The Ultimate Guide to Successfully Adopting Kotlin in a Java-Dominated Environment

> *“You don’t flip a switch to ‘go Kotlin’.*
> *You de-risk, you measure, you celebrate the wins,*
> *and only then do you double down.”*

Adopting Kotlin in an established Java environment isn't just a technical decision – it's a journey that requires careful planning, strategic thinking, and most importantly, winning the hearts and minds of your peers.

After training over 1,000 developers and helping numerous organizations successfully transition to Kotlin, I've seen what works and what doesn't. This guide will walk you through successful recipes for adopting Kotlin, which I have collected over time, from your first experimental playground to large-scale organizational transformation.

This is the journey this blog post will take you through:

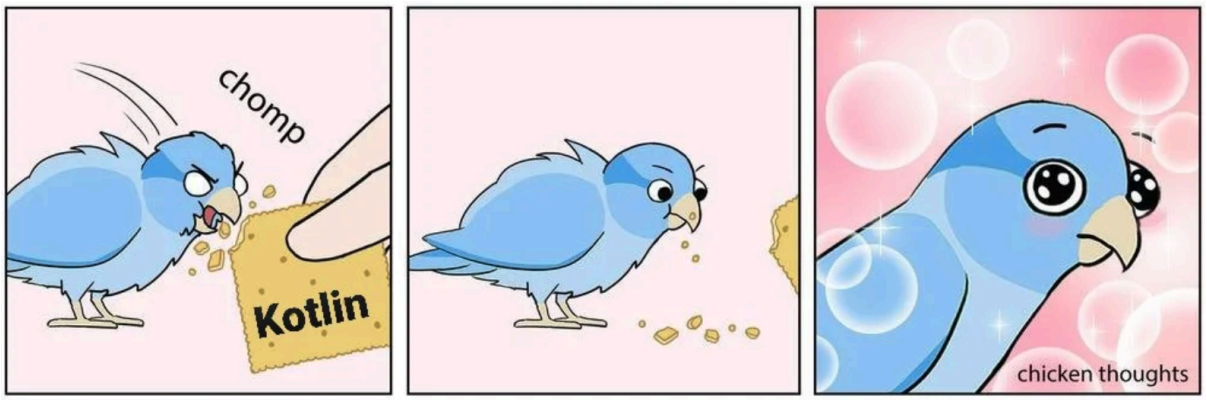
Urs Peter: Senior Software Engineer and JetBrains certified Kotlin Trainer

Urs is a seasoned software engineer, solution architect, conference speaker, and trainer with over 20 years of experience in building resilient, scalable, and mission-critical systems, mostly involving Kotlin and Scala.

Besides his job as a consultant, he is also a passionate trainer and author of a great variety of courses ranging from language courses for Kotlin and Scala to architectural trainings such as Microservices and Event-Driven Architectures.

As a people person by nature, he loves to share knowledge and inspire and get inspired by peers on meetups and conferences. Urs is a JetBrains certified Kotlin trainer.

1. It Always Starts With a Spark, Initiated by You!



Why take the effort to switch to Kotlin? Why not just stick with Java and move on?

The answer depends on numerous factors. While the data clearly shows Kotlin's advantages across multiple domains, the decision isn't purely technical. Subjectivity (“I like my language because I like it”) and skepticism towards something new, which is generally a good thing, play an important role.

However, the evolution of programming languages shows that our preferences and needs change over time. Crucially, each new generation of languages incorporates fresh paradigms – (null)-safety, concise and light-weight syntax, functions as first class citizens, rich standard library, async concurrency, multiplatform support, generative-AI friendly, etc. – that give developers and organizations a decisive advantage.

Without this natural progression, we’d still be coding everything in COBOL or another archaic language, unable to meet today’s demands. Evolution is therefore not optional; it’s built into the history of our industry.

For this evolution to take root inside a company, though, it takes more than technical merit. It requires enablers – people willing to explore, advocate, and show the value of these new paradigms in practice. In my experience, three types of engineers typically become these catalysts for Kotlin adoption:

1. **The pragmatic, productivity-focused Java Engineer**: Experienced developers who see Java as a tool, not a religion. They're always looking for better ways to get the job done faster.
2. **The quality-minded, modern language enthusiast**: Engineers who prioritize readable, concise, and maintainable code. These are often the same people who would have moved to Scala in the past.
3. **Junior Developers**: Juniors who ask the simple but powerful question: "Why should I use Java if I can use Kotlin?" Without the baggage of years of Java experience, Kotlin is often a no-brainer for them.

To which group do you belong?

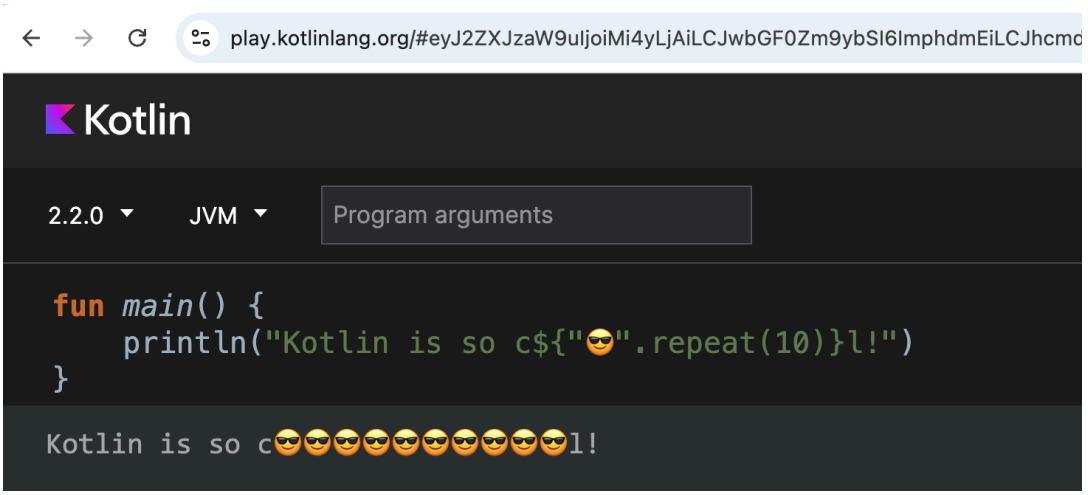
These early adopters ignite the first stage. But how do you start? Read on… ;-)

