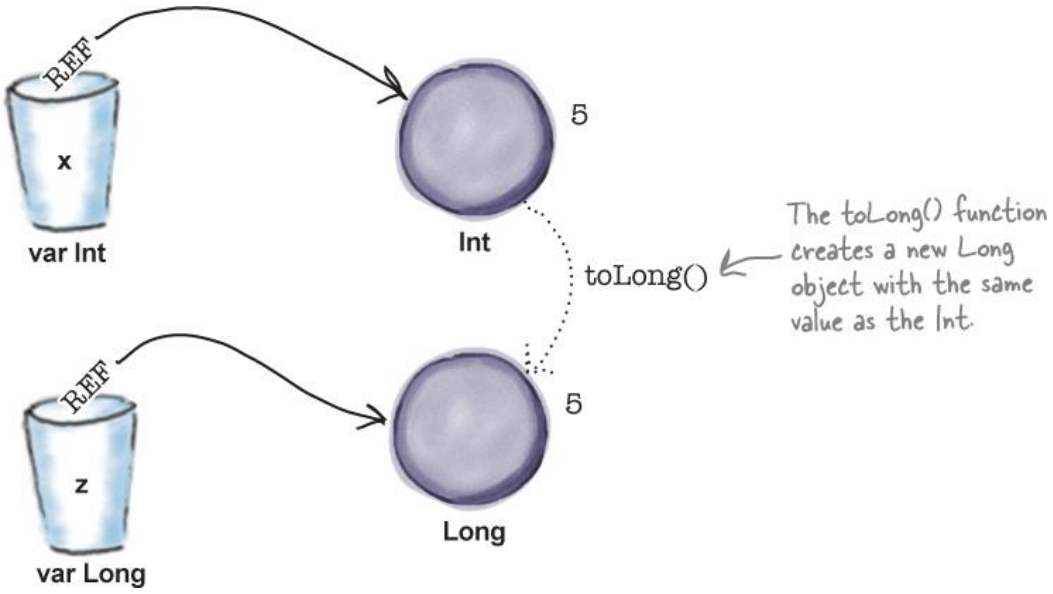
How to convert a numeric value to another type

- var x = 5
- This creates an Int variable named x, and an Int object with a value of 5.
- x holds a reference to that object.



How to convert a numeric value to another type

- var z: Long = x.toLong()
- This creates a new Long variable, z.
- The toLong() function on x's object is called, and this creates a new Long object with a value of 5.
- A reference to the Long object gets put into the z variable.

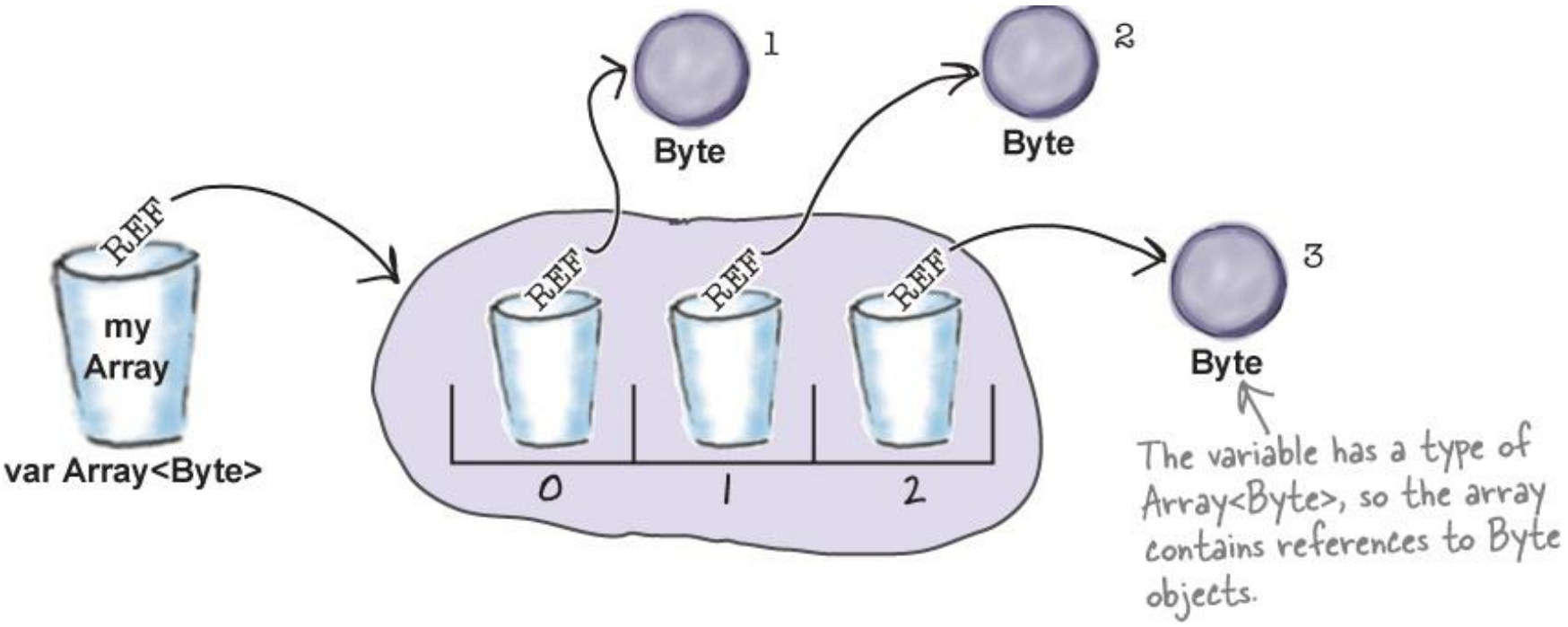


How to explicitly define the array's type

Kotlin

- We can explicitly define what type of items an array should hold.
- As an example, suppose we wanted to declare an array that holds Byte values.
- To do this, you would use code like the following:
- var myArray: Array<Byte> = arrayOf(1, 2, 3)
- The code Array<Byte> tells the compiler that we want to create an array that holds Byte variables.
- In general, simply specify the type of array we want to create by putting the type between the angle brackets (<>).

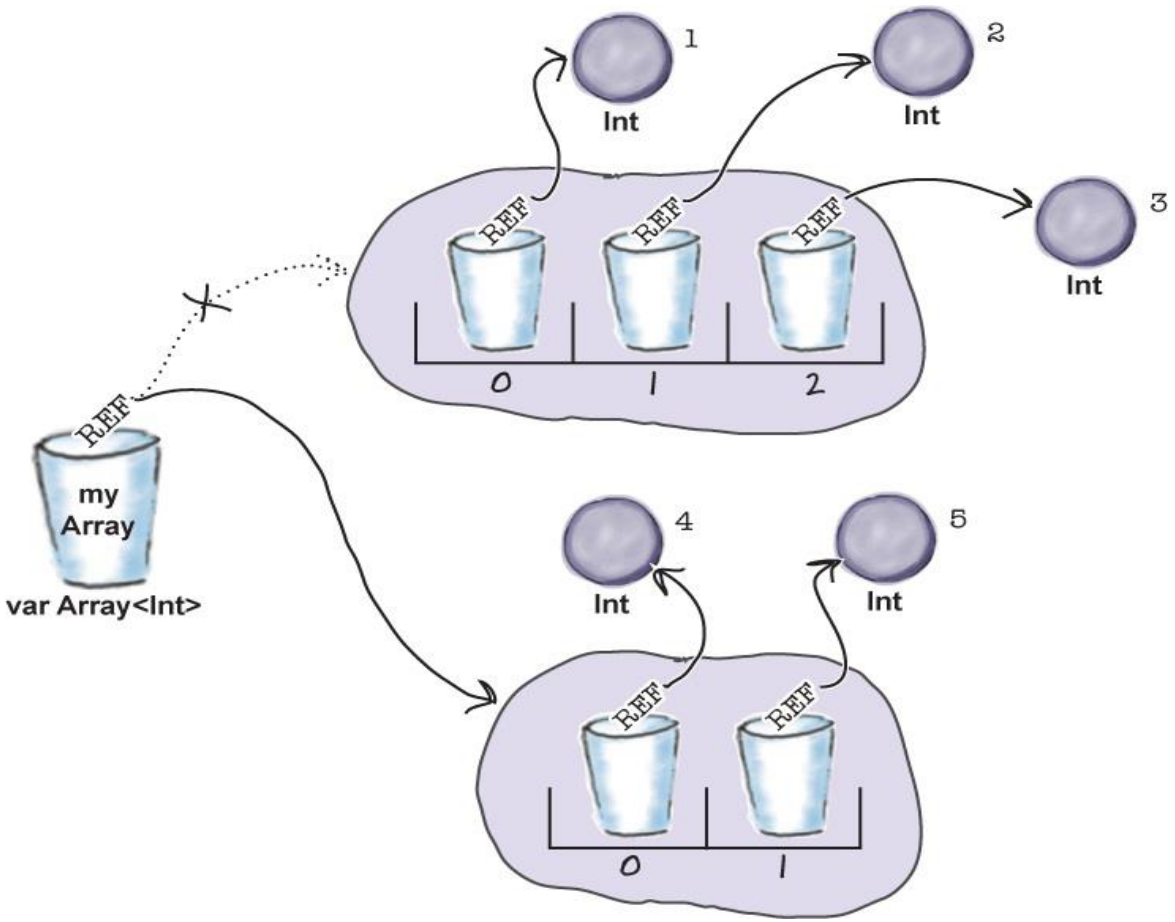
How to explicitly define the array's type



How to explicitly define the array's type

```
var myArray = arrayOf(1, 2, 3)
myArray = arrayOf(4, 5)
```

This is a brand-new array.