2. The Play Around Stage: Start Small – With Tests

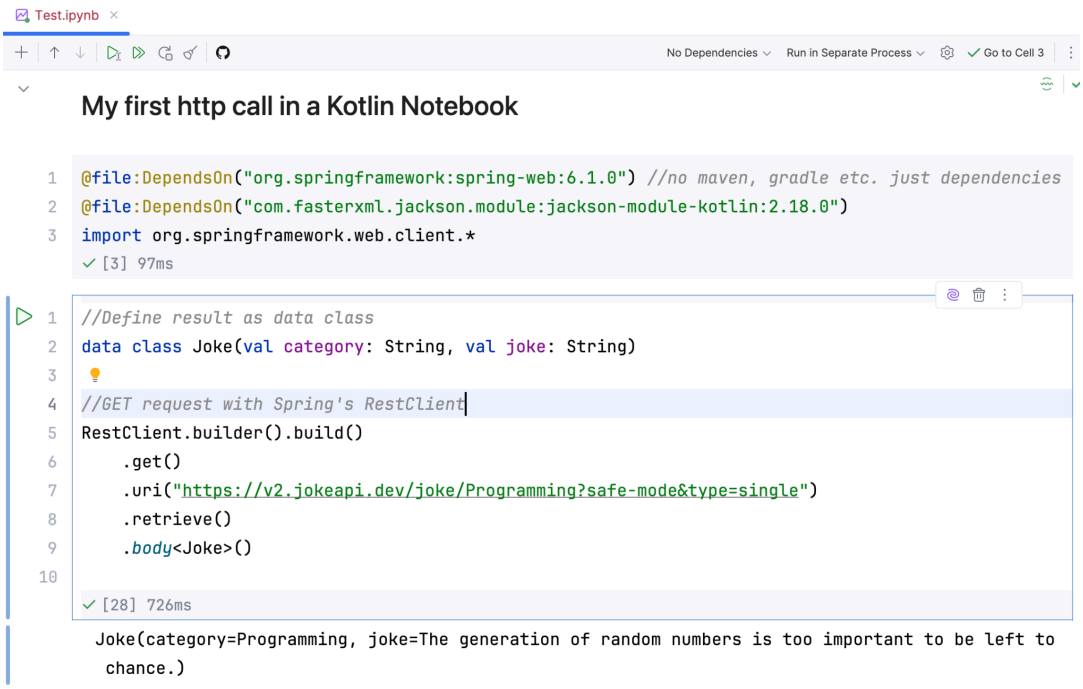
You've heard about Kotlin and want to give it a try without committing to it immediately.

So the first thing you need is a developer tool where your very first Kotlin seeds can be planted. Here are some options:

- https://play.kotlinlang.org/ is a great online playground, simply type and run Kotlin code not only on the JVM but also for various other platforms (JS, WASM, etc.).

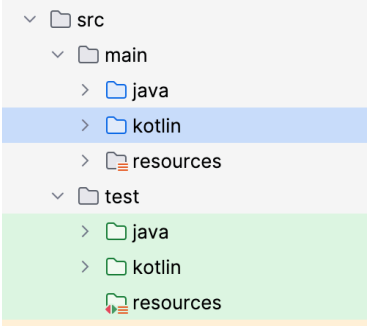


- Kotlin Notebook is a powerful feature for IntelliJ IDEA that allows you to easily import dependencies, execute code, and even work with data, draw graphs, etc. Here is an example that shows how easy it is to do a REST call with Spring's RestClient:



- IntelliJ IDEA has first-class support for Kotlin. This is no surprise, since JetBrains is the maker of Kotlin, and a large portion of IntelliJ IDEA is written in it. So, to start using Kotlin in IntelliJ IDEA – even in your existing Java project – is a breeze:
    - For Maven, simply configure the `kotlin-maven-plugin` and the Kotlin standard library `kotlin-stdlib`.
    - For Gradle, you configure the kotlin plugin.

  …and off you go!



- And there's more! JetBrains recently released the Kotlin Language Server, bringing a full-featured Kotlin development experience to other IDEs beyond IntelliJ IDEA – such as VS Code. Check it out: https://github.com/Kotlin/kotlin-lsp

Now, you can write Kotlin in your favorite development environment. How can you evaluate the language in a real-world context with minimal impact and maximal insights? In the test suite of an existing Java project!

This safe and realistic approach to experimenting with Kotlin offers several advantages:

- **Low Risk**: Tests don't affect production code.
- **Learning Opportunity**: You can explore the language features in a familiar context.
- **Gradual Introduction**: Team members can get comfortable with Kotlin syntax without pressure.

Tips

1. Try Kotest + MockK: to immediately feel the expressiveness of Kotlin's testing DSLs (Domain-Specific Language), like the feature-rich assertions (`shouldHaveSize(...)`, infix (`value shouldBe 1`), etc.
2. Use the concise and powerful Kotlin Collections rather than Java Streams.
3. Play with various language features like Nullable types, destructuring, immutability (`val`, `data classes`), expression constructs (`when`, `try-catch`, `if else`), and many more.

This is what you get:

Java

```java
@Test
void shouldGetAverageRating() {
   when(productRepository.findAll()).thenReturn(products);

   Map<String, Double> ratings = productService.averageRatings();

   assertAll(
           () -> assertThat(ratings).hasSize(4),
           () -> assertEquals(ratings, products
                   .stream()
                   .collect(Collectors.groupingBy(
                           Product::getName,
                           Collectors.flatMapping(
                                   p -> p.getRatings()
                                    .stream()
                                    .mapToDouble(Integer::doubleValue)
                                    .boxed(),
                              Collectors.averagingDouble(Double::doubleVal
                              ue)
                           )))
           )
   );
   verify(productRepository).findAll();

}
```

Kotlin

```kotlin
@Test
fun `should get average rating`() { //descriptive tests using ``
   every { productRepository.findAll() } returns products

   val ratings = productService.averageRatings()

   assertSoftly(ratings) {          //powerful testing DSLs (Kotest)
       shouldHaveSize(4)
       this shouldBe productRepository.findAll()
           .groupBy { it.name }                 //concise collections
           .mapValues { (_, products) ->        //destructuring
                  products.flatMap { it.ratings }.average() }
   }
   verify { productRepository.findAll() }
}
```

3. The Evaluation Stage: Beyond Kotlin as a Playground

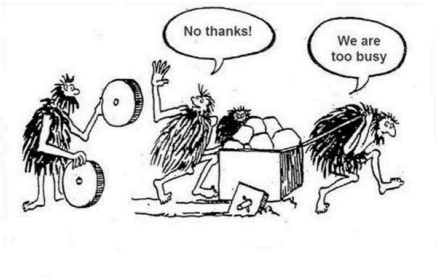
Once you're comfortable with Kotlin in tests, it's time for a more substantial evaluation. You have two main approaches:

- Build a New Micro Service / Application in Kotlin
- Extend/Convert Existing Java Applications

1. Build a new microservice/application in Kotlin

Starting fresh with a new application or microservice provides the full Kotlin experience without the constraints of legacy code. This approach often provides the best learning experience and showcases Kotlin's strengths most clearly.

> **Pro tip**: Get expert help during this stage. While developers are naturally confident in their abilities, avoiding early mistakes in the form of Java-ish Kotlin and a lack of Kotlin-powered libraries can save months of technical debt.



This is how you can avoid common pitfalls when using Kotlin from a Java background:

Pitfall: Choosing a different framework from the one you use in Java.

Tip: **Stick to your existing framework**.

Most likely, you were using Spring Boot with Java, so use it with Kotlin too. Spring Boot Kotlin support is first-class, so there is no additional benefit in using something else. Moreover, you are forced to learn not only a new language but also a new framework, which only adds complexity without providing any advantage.

Important: Spring interferes with Kotlin's 'inheritance by design' principle, which requires you to explicitly mark classes `open` in order to extend them.

In order to avoid adding the `open` keyword to all Spring-related classes (like `@Configuration`, etc.), use the following build plugin: https://kotlinlang.org/docs/all-open-plugin.html#spring-support. If you create a Spring project with the well-known online Spring initializr tool, this build plugin is already configured for you.

Pitfall 1: Writing Kotlin in a Java-ish way, relying on common Java APIs rather than Kotlin's standard library:

This list can be very long, so let's focus on the most common pitfalls:

Pitfall 1: Using Java Stream rather than Kotlin Collections

Tip: **Always use Kotlin Collections.**

Kotlin Collections are fully interoperable with Java Collections, yet equipped with straightforward and feature-rich higher-order functions that make Java Stream obsolete.

As follows is an example that aims to pick the top 3 products by revenue (price * sold) grouped by product category:

Java

```
record Product(String name, String category, double price, int sold){}

List<Product> products = List.of(
           new Product("Lollipop", "sweets", 1.2, 321),
           new Product("Broccoli", "vegetable", 1.8, 5);

Map<String, List<Product>> top3RevenueByCategory =
       products.stream()
          .collect(Collectors.groupingBy(
                Product::category,
                Collectors.collectingAndThen(
                    Collectors.toList(),
                    list -> list.stream()
                              .sorted(Comparator.comparingDouble(
                                  (Product p) -> p.price() * p.sold())
                                   .reversed())
                                   .limit(3)
                                   .toList()
                              )
                )
           );
```

Kotlin

```
val top3RevenueByCategory: Map<String, List<Product>> =
   products.groupBy { it.category }
       .mapValues { (_, list) ->
           list.sortedByDescending { it.price * it.sold }.take(3)
       }
```

Kotlin Java interop lets you work with Java classes and records as if they were native Kotlin, though you could also use a Kotlin (data) class instead.

Pitfall 2: Keeping on using Java's Optional.

Tip: **Embrace Nullable types**.

One of the key reasons Java developers switch to Kotlin is for Kotlin's built-in nullability support, which waves NullPointerExceptions goodbye. Therefore, try to use Nullable types only, no more Optionals. Do you still have Optionals in your interfaces? This is how you easily get rid of them by converting them to Nullable types:

Kotlin

```
//Let's assume this repository is hard to change, because it's a library
you depend on
class OrderRepository {
      //it returns Optional, but we want nullable types
      fun getOrderBy(id: Long): Optional<Order> = ...
}

//Simply add an extension method and apply the orElse(null) trick
fun OrderRepository.getOrderByOrNull(id: Long): Order? =
                                    getOrderBy(id).orElse(null)

//Now enjoy the safety and ease of use of nullable types:

//Past:
 val g = repository.getOrderBy(12).flatMap { product ->
     product.goody.map { it.name }
}.orElse("No goody found")

//Future:
 val g = repository.getOrderByOrNull(12)?.goody?.name ?: "No goody found"
```

Pitfall 3: Continuing to use static wrappers.

Tip: **Embrace Extension methods**.

Extension methods give you many benefits:

- They make your code much more fluent and readable than wrappers.
- They can be found with code completion, which is not the case for wrappers.
- Because Extensions need to be imported, they allow you to selectively use extended functionality in a specific section of your application.

Java

```
//Very common approach in Java to add additional helper methods
public class DateUtils {
      public static final DateTimeFormatter DEFAULT_DATE_TIME_FORMATTER =
                  DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

      public String formatted(LocalDateTime dateTime,
                              DateTimeFormatter formatter) {
         return dateTime.format(formatter);
      }

      public String formatted(LocalDateTime dateTime) {
         return formatted(dateTime, DEFAULT_DATE_TIME_FORMATTER);
      }
}

//Usage
 formatted(LocalDateTime.now());
```

Kotlin

```
val DEFAULT_DATE_TIME_FORMATTER: DateTimeFormatter =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")

//Use an extension method, including a default argument, which omits the
need for an overloaded method.
fun LocalDateTime.asString(
   formatter: DateTimeFormatter = DEFAULT_DATE_TIME_FORMATTER): String =
      this.format(formatter)

//Usage
LocalDateTime.now().formatted()
```

Be aware that Kotlin offers top-level methods and variables. This implies that we can simply declare e.g. the `DEFAULT_DATE_TIME_FORMATTER` top level without the need to bind to an object like is the case in Java.

Pitfall 4: Relying on (clumsily) Java APIs

Tip: **Use Kotlin's slick counterpart.**

The Kotlin standard library uses extension methods to make Java libraries much more user-friendly, even though the underlying implementation is still Java. Almost all major third-party libraries and frameworks, like Spring, have done the same.

Example standard library:

Java

```
String text;
try (
       var reader = new BufferedReader(
                     new InputStreamReader(new FileInputStream("out.txt"),
              StandardCharsets.UTF_8))) {
   text = reader
              .lines()
              .collect(Collectors.joining(System.lineSeparator()));
}
System.out.println("Downloaded text: " +  text + "\n");
```

Kotlin

```
//Kotlin has enhanced the Java standard library with many powerful extension
methods, like on java.io.*, which makes input stream processing a snap due to its
fluent nature, fully supported by code completion

val text = FileInputStream("path").use {
              it.bufferedReader().readText()
            }
println("Downloaded text: $text\n");
```

Example Spring:

Java

```
final var books =  RestClient.create()
       .get()
       .uri("http://.../api/books")
       .retrieve()
       .body( new ParameterizedTypeReference<List<Book>>(){}); // ⇦ inconvenient
ParameterizedTypeReference
```

Kotlin

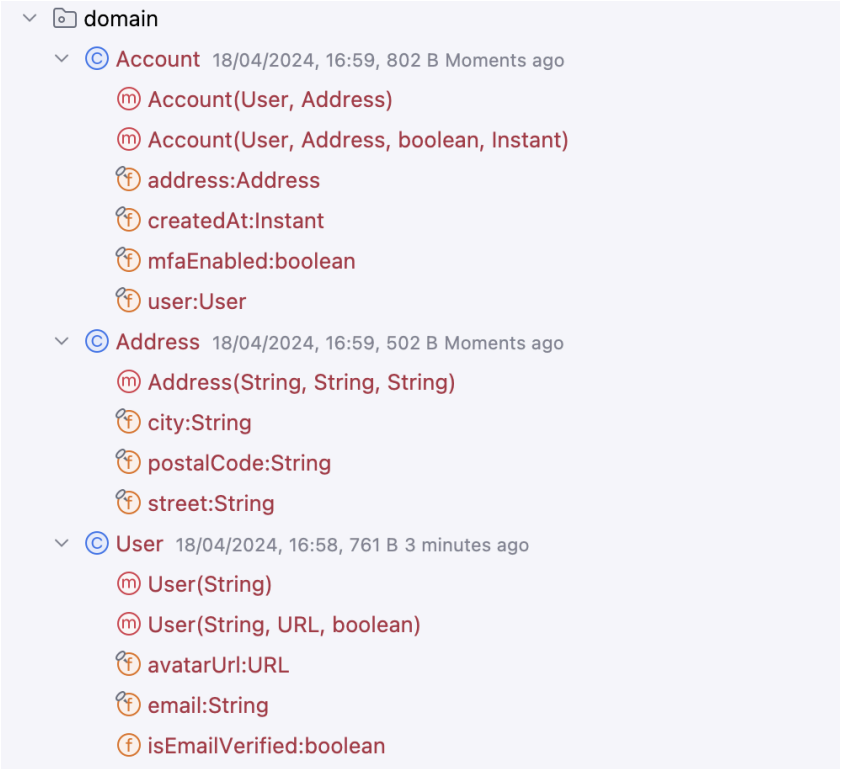
```
import org.springframework.web.client.body
val books = RestClient.create()
   .get()
   .uri("http://.../api/books")
   .retrieve()
   .body<List<Book>>() //⇦ Kotlin offers an extension that only requires the type
without the need for a ParameterizedTypeReference
```

Pitfall 5: Using a separate file for each public class

Tip: **Combine related public classes in a single file.**

This allows you to get a good understanding of how a (sub-)domain is structured without having to navigate dozens of files.

Java



File: `Model.kt`

Kotlin

```kotlin
//For domain classes consider data classes - see why below
data class User(val email: String,
            //Use nullable types for safety and expressiveness
           val avatarUrl: URL? = null,
           var isEmailVerified: Boolean)

data class Account(val user:User,
              val address: Address,
              val mfaEnabled:Boolean,
              val createdAt: Instant)

data class Address(val street: String,
              val city: String,
              val postalCode: String)
```

Pitfall 6: Relying on the mutable programming paradigm

Tip: **Embrace immutability – the default in Kotlin**

The trend across many programming languages – including Java – is clear: immutability is winning over mutability.