Default Imports

Kotlin

- A number of packages are imported into every Kotlin file by default:
  - kotlin.*
  - kotlin.annotation.*
  - kotlin.collections.*
  - kotlin.comparisons.*
  - kotlin.io.*
  - kotlin.ranges.*
  - kotlin.sequences.*
  - kotlin.text.*

Default Imports

Kotlin

- Additional packages are imported depending on the target platform:
- JVM:
  - java.lang.*
  - kotlin.jvm.*
- JS:
  - kotlin.js.*

Imports

Kotlin

- We can import either a single name:
  - import org.example.Message // Message is now accessible without qualification
- or all the accessible contents of a scope: package, class, object, and so on:
  - import org.example.* // everything in 'org.example' becomes accessible
- If there is a name clash, we can disambiguate by using as keyword to locally rename the clashing entity:
  - import org.example.Message // Message is accessible
  - import org.test.Message as testMessage // testMessage stands for 'org.test.Message'

Imports

Kotlin

- The import keyword is not restricted to importing classes; we can also use it to import other declarations:
  - top-level functions and properties
  - functions and properties declared in object declarations
  - enum constants

Visibility of top-level declarations

Kotlin

- If a top-level declaration is marked private, it is private to the file it's declared in.

Functions

Kotlin

A function with two Int parameters and Int return type.

```
//sampleStart
fun sum(a: Int, b: Int): Int {
    return a + b
}
//sampleEnd

fun main() {
    print("sum of 3 and 5 is ")
    println(sum(3, 5))
}
```

Functions

Kotlin

A function body can be an expression. Its return type is inferred.

```
//sampleStart
fun sum(a: Int, b: Int) = a + b
//sampleEnd

fun main() {
    println("sum of 19 and 23 is ${sum(19, 23)}")
}
```

Functions

Kotlin

A function that returns no meaningful value.

```
//sampleStart
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
//sampleEnd

fun main() {
    printSum(-1, 8)
}
```

Functions

Kotlin

Unit return type can be omitted.

```
//sampleStart
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}
//sampleEnd

fun main() {
    printSum(-1, 8)
}
```

Variables

Kotlin

Read-only local variables are defined using the keyword val. They can be assigned a value only once.

```
fun main() {
//sampleStart
    val a: Int = 1  // immediate assignment
    val b = 2   // `Int` type is inferred
    val c: Int  // Type required when no initializer is provided
    c = 3       // deferred assignment
//sampleEnd
    println("a = $a, b = $b, c = $c")
}
```

Variables

Kotlin

Variables that can be reassigned use the var keyword.

```
fun main() {
//sampleStart
    var x = 5 // `Int` type is inferred
    x += 1
//sampleEnd
    println("x = $x")
}
```

Variables

Kotlin

You can declare variables at the top level.

```
//sampleStart
val PI = 3.14
var x = 0

fun incrementX() {
    x += 1
}
//sampleEnd

fun main() {
    println("x = $x; PI = $PI")
    incrementX()
    println("incrementX()")
    println("x = $x; PI = $PI")
}
```

String Templates

Kotlin

```
fun main() {
//sampleStart
    var a = 1
    // simple name in template:
    val s1 = "a is $a"

    a = 2
    // arbitrary expression in template:
    val s2 = "${s1.replace("is", "was")}, but now is $a"
//sampleEnd
    println(s2)
}
```

Conditional Expressions

Kotlin

```
//sampleStart
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}
//sampleEnd

fun main() {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

Conditional Expressions

Kotlin

```
//sampleStart
fun maxOf(a: Int, b: Int) = if (a > b) a else b
//sampleEnd

fun main() {
    println("max of 0 and 42 is ${maxOf(0, 42)}")
}
```

For Loop

Kotlin

```
fun main() {
//sampleStart
    val items = listOf("apple", "banana", "kiwifruit")
    for (item in items) {
        println(item)
    }
//sampleEnd
}
```

For Loop

```
fun main() {
//sampleStart
    val items = listOf("apple", "banana", "kiwifruit")
    for (index in items.indices) {
        println("item at $index is ${items[index]}")
    }
//sampleEnd
}
```

While Loop

Kotlin

```
val items = listOf("apple", "banana", "kiwifruit")
var index = 0
while (index < items.size) {
    println("item at $index is ${items[index]}")
    index++
}
```

When Expression

```
//sampleStart
fun describe(obj: Any): String =
    when (obj) {
        1          -> "One"
        "Hello"    -> "Greeting"
        is Long    -> "Long"
        !is String -> "Not a string"
        else       -> "Unknown"
    }
//sampleEnd

fun main() {
    println(describe(1))
    println(describe("Hello"))
    println(describe(1000L))
    println(describe(2))
    println(describe("other"))
}
```

Range

Kotlin

Check if a number is within a range using in operator.

```
fun main() {
//sampleStart
    val x = 10
    val y = 9
    if (x in 1..y+1) {
        println("fits in range")
    }
//sampleEnd
}
```

Range

Kotlin

Check if a number is out of range.

```
fun main() {
//sampleStart
    val list = listOf("a", "b", "c")

    if (-1 !in 0..list.lastIndex) {
        println("-1 is out of range")
    }
    if (list.size !in list.indices) {
        println("list size is out of valid list indices range, too")
```

Range

Kotlin

Iterate over a range.