The reason is straightforward: immutability prevents unintended side effects, making code safer, more predictable, and easier to reason about. It also simplifies concurrency, since immutable data can be freely shared across threads without the risk of race conditions.

That's why most modern languages – Kotlin among them – either emphasize immutability by default or strongly encourage it. In Kotlin, immutability is the default, though mutability remains an option when truly needed.

Here's a quick guide to Kotlin's **immutability power pack**:

1. Use `val` over `var`

Prefer `val` over `var`. IntelliJ IDEA will notify you if you used a `var`, for which a `val` could be used.

2. Use (immutable) data classes with `copy(...)`

For domain-related classes, use `data` classes with `val`.Kotlin `data` classes are often compared with Java `records`. Though there is some overlap, `data` classes offer the killer feature `copy(...)`, whose absence makes transforming `record` – which is often needed in business logic – so tedious:

Java

```java
//only immutable state
public record Person(String name, int age) {
   //Lack of default parameters requires overloaded constructor
   public Person(String name) {
       this(name, 0);
   }
   //+ due to lack of String interpolation
   public String sayHi() {
       return "Hello, my name is " + name + " and I am " + age + " years
old.";
   }
}

//Usage
final var jack = new Person("Jack", 42);
jack: Person[name=Jack, age=5]

//The issue is here: transforming a record requires manually copying the
identical state to the new instance ☹
final var fred = new Person("Fred", jack.age());
```

Kotlin

```kotlin
//also supports mutable state (var)
data class Person(val name: String,
                  val age: Int = 0) {

   //string interpolation
   fun sayHi() = "Hi, my name is $name and I am $age years old."
}
val jack = Person("Jack", 42)
jack: Person(name=Jack, age=42)

//Kotlin offers the copy method, which, due to the 'named argument'
feature, allows you to only adjust the state you want to change 😃
val fred = jack.copy(name = "Fred")
fred: Person(name=Fred, age=42)
```

Moreover, use data classes for domain-related classes whenever possible. Their immutable nature ensures a safe, concise, and hassle-free experience when working with your application's core.

3. Prefer Immutable over Mutable Collections

Immutable Collections have clear benefits regarding thread-safety, can be safely passed around, and are easier to reason about. Although Java collections offer some immutability features for Collections, their usage is dangerous because it easily causes exceptions at runtime:

Java

```
List.of(1,2,3).add(4); ❌unsafe 😬! .add(...) compiles, but throws UnsupportedOperationException
```

Kotlin

```
//The default collections in Kotlin are immutable (read-only)
listOf(1,2,3).add(4);  //✅safe: does not compile

val l0 = listOf(1,2,3)
val l1 = l0 + 4 //✅safe: it will return a new list containing the added element
l1 shouldBe listOf(1,2,3,4) //✅
```

The same applies for using `Collections.unmodifiableList(...),` which is not only unsafe, but also requires extra allocation:

Java

```
class PersonRepo {
   private final List<Person> cache = new ArrayList<>();
   // Java - must clone or wrap every call
   public List<Person> getItems() {
       return Collections.unmodifiableList(cache);   //⚠️extra alloc
   }
}

//Usage
personRepo.getItems().add(joe) ❌unsafe 😬! .add(...) can be called but throws UnsupportedOperationException
```

Kotlin

```
class PersonRepo {

//The need to type 'mutable' for mutable collections is intentional: Kotlin wants you to use immutable ones by default. But sometimes you need them:

   private val cache: MutableList<Person> = mutableListOf<Person>()

   fun items(): List<Person> = cache //✅safe: though the underlying collection is mutable, by returning it as its superclass List<...>, it only exposes the read-only interface

}

//Usage
personRepo.items().add(joe) //✅safe:😬! Does not compile
```

When it comes to concurrency, immutable data structures, including collections, should be preferred. In Java, more effort is required with special Collections that offer a different or limited API, like `CopyOnWriteArrayList`. In Kotlin, on the other hand, the read-only `List<...>` does the job for almost all use cases.

If you need mutable, Thread-Safe Collections, Kotlin offers Persistent Collections (`persistentListOf(...)`, `persistentMapOf(...)`), which all share the same powerful interface.

Java

```
ConcurrentHashMap<String, Integer> persons = new ConcurrentHashMap<>();
persons.put("Alice", 23);
persons.put("Bob",   21);

//not fluent and data copying going on
Map<String, Integer> incPersons = new HashMap<>(persons.size());
persons.forEach((k, v) -> incPersons.put(k, v + 1));

//wordy and data copying going on
persons
   .entrySet()
   .stream()
   .forEach(entry ->
      entry.setValue(entry.getValue() + 1));
```

Kotlin

```
persistentMapOf("Alice" to 23, "Bob" to 21)
         .mapValues { (key, value) -> value + 1 } //✅same rich API like
any other Kotlin Map type and not data copying going on
```

Pitfall 7: Keeping on using builders (or even worse: trying to use Lombok)

Tip: **Use named arguments.**

Builders are very common in Java. Although they are convenient, they add extra code, are unsafe, and increase complexity. In Kotlin, they are of no use, as a simple language feature renders them obsolete: named arguments.

Java

```java
public record Person(String name, int age) {

    // Builder for Person
    public static class Builder {
        private String name;
        private int age;

        public Builder() {}

        public Builder name(String name) {
            this.name = name;
            return this;
        }

        public Builder age(int age) {
            this.age = age;
            return this;
        }

        public Person build() {
            return new Person(name, age);
        }
    }
}

//Usage
new JPerson.Builder().name("Jack").age(36).build(); //compiles and
succeeds at runtime

new JPerson.Builder().age(36).build(); //❌unsafe 😬: compiles but fails
at runtime.
```

Kotlin

```kotlin
data class Person(val name: String, val age: Int = 0)

//Usage - no builder, only named arguments.
Person(name = "Jack") //✅safe: if it compiles, it always succeeds at
runtime
Person(name = "Jack", age = 36) //✅
```

2. Extend/convert an existing Java application

If you have no greenfield option for trying out Kotlin, adding new Kotlin features or whole Kotlin modules to an existing Java codebase is the way to go. Thanks to Kotlin's seamless Java interoperability, you can write Kotlin code that looks like Java to Java callers. This approach allows for:

- Gradual migration without big-bang rewrites
- Real-world testing of Kotlin in your specific context
- Building team confidence with production Kotlin code

Rather than starting *somewhere*, consider these different approaches:

Outside-in:

Start in the “leaf” section of your application, e.g. controller, batch job, etc. and then work your way towards the core domain. This will give you the following advantages:

- **Compile-time isolation**: Leaf classes rarely have anything depending *on them*, so you can flip them to Kotlin and still build the rest of the system unchanged.
- **Fewer ripple edits.** A converted UI/controller can call existing Java domain code with almost no changes thanks to seamless interop.
- **Smaller PRs, easier reviews.** You can migrate file-by-file or feature-by-feature.

Inside-out:

Starting at the core and then moving to the outer layers is often a riskier approach, as it compromises the advantages of the outside-in approach mentioned above. However, it is a viable option in the following cases:

- **Very small or self-contained core.** If your domain layer is only a handful of POJOs and services, flipping it early may be cheap and immediately unlock idiomatic constructs (data class, value classes, sealed hierarchies).
- **Re-architecting anyway.** If you plan to refactor invariants or introduce DDD patterns (value objects, aggregates) while you migrate, it’s sometimes cleaner to redesign the domain in Kotlin first.
- **Strict null-safety contracts.** Putting Kotlin at the center turns the domain into a “null-safe fortress”; outer Java layers can still send null, but boundaries become explicit and easier to police.

Module by module

- If your architecture is organized by functionality rather than layers, and the modules have a manageable size, converting them one by one is a good strategy.

Language features for converting Java to Kotlin

Kotlin offers a variety of features – primarily annotations – that allow your Kotlin code to behave like native Java. This is especially valuable in hybrid environments where Kotlin and Java coexist within the same codebase.

Kotlin

```kotlin
class Person @JvmOverloads constructor(val name: String,
                           var age: Int = 0) {
  companion object {

   @JvmStatic
   @Throws(InvalidNameException::class)
   fun newBorn(name: String): Person = if (name.isEmpty())
       throw InvalidNameException("name not set")
     else Person(name, 0)


   @JvmField
   val LOG = LoggerFactory.getLogger(KPerson.javaClass)
  }
}
```

Java

```java
//thanks to @JvmOverloads an additional constructor is created, propagating
Kotlin's default arguments to Java
var john =  new Person("John");

//Kotlin automatically generates getters (val) and setters (var) for Java
john.setAge(23);
var name = ken.getName();

//@JvmStatic and @JvmField all accessing (companion) object fields and methods as
statics in Java

//Without @JvmStatic it would be: Person.Companion.newBorn(...)
var ken =  Person.newBorn("Ken");

//Without @JvmField it would be: Person.Companion.LOG
Person.LOG.info("Hello World, Ken ;-)");

//@Throws(...) will put the checked Exception in the method signature
try {
   Person ken =  Person.newBorn("Ken");
} catch (InvalidNameException e) {
   //…
}
```

Kotlin

```kotlin
@file:JvmName("Persons")
package org.abc

@JvmName("prettyPrint")
fun Person.pretty() =
      Person.LOG.info("$name is $age old")
```

Java

```java
//@JvmName for files and methods makes accessing static fields look like
Java: without it would be: PersonKt.pretty(...)
Persons.prettyPrint(ken)
```

IntelliJ IDEA's Java to Kotlin Converter

IntelliJ IDEA offers a Java to Kotlin Converter, so theoretically, the tool can do it for you. However, the resulting code is far from perfect, so use it only as a starting point. From there, convert it to a more Kotlin-esque representation. More on this topic will be discussed in the final section of this blog post: *Success Factors for Large-Scale Kotlin Adoption*.

Taking Java as a starting point will most likely make you write Java-ish Kotlin, which gives you some benefits, but will not unleash the power of Kotlin's potential. Therefore, writing a new application is the approach I prefer.

4. Spread the Word: Win the Hearts and Minds of your Fellow Developers

By now, you have a core team convinced of Kotlin's benefits. Now comes the critical phase: How do you expand adoption?

Key in this phase is to win the hearts and minds of skeptical Java developers. Hard and soft factors can make the difference here:

Hard factors: the code

- Let the code speak

Soft factors: support and connect developers

- Facilitate easy onboarding
- Offer self-study material
- Establish an in-house Kotlin community
- Be patient…

Let the code speak

Double down on the experience you gained by (re-)writing an application in Kotlin and show the benefits in a tangible, accessible way:

- **Show, don't tell**: present the gained conciseness of Kotlin by comparing Java with Kotlin snippets.
- **Zoom out to focus on the overarching paradigms**: Kotlin is not just *different* from Java: Kotlin is built on the foundation of safety, conciseness for maximal readability and maintainability, functions as a first-class citizen, and extensibility, which together solve fundamental shortcomings of Java, while still being fully interoperable with the Java ecosystem.

Here are some tangible examples:

1. Null-safety: No more billion-dollar mistakes

Java

```
//This is why we have the billion-dollar mistake (not only in Java…)
Booking booking = null; //🤨 This is allowed but causes:
booking.destination; //Runtime error 😱

Optional<Booking> booking = null; //Optionals aren't save from null either:
🤨
booking.map(Destination::destination); //Runtime error 😱
```

Java has added some null-safety features over time, like Optional and Annotations (@NotNull etc.), but it has failed to solve this fundamental problem. Additionally, project Valhalla (Null-Restricted and Nullable types) will not introduce null-safety to Java, but rather provide more options to choose from.

Kotlin

```
//Important to realize that null is very restricted in Kotlin:
val booking:Booking? = null //…null can only be assigned to Nullable types
✅
val booking:Booking = null //null assigned to a Non-nullable types yield a
Compilation error 😃

booking.destination //unsafely accessing a nullable type directly causes a
Compilation error 😃
booking?.destination //only safe access is possible  ✅
```

The great thing about Kotlin's null-safety is that it is not only safe, but also offers great usability. A classic example of "have your cake and eat it, too":

Say we have this Domain:

Kotlin

```kotlin
data class Booking(val destination:Destination? = null)
data class Destination(val hotel:Hotel? = null)
data class Hotel(val name:String, val stars:Int? = null)
```

Java

```java
public record Booking(Optional<Destination> destination) {

    public Booking() { this(Optional.empty()); }

    public Booking(Destination destination) {
        this(Optional.ofNullable(destination));
    }
}

public record Destination(Optional<Hotel> hotel) {

    public Destination() { this(Optional.empty()); }

    public Destination(Hotel hotel) {
        this(Optional.ofNullable(hotel));
    }
}

public record Hotel(String name, Optional<Integer> stars) {

    public Hotel(String name) {
        this(name, Optional.empty());
    }

    public Hotel(String name, Integer stars) {
        this(name, Optional.ofNullable(stars));
    }
}
```

Constructing objects

Java

```
//Because Optional is a wrapper, the number of nested objects grows, which
doesn't help readability
final Optional<Booking> booking = Optional.of(new Booking(
      Optional.of(new Destination(Optional.of(
            new Hotel("Sunset Paradise", 5))))));
```

Kotlin

```
//Since nullability is part of the type system, no wrapper is needed: The
required type or null can be used.
val booking:Booking? = Booking(Destination(Hotel("Sunset Paradise", 5)))
```

Traversing nested objects

Java

```
//traversing a graph of Optionals requires extensive unwrapping
final var stars = "*".repeat(booking
                        .flatMap(Booking::getDestination)
                        .flatMap(Destination::getHotel)
                        .map(Hotel::getStars).orElse(0)); //-> "*****"
```

Kotlin

```
//Easily traverse a graph of nullable types with: '?', use  ?: for the
'else' case.
val stars = "*".repeat(booking?.destination?.hotel?.stars ?: 0) //-> "*****"
```

Unwrap nested object

Java

```
//extensive unwrapping is also needed for printing a leaf
booking.getDestination()
       .flatMap(Destination::getHotel)
       .map(Hotel::getName)
       .map(String::toUpperCase)
         .ifPresent(System.out::println);
```

Kotlin

```
//In Kotlin we have two elegant options:
//1. we can again traverse the graph with '?'
booking?.destination?.hotel.?name?.uppercase()?.also(::println)

//2. We can make use of Kotlin's smart-cast feature
if(booking?.destination?.hotel != null) {
   //The compiler has checked that all the elements in the object graph are
not null, so we can access the elements as if they were non-nullable types
   println(booking.destination.hotel.uppercase())
}
```

The lack of null-safety support in Java is a key pain-point for developers, leading to defensive programming, different nullability constructs (or none at all), including increased verbosity. Moreover, NullPointerExceptions are responsible for roughly one-third of application crashes (The JetBrains Blog). Kotlin's compile-time null checking prevents these runtime failures entirely. That's why, still today, **Null-safety is the primary driver for migration to Kotlin**.