```
fun main() {
//sampleStart
    for (x in 1..5) {
        print(x)
    }
//sampleEnd
}
```

Or over a progression.

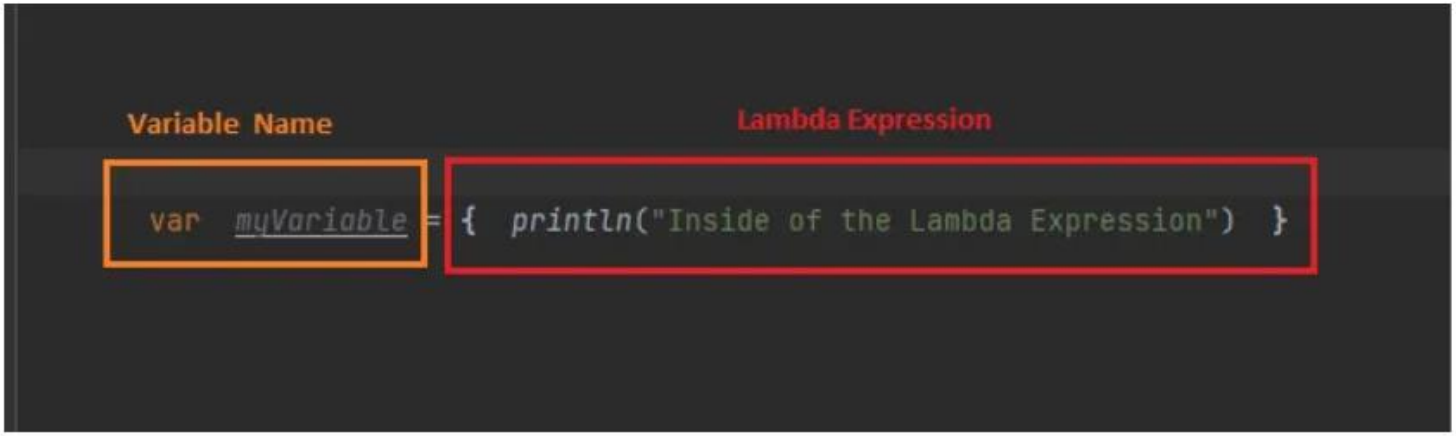
```
fun main() {
//sampleStart
    for (x in 1..10 step 2) {
        print(x)
    }
    println()
    for (x in 9 downTo 0 step 3) {
        print(x)
    }
//sampleEnd
}
```

Lambdas

Kotlin

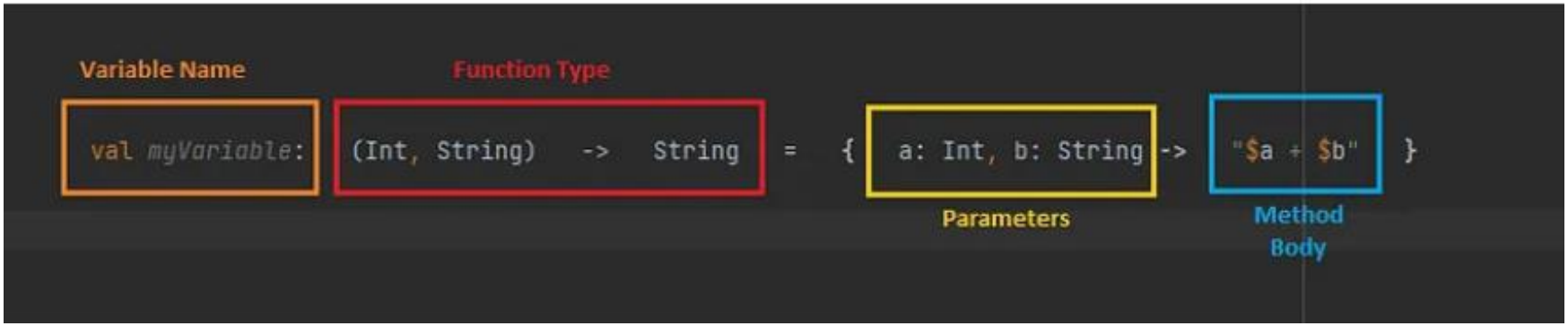
- A lambda expression in Kotlin is a concise, unnamed function enclosed in braces, used for defining code blocks that can be passed as values or stored as variables.

```
//Basic Usage of Lambda expression
 var  myVariable = {  println("Inside of the Lambda Expression")  }
```



Lambdas

- Lambda Expressions can take arguments and return values.




```
1   //There are two shorter syntax.
2
3   // 1- Skip the function type.
4   val myVariable = { a: Int, b: String -> "$a + $b" }
5
6   // 2- Skip the data types inside the curly brackets
7   val myVariable2: (Int, String) -> String = { a, b -> "$a + $b" }
```

Lambdas

Kotlin

```
//There are four function types, varying based on parameters and return types.

// 1-With Parameters and No Return Value:
val myVariable: (Int, String) -> Unit = { a: Int, b: String -> println("$a + $b") }

// 2-With Parameters and Return Value:
val myVariable2: (Int, String) -> String = { a: Int, b: String -> "$a + $b" }

// 3-No Parameters and No Return Value:
val myVariable3: () -> Unit = { println("No Parameters and No Return Value") }

// 4-No Parameters and Return Value:
val myVariable4: () -> String = { "Return String" }
```

Anonymous Function

Kotlin

- In Kotlin, anonymous functions and lambda expressions are both unnamed functions that can be passed as values.
- However, anonymous functions use the ‘fun’ keyword, while lambda expressions are defined with curly braces {}.

```
//Syntax of Anonymous Function
val myVariableName: (FirstDataType,SecondDataType) -> ReturnType =
            fun(firstParameter,secondParameter): ReturnType { MethodBody }

//Example of Anonymous Function:
val myVariable: (String, String) -> String = fun(a, b): String {
    return "$a $b"
}
```

Anonymous Function

Kotlin

```
// Shorter Syntax Of Anonymous Function
val myVariableName = fun(FirstDataType,SecondDataType) : ReturnType { MethodBody }

// Shorter Syntax Of Anonymous Function:
val myVariable = fun(a:String,b:String): String { return "$a + $b" }

// When the method body contains just one statement,
// the return keyword and braces can be omitted.
val myVariable2 = fun(a:String,b:String): String = "$a + $b"
```

Anonymous Function

Kotlin

```
// Let's explore various anonymous function formats,
// depending on the parameters and return type.

// 1- With Parameters and No Return Value:
val myVariable = fun(a: String, b: String): Unit {
    println("$a $b")
}

// 2- With Parameters and Return Value:
val myVariable2 = fun(a: String, b: String): String {
    return "$a $b"
}
```

Anonymous Function

Kotlin

```
// 3- No Parameters and No Return Value:
val myVariable3 = fun(): Unit {
    println("No Parameters and No Return Value:")
}

// 4- No Parameters and Return Value:
val myVariable4 = fun(): String {
    return "Hi!"
}

fun main() {
    myVariable("Hüseyin","Özkoç")
    println(myVariable2("Orkun","Ozan"))
    myVariable3()
    println(myVariable4())
}
```

High Order Functions

Kotlin

- A higher order function is a function that takes one or more functions as parameters and/or returns a function.
- It allows us to abstract over actions, enabling you to write more reusable and modular code.
- By passing functions as arguments, we can customize the behavior of a higher order function without modifying its implementation.

Benefits of Higher Order Functions

Kotlin

- Code Reusability: Higher order functions promote code reuse by allowing you to extract common patterns into reusable functions. By passing different functions as arguments, you can achieve different behaviors without duplicating code.

- Abstraction: Higher order functions abstract away the details of how a specific task is performed. They focus on what needs to be done rather than how it should be done. This level of abstraction improves code readability and maintainability.

Benefits of Higher Order Functions

Kotlin

- Flexibility: Higher order functions provide flexibility by allowing you to change the behavior of a function at runtime. This dynamic behavior can be useful in various scenarios, such as filtering, mapping, or sorting collections based on different criteria.

Higher Order Functions

Kotlin

The forEach function is a higher order function available on collections in Kotlin. It takes a lambda function as an argument and applies it to each element in the collection.

```
val numbers = listOf(1, 2, 3, 4, 5)
numbers.forEach { println(it) }
```

Higher Order Functions

Kotlin

Example 2: **filter**

The `filter` function is another higher order function that takes a predicate function as an argument. It returns a new collection containing only the elements that satisfy the predicate.

```
val numbers = listOf(1, 2, 3, 4, 5)
val evenNumbers = numbers.filter { it % 2 == 0 }
println(evenNumbers) // Output: [2, 4]
```

Higher Order Functions

Kotlin

Example 3: **map**

The `map` function is used to transform each element in a collection by applying a transformation function to it.

```
val numbers = listOf(1, 2, 3, 4, 5)
val squaredNumbers = numbers.map { it * it }
println(squaredNumbers) // Output: [1, 4, 9, 16, 25]
```

Higher Order Functions

Kotlin

```kotlin
// Higher order function example: Calculator
fun calculate(x: Int, y: Int, operation: (Int, Int) -> Int): Int {
  return operation(x, y)
}
fun add(x: Int, y: Int): Int {
  return x + y
}
fun subtract(x: Int, y: Int): Int {
  return x - y
}
fun multiply(x: Int, y: Int): Int {
  return x * y
}
fun main() {
  val result1 = calculate(10, 5, ::add)
  println("Addition: $result1") // Output: Addition: 15
  val result2 = calculate(10, 5, ::subtract)
  println("Subtraction: $result2") // Output: Subtraction: 5
  val result3 = calculate(10, 5, ::multiply)
  println("Multiplication: $result3") // Output: Multiplication: 50
}
```

Inline Functions

Kotlin

- An inline function is declared with a keyword inline.
- Inline function instruct compiler to insert the complete body of the function wherever that function gets used in the code.
- The inline function tells the compiler to copy parameters and functions to the call site.
- The virtual function or local function cannot be declared as inline.

Inline Functions

Kotlin

- Following are some expressions and declarations which are not supported anywhere inside the inline functions:
  - Declaration of local classes
  - Declaration of inner nested classes
  - Function expressions
  - Declarations of local function
  - Default value for optional parameters

Inline Functions

Kotlin