2. Collections are your friend, not your enemy

The creator of Spring, Rod Johnson, stated in a recent interview that it was not Nullable-types that made him try out Kotlin, but the overly complicated Java Streams API: [Creator of Spring: No desire to write Java].

The following example depicts the various reasons why the Java Streams API is so horribly complicated and how Kotlin solves all of the issues:

Java

```
public record Product(String name, int... ratings){}
List<Product> products = List.of(
  new Product("gadget", 9, 8, 7),
  new Product("goody", 10, 9)
);

Map<String, Integer> maxRatingsPerProduct =
   //🤨 1. Stream introduces indirection
   products.stream()
     //🤨 1. Always to and from Stream conversion
      .collect(
        //🤨 2. Lacks extension methods, so wrappers are required
        Collectors.groupingBy(
           Product::name,
            //🤨 2. Again…
           Collectors.mapping( groupedProducts ->
                //🤨 3. (too) low-level types, arrays, and primitives cause extra
complexity
                //🤨 4. No API on Array, always wrap in stream
              Arrays.stream(groupedProducts.ratings())
                  .max()
                       //🤨 5. Extra verbosity due to Optional
                    .orElse(0),
               //🤨 6. No named arguments: what does this do?
                Collectors.reducing(0, Integer::max)
              )
           ));
```

Kotlin

```
//😃 rich and uniform Collection API - even on Java collections - due to extension
methods
val maxRatingsPerProduct = products.
       .groupBy { it.name }
       .mapValues { (_, groupedProducts) -> //😃 destructuring for semantic
       precision
              //😃 built-in nullability support, and the same API for
              //arrays like other Collections
              groupedProducts.flatMap { it.ratings }
                     .maxOrNull() ?: 0
              }
     }
```

Due to Kotlin's uniform Collection framework, converting between different collection types is very straightforward:

Java

```java
int[] numbers = {1, 3, 3, 5, 2};

Set<Integer> unique =
Arrays.stream(numbers).boxed().collect(Collectors.toSet());

Map<Integer, Boolean> evenOrOdd = unique.stream()
       .collect(Collectors.toMap(
               n -> n,
               n -> n % 2 == 0));
```

Kotlin

```kotlin
val numbers = arrayOf(1, 3, 3, 5, 2)

val unique: Set<Int> = numbers.toSet() //😃 simply call to<Collection>
to do the Conversion

val evenOrOdd: Map<Int, Boolean> = unique.associateWith { it % 2 == 0 }
```

Net effect:

- **Feature-rich, intuitive Collection API:** pipelines read left-to-right like English, not inside nested collector calls.
- **Less boilerplate and complexity:** no `Collectors.groupingBy`, no `Stream`, no `Optional`, no `Arrays.stream`.
- **One mental model:** whether you start with a `List`, `Set`, `Array`, or primitive array, you reach for the *same* collection operators.
- **Performance without pain:** the compiler inserts boxing/unboxing only where unavoidable; you write normal code.
- **Null-safety integrated:** Collection API fully supports nullability with various helpers often suffixed with `orNull(...)`.
- **Seamless Java-interop with idiomatic wrappers:** Due to extension methods, you get the "best of both worlds" – feature-rich Collections on top of Java collections.

Put simply, Kotlin lifts the everyday collection tasks – filtering, mapping, grouping etc. – into first-class, *composable* functions, so you express *what* you want, not the ceremony of *how* to get there.

3. No more checked Exceptions, yet safer code in Kotlin

Java is one of the only languages that still supports checked Exceptions. Though initially implemented as a safety feature, they have not lived up to their promise. Their verbosity, pointless catch blocks that do nothing or rethrow as an Exception as a `RuntimeException`, and their misalignment with lambdas are a few reasons why they get in the way rather than making your code safer.

Kotlin follows the proven paradigm used by almost all other programming languages—C#, Python, Scala, Rust, and Go—of using exceptions only for unrecoverable situations.

The following examples highlight the obstacles that checked Exceptions introduce into your code, without adding any safety:

Java

```
public String downloadAndGetLargestFile(List<String> urls) {
   List<String> contents = urls.stream().map(urlStr -> {
               Optional<URL> optional;
               try {
                   optional = Optional.of(new URI(urlStr).toURL());
                   //🤨 Within lambdas checked exceptions are not supported and must
always be caught...
               } catch (URISyntaxException | MalformedURLException e) {
                   optional = Optional.empty();
               }
               return optional;
           }).filter(Optional::isPresent)       //Quite a mouthful to get rid of the
Optional...
           .map(Optional::get)
           .map(url -> {
               try (InputStream is = url.openStream()) {
                   return new String(is.readAllBytes(), StandardCharsets.UTF_8);
               } catch (IOException e) {
                   //🤨... or re-thrown, which is annoying, I don't really care about IOE
                   throw new IllegalArgumentException(e);
               }
           }).toList();
   //🤨 An empty List results in a NoSuchElementException, so why is it not checked? The
chance that the List is empty is as high as the other two cases above...
   return Collections.max(contents);
}
```

Kotlin

```
//😃 safe return type
fun downloadAndGetLargestFile(urls: List<String>): String? =
   urls.mapNotNull {    //😃 convenient utility methods to rid of null
       //😃 try catch is possible, yet runCatching is an elegant way to convert an
exception to null
       runCatching { URI(it).toURL() }.getOrNull()
   }.maxOfOrNull{  //😃 safe way to retrieve the max value
       it.openStream().use{ it.reader().readText() }  //😃 convenient extension methods to
make java.io streams fluent
}
```

4. Functions as first-class citizens

Kotlin treats functions as first-class citizens, which may raise questions if you're coming from Java: What does that mean, and why does it matter?

The key difference lies in Java's limited approach – its functional capabilities focus mostly on the call-site via lambdas, while the declaration-side remains tied to verbose and less intuitive functional interfaces. In Kotlin, you can define, pass, return, and compose functions without any boilerplate, making functional programming far more expressive and natural.

Java

```java
public void doWithImage(
      URL url,
      //🤨 Function interfaces introduce an indirection: because we don't see the signature of a Function we don't know what a  BiConsumer does, unless we look it up
      BiConsumer<String, BufferedImage> f) throws IOException {
        f.accept(url.getFile(), ImageIO.read(url));
}

//🤨 Same here
public void debug(Supplier<String> f) {
   if(isDebugEnabled()) {
      logger.debug("Debug: " + f.get());
   }
}

//🤨 calling no-argument lambdas is verbose
debug(() -> "expensive concat".repeat(1000));
```

Kotlin

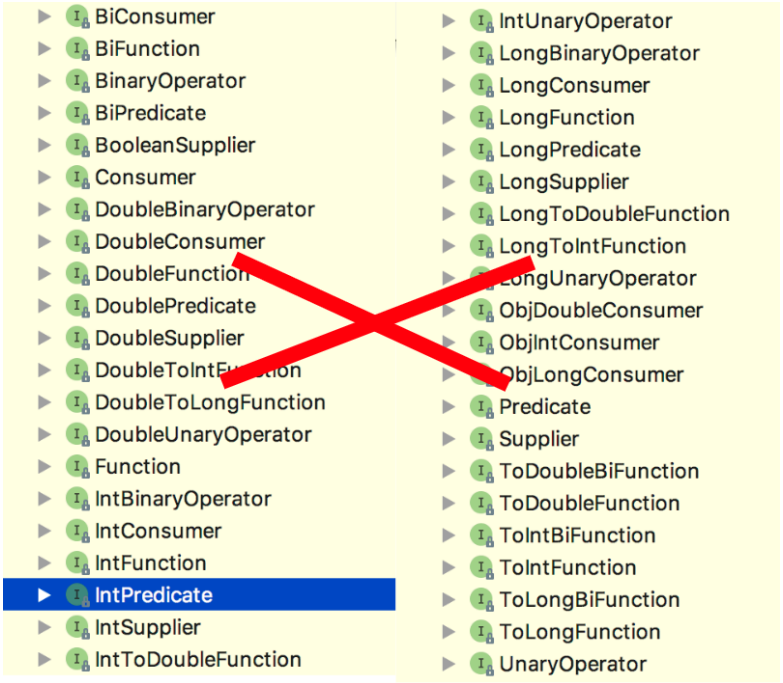
```kotlin
fun doWithImage(
   url: URL,
  //😃 Kotlin has a syntax for declaring functions: from the signature, we see what goes in and what goes out
   f:(String, BufferedImage) -> Unit) =
      f(url.file, ImageIO.read(url))

  //😃 same here: nothing goes in, a String goes out
fun debug(msg: () -> String) {
   if(isDebugEnabled) {
            logger.debug(msg())
   }
}

//😃 convenient syntax to pass a lambda: {<lambda body>}
debug{"expensive concat".repeat(1000)}
```

Kotlin provides a clear and concise syntax for declaring functions – just from the signature, you can immediately see what goes in and what comes out, without having to navigate to an external functional interface.

Java “leaks” its functional programming features through a large set of `java.util.function.*` interfaces with verbose and complex signatures, which often makes functional programming cumbersome to use. Kotlin, by contrast, treats functions as first-class citizens: these interfaces remain hidden from the developer, yet remain fully interoperable with Java’s approach:



As a result, using functions in Kotlin is significantly more straightforward and intuitive, which lowers the threshold considerably for leveraging this powerful programming concept in your own code.

5. Headache-free concurrency with coroutines

If you need high throughput, parallel processing within a single request, or streaming, the only option you have in Java is a reactive library, like Reactor and RxJava, available in frameworks like Spring WebFlux, Vert.X, Quarkus etc.

The problem with these libraries is that they are notoriously complicated, forcing you into functional programming. As such, they have a steep learning curve, yet it is very easy to make errors that can have dire consequences when the application is under load. That's most probably why reactive programming never became mainstream.

> **Important**: VirtualThreads are not a replacement for reactive libraries, even though there is some overlap. VirtualThreads offer non-blocking I/O, but do not provide features such as parallel processing or reactive streams. Structured concurrency and Scoped Values will also enable parallel processing once the major frameworks have support for it. For reactive streams, you will always have to rely on a reactive library.

So let's assume you are a Java developer using Spring Boot and you want to make a parallel call within a single request. This is what you end up with:

```
@PostMapping("/users")
@ResponseBody
@Transactional
public Mono<User> storeUser(@RequestBody User user) {
   Mono<URL> avatarMono = avatarService.randomAvatar();
   Mono<Boolean> validEmailMono =
emailService.verifyEmail(user.getEmail());
  //🤨 what does 'zip' do?
   return Mono.zip(avatarMono, validEmailMono).flatMap(tuple ->
       if(!tuple.getT2()) //what is getT2()? It's the validEmail Boolean…
         //🤨 why can I not just throw an exception?
          Mono.error(new InvalidEmailException("Invalid Email"));
       else personDao.save(UserBuilder.from(user)
                                      .withAvatarUrl(tuple.getT1()));
     );
  }
```

Even though, from a runtime perspective, this code performs perfectly well, the accidental complexity introduced is enormous:

- The code is dominated by `Mono`/`Flux`-es, throughout the whole call chain, which enforces you to wrap all domain objects.
- There are many complex operators everywhere, like `zip`, `flatMap`, etc.
- You cannot use standard programming constructs like throwing exceptions
- The business intent of your code suffers significantly: the code focuses on `Mono`s and `flatMap`, thereby obscuring what is truly happening from a business perspective.

The good news is that there is a powerful remedy in the form of Kotlin Coroutines. Coroutines can be framed as a reactive implementation on the language level. As such, they combine the best of both worlds:

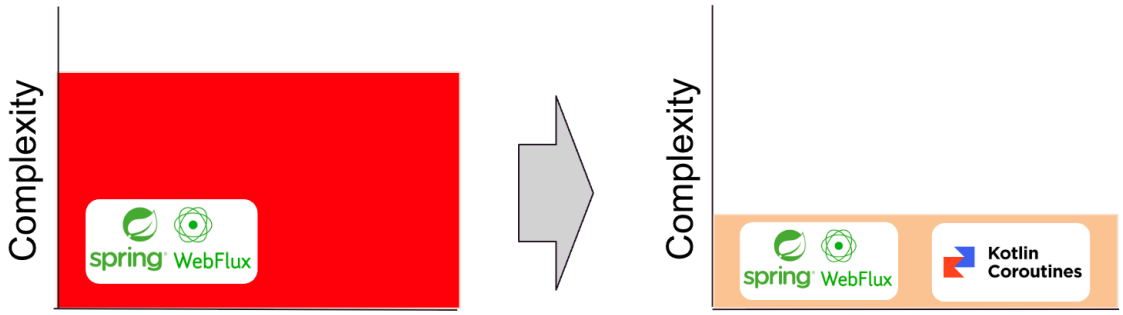
- You write sequential code like you did before.
- The code is executed asynchronously / in parallel at runtime.

The above Java code converted to coroutines looks as follows:

```
@GetMapping("/users")
@ResponseBody
@Transactional
suspend fun storeUser(@RequestBody user:User):User = coroutineScope {
   val avatarUrl = async { avatarService.randomAvatar() }
   val validEmail = async { emailService.verifyEmail() }
   if(!validEmail.await()) throw InvalidEmailException("Invalid email")
   personRepo.save(user.copy(avatar = avatarUrl.await()))
}
```

Kotlin's suspend keyword enables structured, non-blocking execution in a clear and concise manner. Together with `async{}` and `await()`, it facilitates parallel processing without the need for deeply nested callbacks or complex constructs such as `Mono` or `CompletableFuture`.

That's why complexity will decrease, and developer joy and maintainability will increase, with the exact same performance characteristics.



Note: Not all major Java-based web frameworks support Coroutines as well. Spring does an excellent job, as does Micronaut. Quarkus currently offers limited Coroutine support.