```
fun guide() {
    print("guide start")
    teach()
    print("guide end")
}

fun teach() {
    print("teach")
}
```

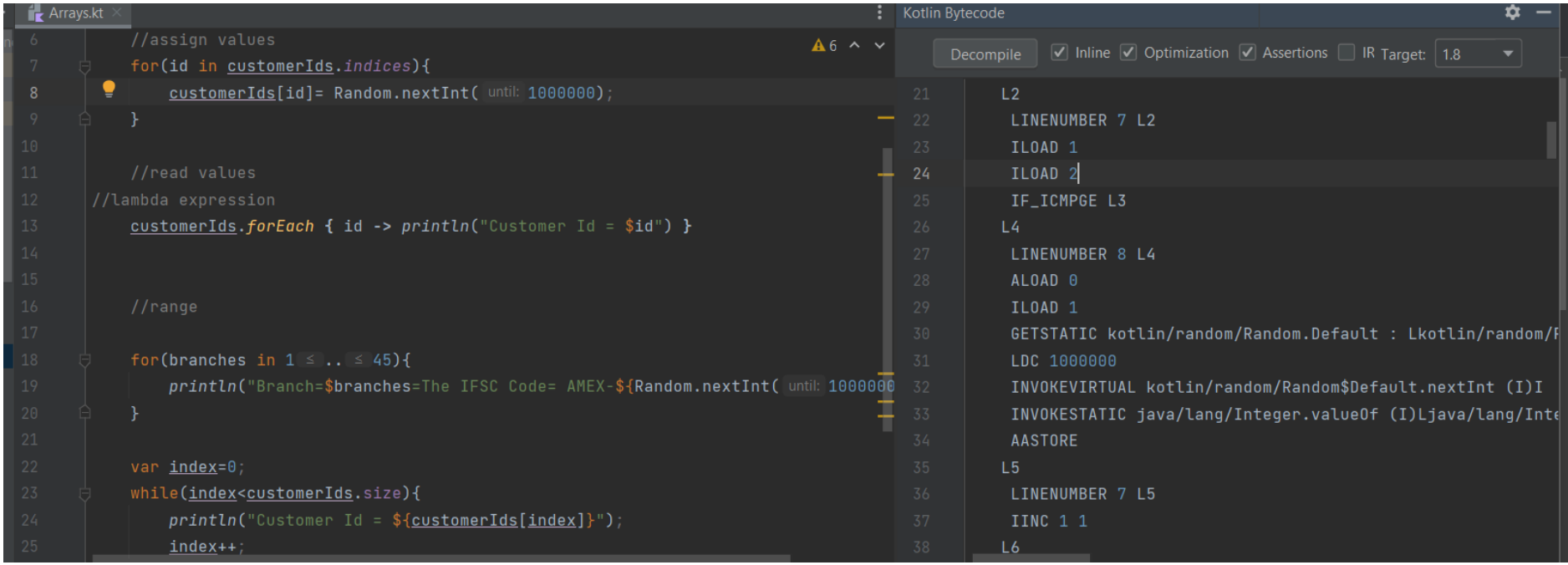
Inline Functions

Let's see the decompiled code in order to understand it.

For that, we will have to convert this Kotlin source file to a Java source file.

Steps to convert from Kotlin source file to Java source file and decompile in Android Studio:

- `Tools` > `Kotlin` > `Show Kotlin Bytecode`. You will get the bytecode of your Kotlin file.
- Now click on the `Decompile` button to get your Java code from the bytecode.



Inline Functions

Kotlin

```
public void guide() {
    System.out.print("guide start");
    teach();
    System.out.print("guide end");
}

public void teach() {
    System.out.print("teach");
}
```

Inline Functions

Kotlin

Now let's add the `inline` keyword to the `teach()` function.

```
fun guide() {
    print("guide start")
    teach()
    print("guide end")
}

inline fun teach() {
    print("teach")
}
```

Again, let's see the decompiled code. The decompiled code is as below:

```
public void guide() {
   System.out.print("guide start");
   System.out.print("teach");
   System.out.print("guide end");
}
```

Inline Functions

Kotlin

- When the function code is very small, it's a good idea to make the function inline.
- When the function code is large and called from so many places, it's a bad idea to make the function inline, as the large code will be repeated and again.

Inline Functions

Kotlin

```kotlin
fun guide() {
    print("guide start")
    teach {
        print("teach")
    }
    print("guide end")
}

inline fun teach(abc: () -> Unit) {
    abc()
}
```

Again, let's go to the decompiled code. The decompiled code is as below:

```java
public void guide() {
    System.out.print("guide start");
    System.out.print("teach");
    System.out.print("guide end");
}
```

Nullable Checks and Null Checks

Kotlin

A reference must be explicitly marked as nullable when null value is possible. Nullable type names have ? at the end.

Return null if str does not hold an integer:

```
fun parseInt(str: String): Int? {
    // ...
}
```

Nullable Checks and Null Checks

Kotlin

```
fun parseInt(str: String): Int? {
    return str.toIntOrNull()
}

//sampleStart
fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

    // Using `x * y` yields error because they may hold nulls.
    if (x != null && y != null) {
        // x and y are automatically cast to non-nullable after null check
        println(x * y)
    }
    else {
        println("'$arg1' or '$arg2' is not a number")
    }
}
//sampleEnd

fun main() {
    printProduct("6", "7")
    printProduct("a", "7")
    printProduct("a", "b")
}
```

Nullable Checks and Null Checks

Kotlin

```
fun parseInt(str: String): Int? {
    return str.toIntOrNull()
}

fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

//sampleStart
```

Nullable Checks and Null Checks

Kotlin

```
    // ...
    if (x == null) {
        println("Wrong number format in arg1: '$arg1'")
        return
    }
    if (y == null) {
        println("Wrong number format in arg2: '$arg2'")
        return
    }

    // x and y are automatically cast to non-nullable after null check
    println(x * y)
//sampleEnd
}

fun main() {
    printProduct("6", "7")
    printProduct("a", "7")
    printProduct("99", "b")
}
```

Type checks and automatic casts

Kotlin

- The is operator checks if an expression is an instance of a type.
- If an immutable local variable or property is checked for a specific type, there's no need to cast it explicitly.

Type checks and automatic casts

Kotlin

```
//sampleStart
fun getStringLength(obj: Any): Int? {
    // `obj` is automatically cast to `String` on the right-hand side of `&&`
    if (obj is String && obj.length > 0) {
        return obj.length
    }

    return null
}
//sampleEnd

fun main() {
    fun printLength(obj: Any) {
        println("Getting the length of '$obj'. Result: ${getStringLength(obj) ?: "Error: The object is not a
string"} ")
    }
    printLength("Incomprehensibilities")
    printLength("")
    printLength(1000)
}
```

Classes

Kotlin

Classes in Kotlin are declared using the keyword class:

```
class Person { /*...*/ }
```

- The class declaration consists of the class name, the class header (specifying its type parameters, the primary constructor, and some other things), and the class body surrounded by curly braces.
- Both the header and the body are optional; if the class has no body, the curly braces can be omitted.

```
class Empty
```

Constructors

Kotlin

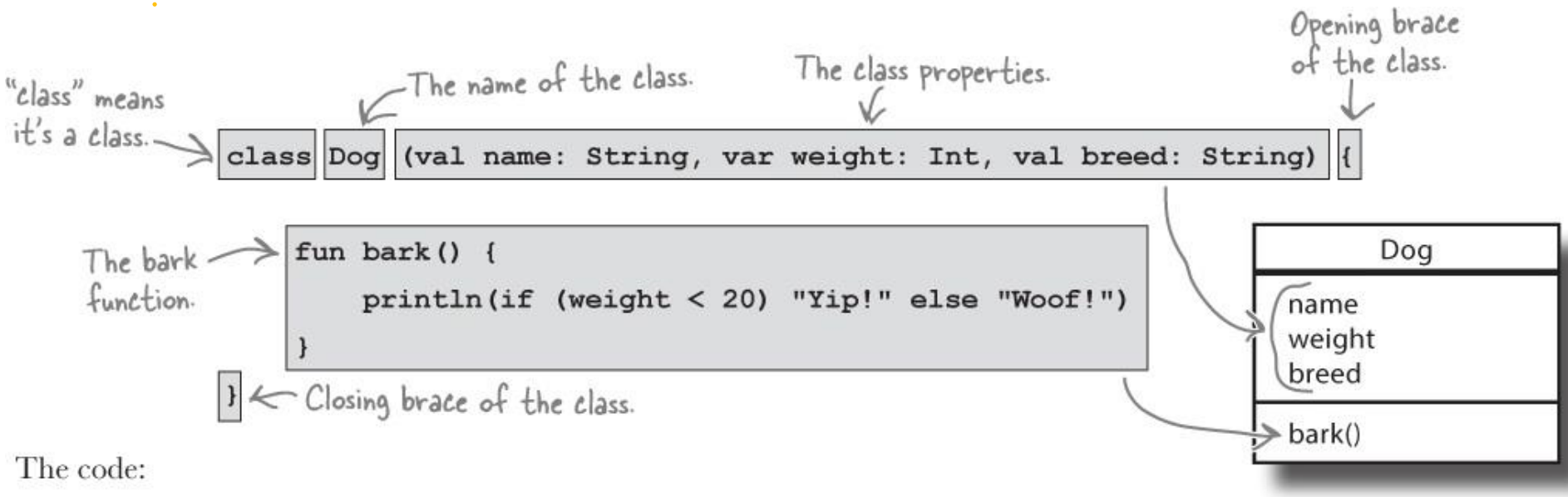
- A class in Kotlin can have a primary constructor and one or more secondary constructors.
- The primary constructor is a part of the class header, and it goes after the class name and optional type parameters.

```
class Person constructor(firstName: String) { /*...*/ }
```

If the primary constructor does not have any annotations or visibility modifiers, the constructor keyword can be omitted:

```
class Person(firstName: String) { /*...*/ }
```

Constructors



The code:

```
class Dog(val name: String, var weight: Int, val breed: String) {
    ...
}
```

Constructors

Kotlin

- A class in Kotlin can have a primary constructor and one or more secondary constructors.
- The primary constructor is a part of the class header, and it goes after the class name and optional type parameters.

```
class Person constructor(firstName: String) { /*...*/ }
```

- If the primary constructor does not have any annotations or visibility modifiers, the constructor keyword can be omitted:

```
class Person(firstName: String) { /*...*/ }
```

Constructors

- The primary constructor cannot contain any code.
- Initialization code can be placed in initializer blocks prefixed with the init keyword.
- During the initialization of an instance, the initializer blocks are executed in the same order as they appear in the class body, interleaved
- with the property initializers:

```
//sampleStart
class InitOrderDemo(name: String) {
    val firstProperty = "First property: $name".also(::println)

    init {
        println("First initializer block that prints $name")
    }

    val secondProperty = "Second property: ${name.length}".also(::println)

    init {
        println("Second initializer block that prints ${name.length}")
    }
}
//sampleEnd

fun main() {
    InitOrderDemo("hello")
}
```

Constructors

Kotlin

Primary constructor parameters can be used in the initializer blocks. They can also be used in property initializers declared in the class body:

```
class Customer(name: String) {
    val customerKey = name.uppercase()
}
```

Kotlin has a concise syntax for declaring properties and initializing them from the primary constructor:

```
class Person(val firstName: String, val lastName: String, var age: Int)
```

Such declarations can also include default values of the class properties:

```
class Person(val firstName: String, val lastName: String, var isEmployed: Boolean = true)
```

You can use a trailing comma when you declare class properties:

```
class Person(
    val firstName: String,
    val lastName: String,
    var age: Int, // trailing comma
) { /*...*/ }
```

Constructors

Kotlin

If the constructor has annotations or visibility modifiers, the constructor keyword is required and the modifiers go before it:

```
class Customer public @Inject constructor(name: String) { /*...*/ }
```

Secondary constructors

A class can also declare secondary constructors, which are prefixed with constructor:

```
class Person(val pets: MutableList<Pet> = mutableListOf())

class Pet {
    constructor(owner: Person) {
        owner.pets.add(this) // adds this pet to the list of its owner's pets
    }
}
```

Constructors

Kotlin

- If the class has a primary constructor, each secondary constructor needs to delegate to the primary constructor, either directly or indirectly through another secondary constructor(s).
- Delegation to another constructor of the same class is done using the **this** keyword:

```
class Person(val name: String) {
    val children: MutableList<Person> = mutableListOf()
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Constructors

Kotlin

- Code in initializer blocks effectively becomes part of the primary constructor.
- Delegation to the primary constructor happens as the first statement of a secondary constructor, so the code in all initializer blocks and property initializers is executed before the body of the secondary constructor.

```
//sampleStart
class Constructors {
    init {
        println("Init block")
    }

    constructor(i: Int) {
        println("Constructor $i")
    }
}
//sampleEnd

fun main() {
    Constructors(1)
}
```

Constructors

Kotlin

- If we don't want yr class to have a public constructor, declare an empty primary constructor with non-default visibility:
- class DontCreateMe private constructor() { /*...*/ }

On the JVM, if all of the primary constructor parameters have default values, the compiler will generate an additional parameterless constructor which will use the default values. This makes it easier to use Kotlin with libraries such as Jackson or JPA that create class instances through parameterless constructors.

```
class Customer(val customerName: String = "")
```

Creating Instances of Class

Kotlin

To create an instance of a class, call the constructor as if it were a regular function:

```
val invoice = Invoice()

val customer = Customer("Joe Smith")
```

Kotlin does not have a new keyword.

Creating Instances of Class

Kotlin

One class

Many objects

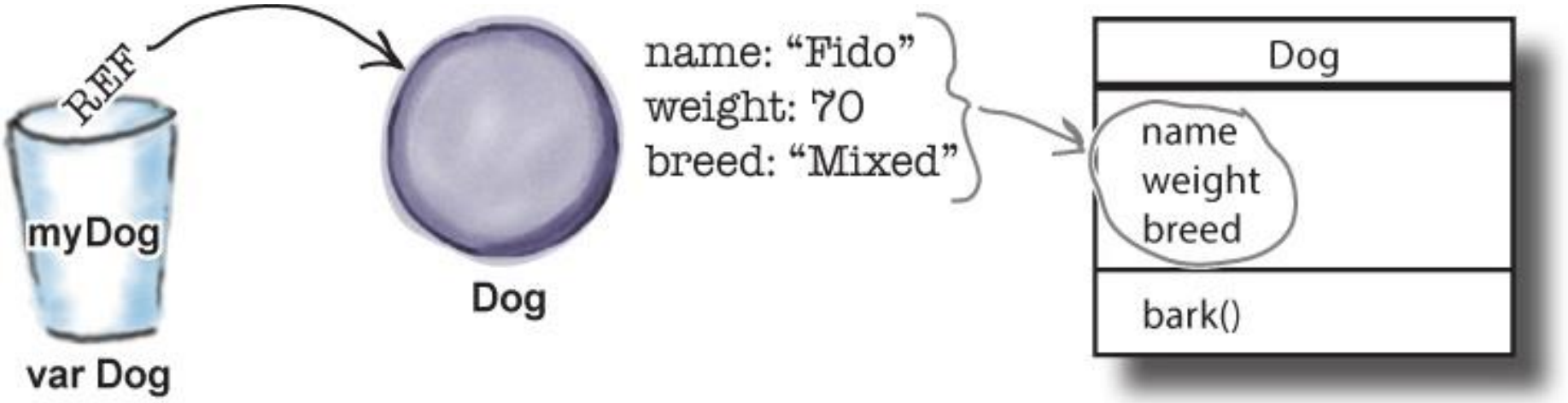
```
var myDog = Dog("Fido", 70, "Mixed")
```

You create a Dog by passing it arguments for the three properties.

The code passes three arguments to the `Dog` object. These match the properties we defined in the `Dog` class: the `Dog`'s `name`, `weight` and `breed`:

```
class Dog(val name: String, var weight: Int, val breed: String) {
    ...
}
```

Creating Instances of Class

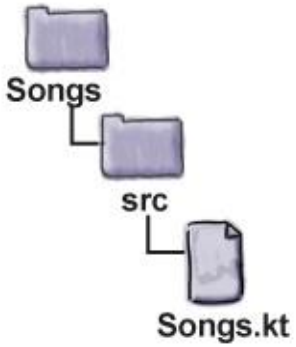


Creating Instances of Class

```
class Song(val title: String, val artist: String) {
    fun play() {
        println("Playing the song $title by $artist")
    }

    fun stop() {
        println("Stopped playing $title")
    }
}

fun main(args: Array<String>) {
    val songOne = Song("The Mesopotamians", "They Might Be Giants")
    val songTwo = Song("Going Underground", "The Jam")
    val songThree = Song("Make Me Smile", "Steve Harley")
    songTwo.play()
    songTwo.stop()
    songThree.play()
}
```

Define title and artist properties.

Add play and stop functions.

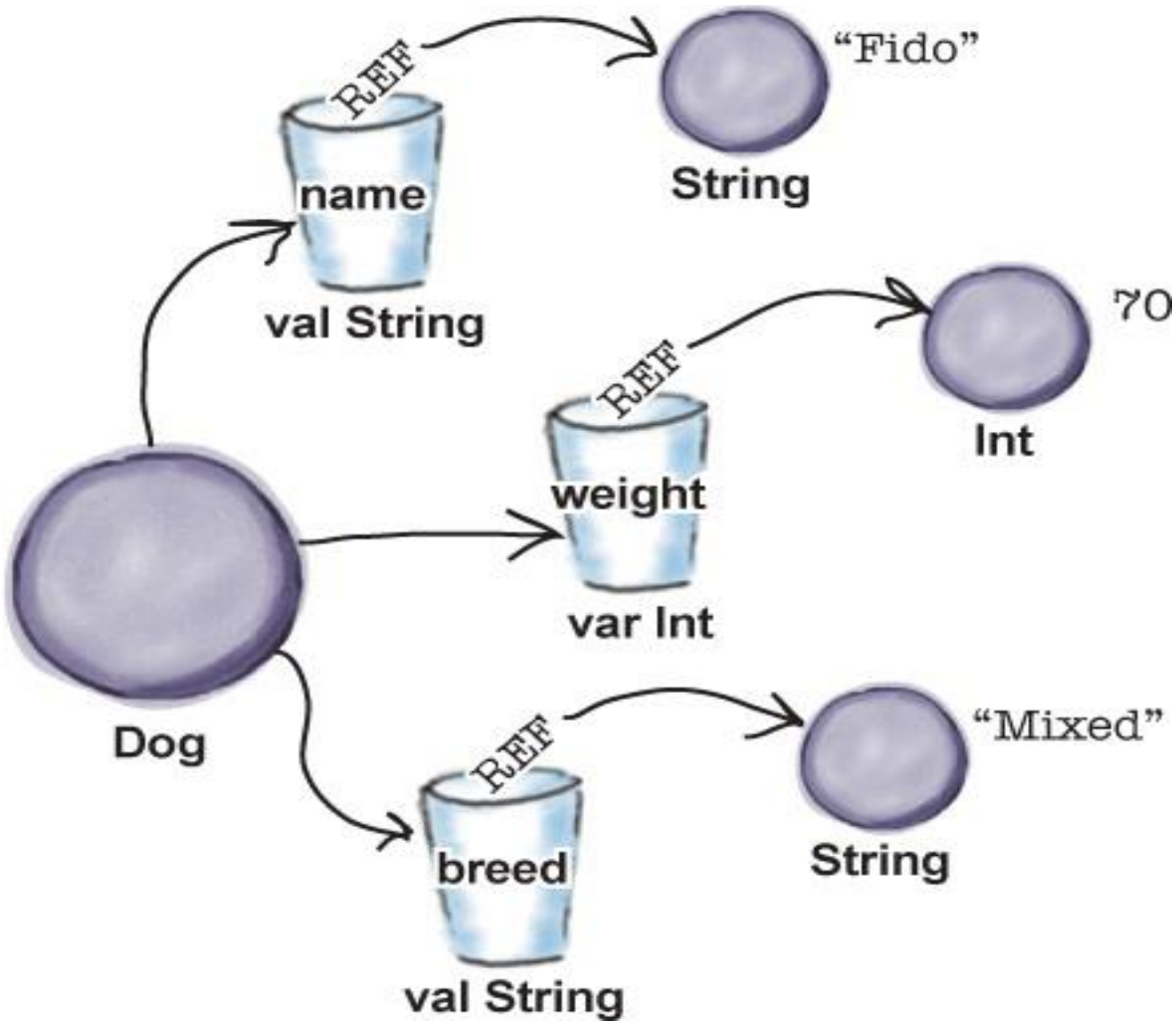


Create three Songs.

Play songTwo, stop it, then play songThree.

Creating Instances of Class

```
class Dog(val name: String,
          var weight: Int,
          val breed: String) {
}
```



Behind the scenes without @JvmStatic

Kotlin

Kotlin code

```
class Plant {
    companion object {
        fun waterAll() { }
    }
}
```

Decompiled Java code

```
public final class Plant {

   public static final Plant.Companion Companion = new Plant.Companion();

   public static final class Companion {

      public final void waterAll() { }

      private Companion() { }
   }
}
```

Behind the scenes without @JvmStatic

Kotlin

- The simplified decompiled Java code above, a class named Companion is generated to represent the companion object.
- The class Plant holds the singleton instance new Plant.Companion() of the class Plant.Companion.
- The instance is also named as Companion.
- This is the reason you need to call the functions/properties of the companion object in Java using the Plant.Companion:
  - Plant.Companion.waterAll();

Behind the scenes with @JvmStatic

Kotlin

Kotlin code

```
class Plant {
    companion object {
        @JvmStatic
        fun waterAll() { }
    }
}
```

Decompiled Java code

```
public final class Plant {

   public static final Plant.Companion Companion = new Plant.Companion();

   @JvmStatic
   public static final void waterAll() { Companion.waterAll();}

   public static final class Companion {
      @JvmStatic
      public final void waterAll() { }

      private Companion() { }
   }
}
```

Behind the scenes with @JvmStatic

Kotlin

- When we annotate a function of a companion object with @JvmStatic in Kotlin, a pure static function waterAll() is generated in addition to the non static function waterAll().
- So, now we can call the function without the Companion name which is more idiomatic to Java:
  - Plant.waterAll();

@JvmStatic and without @JvmStatic

Kotlin

- There is no performance gain or loss in terms of memory allocation.
- The reason is that, as we can see in the code above, the extra static function that is generated delegates its work to the non static function Companion.waterAll().
- This means, creation of the Companion instance is required in both the cases, with @JvmStatic as well as without @JvmStatic.
- The behavior of both the setups is the same apart from the extra method that is generated.
- In Android, if we do the method count, we may need to keep an eye on this because an extra copy is created for each annotated function.

When to use @JvmStatic

Kotlin

- When we know that your Kotlin code won't be used in Java, we don't have to worry about adding the @JvmStatic annotation.
- This keeps our code cleaner.
- However, if our Kotlin code is called from Java, it makes sense to add the annotation.
- This will prevent our Java code from polluting with the name Companion everywhere.
- It's not like an additional keyword on either side.
- If we add @JvmStatic in one place, we can prevent writing the extra Companion word in thousands of places, wherever you call that function.
- This is especially useful for library creators, if they add @JvmStatic in their Kotlin library, the users of that library won't have to use the Companion word in their Java code.

Class Members

Kotlin

Classes can contain:

- Constructors and initializer blocks
- Functions
- Properties
- Nested and inner classes
- Object declarations

Properties

Kotlin

```
class Address {
    var name: String = "Holmes, Sherlock"
    var street: String = "Baker"
    var city: String = "London"
    var state: String? = null
    var zip: String = "123456"
}
```

To use a property, simply refer to it by its name:

```
fun copyAddress(address: Address): Address {
    val result = Address() // there's no 'new' keyword in Kotlin
    result.name = address.name // accessors are called
    result.street = address.street
    // ...
    return result
}
```

Properties

```
//sampleStart
class Rectangle(val width: Int, val height: Int) {
    val area: Int // property type is optional since it can be inferred from the getter's return type
        get() = this.width * this.height
}
//sampleEnd
fun main() {
    val rectangle = Rectangle(3, 4)
    println("Width=${rectangle.width}, height=${rectangle.height}, area=${rectangle.area}")
}
```

Open keyword in Kotlin

Kotlin

- The open keyword with the class means the class is open for the extension meaning that we can create a subclass of that open class.
- In Kotlin, all the classes are final by default meaning that they can not be inherited.
- In Kotlin, we can mark a class, a function, or a variable with the open keyword like below:

Open keyword in Kotlin

Kotlin

Class

```
open class Mentor {

}
```

Function

```
open fun guide() {

}
```

Variable

```
open val slotsAvailable = 5
```

open keyword with class

Kotlin

```
class Mentor {

}
```

We will not be able to create a subclass of the class.

```
class ExperiencedMentor: Mentor() {

}
```

The compiler will show an error.

```
This type is final, so it cannot be inherited from
```

open keyword with class

Kotlin

Our updated `Mentor` class:

```
open class Mentor {

}
```

Now, we will be able to create a subclass of this class.

```
class ExperiencedMentor: Mentor() {

}
```

open keyword with function

Kotlin

Consider a function `guide()` inside the `Mentor` class:

```
open class Mentor {

    fun guide() {

    }

}
```

We will not be able to override the function.

```
class ExperiencedMentor : Mentor() {

    override fun guide() {

    }

}
```

The compiler will show an error.

```
guide' in 'Mentor' is final and cannot be overridden
```

open keyword with function

Kotlin

Our updated `Mentor` class:

```
open class Mentor {

    open fun guide() {

    }

}
```

We will be able to override the function now like below:

```
class ExperiencedMentor : Mentor() {

    override fun guide() {

    }

}
```

It will work perfectly.

open keyword with variable

Kotlin

Here is the base class `Mentor`:

```
open class Mentor {

    val slotsAvailable = 5

}
```

Here is the child class.

```
class ExperiencedMentor : Mentor() {

    override val slotsAvailable = 10

}
```

We will not be able to override the variable.

The compiler will show an error.

```
'slotsAvailable' in 'Mentor' is final and cannot be overridden
```

open keyword with variable

Kotlin

Our updated `Mentor` class:

```
open class Mentor {

    open val slotsAvailable = 5

}
```

We will be able to override the variable now like below:

```
class ExperiencedMentor : Mentor() {

    override val slotsAvailable = 10

}
```

It will work perfectly.

Visibilities- Class members

Kotlin

- private means that the member is visible inside this class only (including all its members).
- protected means that the member has the same visibility as one marked as private, but that it is also visible in subclasses.
- internal means that any client inside this module who sees the declaring class sees its internal members.
- public means that any client who sees the declaring class sees its public members.

Visibilities- Class members

Kotlin

```
open class Outer {
    private val a = 1
    protected open val b = 2
    internal open val c = 3
    val d = 4  // public by default

    protected class Nested {
        public val e: Int = 5
    }
}

class Subclass : Outer() {
    // a is not visible
    // b, c and d are visible
    // Nested and e are visible

    override val b = 5   // 'b' is protected
    override val c = 7   // 'c' is internal
}

class Unrelated(o: Outer) {
    // o.a, o.b are not visible
    // o.c and o.d are visible (same module)
    // Outer.Nested is not visible, and Nested::e is not visible either
}
```

Data Classes

Kotlin

- Data classes are a special type of class in Kotlin that are primarily designed to hold data and represent simple value objects.
- They are concise and come with built-in functionality that reduces boilerplate code.
- When defining a data class, you don't have to write tedious and repetitive code for things like property declaration, equals(), hashCode(), toString(), and copy() methods.
- Kotlin generates all these methods automatically based on the properties defined in the class.

Declaring a Data Class

Kotlin

Declaring a data class in Kotlin is as simple as adding the `data` keyword before the `class` keyword. Here's an example:

```
data class Person(val name: String, val age: Int)
```

Data Class Equivalent Regular Class

Kotlin

```
class Person(val name: String, val age: Int) {
    // Equals method
    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other !is Person) return false
        if (name != other.name) return false
        if (age != other.age) return false
        return true
    }

    // HashCode method
    override fun hashCode(): Int {
        var result = name.hashCode()
        result = 31 * result + age
        return result
    }

    // ToString method
    override fun toString(): String {
        return "Person(name='$name', age=$age)"
    }

    // Copy method
    fun copy(name: String = this.name, age: Int = this.age): Person {
        return Person(name, age)
    }
}
```

Create DTOs (POJOs/POCOs)

Kotlin

- data class Customer(val name: String, val email: String)

- provides a Customer class with the following functionality:
  - getters (and setters in case of vars) for all properties
  - equals()
  - hashCode()
  - toString()
  - copy()
  - component1(), component2(), ..., for all properties

Abstract Class

```
abstract class Polygon {
    abstract fun draw()
}

class Rectangle : Polygon() {
    override fun draw() {
        // draw the rectangle
    }
}
```

We can override a non-abstract open member with an abstract one.

```
open class Polygon {
    open fun draw() {
        // some default polygon drawing method
    }
}

abstract class WildShape : Polygon() {
    // Classes that inherit WildShape need to provide their own
    // draw method instead of using the default on Polygon
    abstract override fun draw()
}
```

Inheritance

Kotlin

All classes in Kotlin have a common superclass, Any, which is the default superclass for a class with no supertypes declared:

```
class Example // Implicitly inherits from Any
```

Any has three methods: equals(), hashCode(), and toString(). Thus, these methods are defined for all Kotlin classes.

By default, Kotlin classes are final - they can't be inherited. To make a class inheritable, mark it with the open keyword:

```
open class Base // Class is open for inheritance
```

To declare an explicit supertype, place the type after a colon in the class header:

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

If the derived class has a primary constructor, the base class can (and must) be initialized in that primary constructor according to its parameters.

Inheritance

Kotlin

- If the derived class has no primary constructor, then each secondary constructor must initialize the base type using the super keyword or it must delegate to another constructor which does.
- Note that in this case different secondary constructors can call different constructors of the base type:

```
class MyView : View {
    constructor(ctx: Context) : super(ctx)

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)
}
```

Overriding Methods

Kotlin

Kotlin requires explicit modifiers for overridable members and overrides:

```
open class Shape {
    open fun draw() { /*...*/ }
    fun fill() { /*...*/ }
}

class Circle() : Shape() {
    override fun draw() { /*...*/ }
}
```

- The override modifier is required for Circle.draw(). If it were missing, the compiler would complain.
- If there is no open modifier on a function, like Shape.fill(), declaring a method with the same signature in a subclass is not allowed, either with override or without it.
- The open modifier has no effect when added to members of a final class – a class without an open modifier.

Overriding Methods

Kotlin

- A member marked override is itself open, so it may be overridden in subclasses. If you want to prohibit re-overriding, use final:

```
open class Rectangle() : Shape() {
    final override fun draw() { /*...*/ }
}
```

Overriding Properties

Kotlin

- The overriding mechanism works on properties in the same way that it does on methods.
- Properties declared on a superclass that are then redeclared on a derived class must be prefaced with override, and they must have a compatible type.
- Each declared property can be overridden by a property with an initializer or by a property with a get method:

```
open class Shape {
    open val vertexCount: Int = 0
}

class Rectangle : Shape() {
    override val vertexCount = 4
}
```

Overriding Properties

Kotlin

- We can also override a val property with a var property, but not vice versa.
- This is allowed because a val property essentially declares a get method and overriding it as a var additionally declares a set method in the derived class.
- Note that you can use the override keyword as part of the property declaration in a primary constructor:

Overriding Properties

Kotlin

```
interface Shape {
    val vertexCount: Int
}

class Rectangle(override val vertexCount: Int = 4) : Shape // Always has 4 vertices

class Polygon : Shape {
    override var vertexCount: Int = 0  // Can be set to any number later
}
```

Derived Class Initialization Order

Kotlin

- During the construction of a new instance of a derived class, the base class initialization is done as the first step (preceded only by evaluation of the arguments for the base class constructor).
- It means that it happens before the initialization logic of the derived class is run.

Derived Class Initialization Order

Kotlin

```
//sampleStart
open class Base(val name: String) {

    init { println("Initializing a base class") }

    open val size: Int =
        name.length.also { println("Initializing size in the base class: $it") }
}

class Derived(
    name: String,
    val lastName: String,
) : Base(name.replaceFirstChar { it.uppercase() }.also { println("Argument for the base class: $it") }) {

    init { println("Initializing a derived class") }

    override val size: Int =
        (super.size + lastName.length).also { println("Initializing size in the derived class: $it") }
}
//sampleEnd

fun main() {
    println("Constructing the derived class(\"hello\", \"world\")")
    Derived("hello", "world")
}
```

Derived Class Initialization Order

Kotlin

- This means that when the base class constructor is executed, the properties declared or overridden in the derived class have not yet been initialized.
- Using any of those properties in the base class initialization logic (either directly or indirectly through another overridden open member implementation) may lead to incorrect behavior or a runtime failure.
- When designing a base class, we should therefore avoid using open members in the constructors, property initializers, or init blocks.

Calling the superclass implementation

Kotlin

```
open class Rectangle {
    open fun draw() { println("Drawing a rectangle") }
    val borderColor: String get() = "black"
}

class FilledRectangle : Rectangle() {
    override fun draw() {
        super.draw()
        println("Filling the rectangle")
    }

    val fillColor: String get() = super.borderColor
}
```

Calling the superclass implementation

Kotlin

```
open class Rectangle {
    open fun draw() { println("Drawing a rectangle") }
    val borderColor: String get() = "black"
}

//sampleStart
class FilledRectangle: Rectangle() {
    override fun draw() {
        val filler = Filler()
        filler.drawAndFill()
    }

    inner class Filler {
        fun fill() { println("Filling") }
        fun drawAndFill() {
            super@FilledRectangle.draw() // Calls Rectangle's implementation of draw()
            fill()
            println("Drawn a filled rectangle with color ${super@FilledRectangle.borderColor}") // Uses
Rectangle's implementation of borderColor's get()
        }
    }
}
//sampleEnd

fun main() {
    val fr = FilledRectangle()
        fr.draw()
}
```

Interface

Kotlin

- Interfaces are custom types provided by Kotlin that cannot be instantiated directly.
- Instead, these define a form of behavior that the implementing types must follow.
- With the interface, we can define a set of properties and methods, that the concrete types must follow and implement.

```
interface Vehicle()
{
  fun start()
  fun stop()
}
```

Interface

Kotlin

- Interfaces are custom types provided by Kotlin that cannot be instantiated directly.
- Instead, these define a form of behavior that the implementing types must follow.
- With the interface, we can define a set of properties and methods, that the concrete types must follow and implement.

```
interface Vehicle()
{
  fun start()
  fun stop()
}
```

Interface

```
interface Vehicle {
    fun start()
    fun stop()
}

class Car : Vehicle {
    override fun start()
    {
        println("Car started")
    }

    override fun stop()
    {
        println("Car stopped")
    }
}
```

```
fun main()
{
    val obj = Car()
    obj.start()
    obj.stop()
}
```

Properties in Interfaces

- We can declare properties in interfaces.
- A property declared in an interface can either be abstract or provide implementations for accessors.
- Properties declared in interfaces can't have backing fields, and therefore accessors declared in interfaces can't reference them.

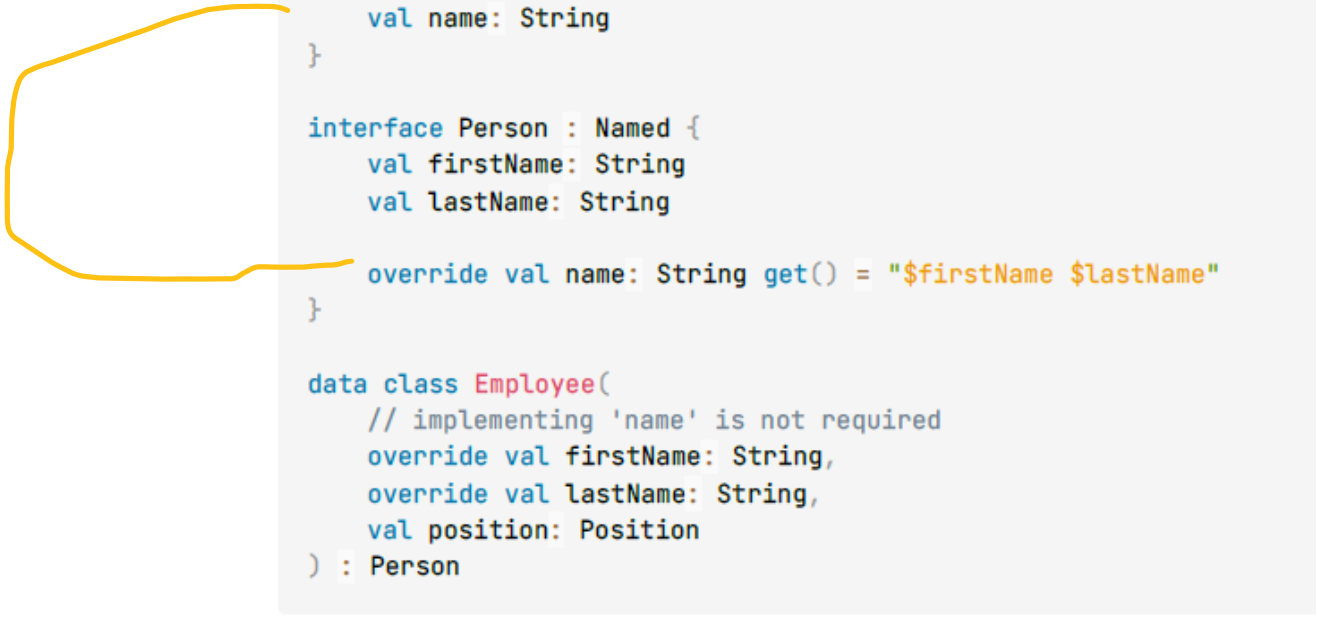
```
interface InterfaceProperties {
    val a : Int
    val b : String
        get() = "Hello"
}
class PropertiesDemo : InterfaceProperties {
    override val a : Int = 5000
    override val b : String = "Property Overridden"
}
```

```
fun main()
{
    val x = PropertiesDemo()
    println(x.a)
    println(x.b)
}
```

Interface Inheritance

Kotlin

- An interface can derive from other interfaces, meaning it can both provide implementations for their members and declare new functions and properties.
- Classes implementing such an interface are only required to define the missing implementations



```
interface Named {
    val name: String
}

interface Person : Named {
    val firstName: String
    val lastName: String

    override val name: String get() = "$firstName $lastName"
}

data class Employee(
    // implementing 'name' is not required
    override val firstName: String,
    override val lastName: String,
    val position: Position
) : Person
```

Resolving overriding conflicts

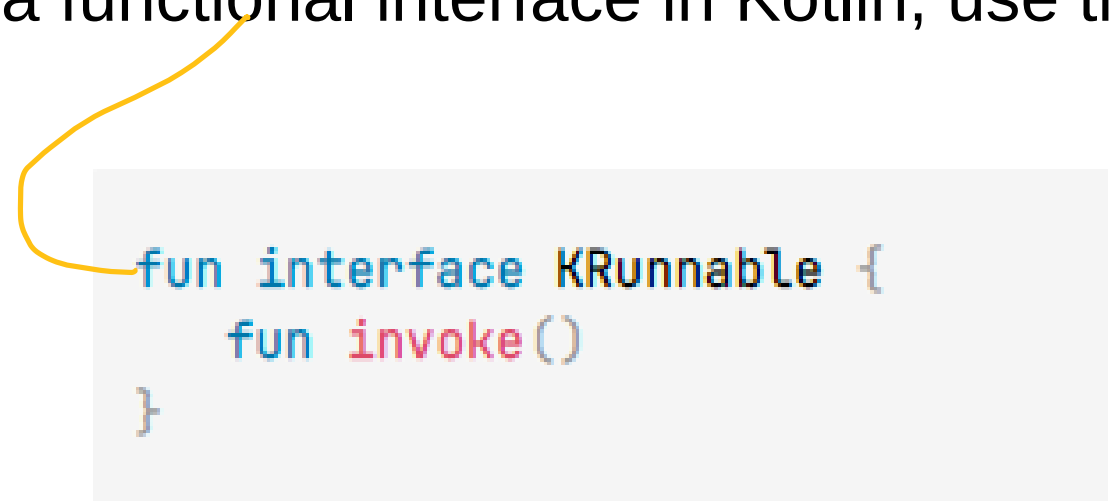
Kotlin

```
interface A {
    fun foo() { print("A") }
    fun bar()
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() { print("bar") }
}

class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }

    override fun bar() {
        super<B>.bar()
    }
}
```

Functional (SAM) interfaces

Kotlin

- An interface with only one abstract method is called a functional interface, or a Single Abstract Method (SAM) interface.
- The functional interface can have several non-abstract members but only one abstract member.
- To declare a functional interface in Kotlin, use the fun modifier.



```
fun interface KRunnable {
    fun invoke()
}
```

Functional (SAM) interfaces

Kotlin

- An interface with only one abstract method is called a functional interface, or a Single Abstract Method (SAM) interface.
- The functional interface can have several non-abstract members but only one abstract member.
- To declare a functional interface in Kotlin, use the fun modifier.

```
fun interface IntPredicate {
    fun accept(i: Int): Boolean
}

val isEven = IntPredicate { it % 2 == 0 }

fun main() {
    println("Is 7 even? - ${isEven.accept(7)}")
}
```

Singleton Object

Kotlin

```
object Singleton {
    init {
        println("Singleton initialized")
    }

    var message = "Kotlin rock"

    fun showMessage() {
        println(message)
    }
}

class Test {
    init {
        Singleton.showMessage()
    }
}

fun main() {
    Singleton.showMessage()
    Singleton.message = "Kotlin is cool"

    val test = Test()
}
```