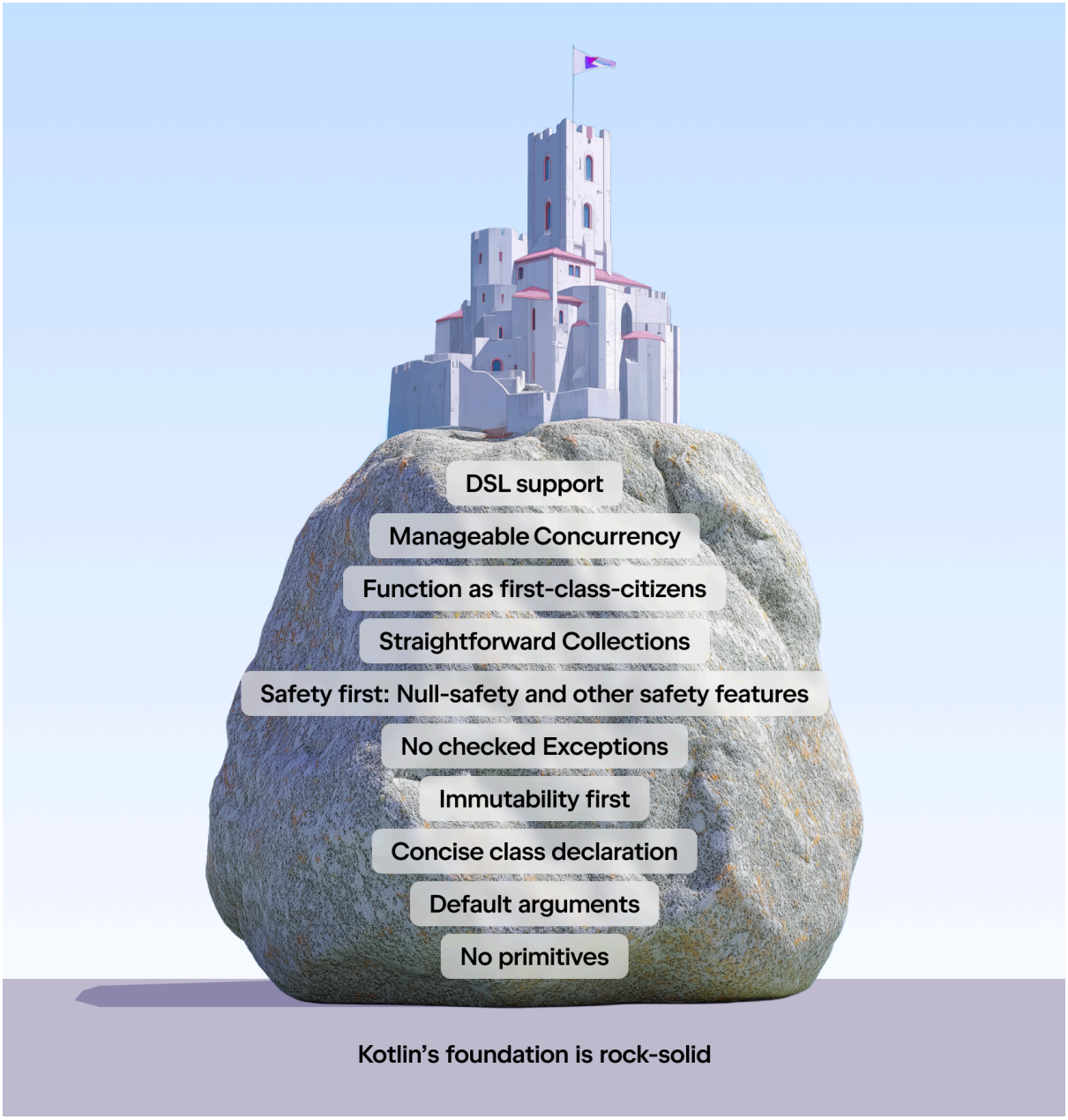
6. But hey, Java is evolving too!

Java keeps moving forward with features like records, pattern matching, and upcoming projects such as Amber, Valhalla, and Loom. This steady evolution strengthens the JVM and benefits the entire ecosystem.

But here's the catch: most of these "new" Java features are things Kotlin developers have enjoyed for years. Null safety, value classes, top-level functions, default arguments, concise collections, and first-class functions are all baked into Kotlin's design—delivered in a more unified and developer-friendly way. That's why Kotlin code often feels cleaner, safer, and far more productive.

And there is more: Kotlin also profits from Java's innovation. JVM-level advances like Virtual Threads and Loom in general, or Valhalla's performance boosts apply seamlessly to Kotlin too.

In short: Java evolves, but Kotlin was designed from day one to give developers the modern tools they need—making it a safe, modern, and forward-looking choice for building the future.



7. Kotlin's evolutionary advantage

Older programming languages inevitably carry legacy baggage compared to modern alternatives. Updating a language while supporting massive existing codebases presents unique challenges for language designers. Kotlin enjoys two crucial advantages:

Standing on the shoulders of giants: Rather than reinventing the wheel, Kotlin's initial design team gathered proven paradigms from major programming languages and unified them into a cohesive whole. This approach maximized evolutionary learning from the broader programming community.

Learning from Java's shortcomings: Kotlin's designers could observe Java's pitfalls and develop solid solutions from the ground up.

For deeper insights into Kotlin's evolution, Andrey Breslav from Kotlin's original design team gave an excellent talk at KotlinDevDay Amsterdam: *Shoulders of Giants: Languages Kotlin Learned From* – Andrey Breslav

Soft factors: Support and connect developers in their Kotlin adoption journey

1. Facilitate easy onboarding

The goal of the expressive Java vs. Kotlin code snippets is to whet the appetite for Kotlin. Code alone, however, is not enough to win the hearts and minds of Java developers. To accelerate adoption and ensure a smooth start, provide:

- **Sample project**: A ready-to-run project with both Java and Kotlin code side-by-side, giving teams a practical reference during migration.
- **Built-in quality checks**: Preconfigured with tools like SonarQube, ktlint, and detekt to promote clean, consistent, and maintainable code from day one. This will enable you to apply consistent lint rules, testing frameworks, libraries, and CI pipelines to reduce friction across teams.
- **Coaching and support**: Experienced Kotlin engineers available to coach new teams, answer questions, and provide hands-on advice during the early stages of development.
    - This one is particularly important: with only a few hours of guidance from an experienced developer from another team who has gone through these stages before, much damage and technical debt can be prevented.

      A bit of support and coaching is the most powerful way to nurture lasting enthusiasm for Kotlin.

2. Offer (self-)study material

Especially when coming from Java, learning the Kotlin basics can be done on your own. Providing some resources upfront eases the path to getting productive with Kotlin and lowers the threshold.

- **JetBrains "Tour of Kotlin" (Get Started guide)**
  A short, browser-based tutorial on the official Kotlin site.
- **Kotlin Koans**
  A set of bite-sized coding challenges maintained by JetBrains.
- **Udacity "Kotlin Bootcamp for Programmers"**
  A full video-based course created in partnership with Google.

Note: Although self-study is valuable for grasping the basics, it also has some drawbacks. One of these drawbacks is an optionality: being caught up in the day's frenzy, it's tempting to skip self-study. Moreover, you will lack feedback from a practitioner who knows the subtle nuances of correctly applied idiomatic Kotlin. There is a big chance that after a self-study course, you will write Java-ish Kotlin, which gives you some benefits but does not unlock the full potential of the language.

Unless you don't have good coaching, a classical course can be very beneficial. There is a need to attend; you can exchange with peers of the same level and have your questions answered by an experienced professional, which will get you up to speed much faster with less non-idiomatic Kotlin during the initial transition.

3. Establish an in-house Kotlin community

One of the fastest ways to level up Kotlin expertise across your company is to create and foremost nurture an in-house community.

- **Launch an internal Kotlin community**
    - First look for a core-team of at least 3–6 developers who are willing to invest in a Kotlin community. Also ensure they get time and credits from their managers for their task.
    - Once the team is formed, organize a company-wide kick-off with well-known speakers from the Kotlin community. This will ignite the Kotlin flame and give it momentum.
    - Schedule regular meet-ups (monthly or bi-weekly) so momentum never drops.
    - Create a shared chat channel/wiki where questions, code snippets, and event notes live.

- **Invite (external) speakers**
    - Bring in engineers who've shipped Kotlin in production to share candid war stories.
    - Alternate between deep-dive technical talks (coroutines, KMP, functional programming) and higher-level case studies (migration strategies, tooling tips).

- **Present lessons learned from other in-house projects**
    - Ask project leads to present their Kotlin learnings – what worked, what didn't, and measurable outcomes.
    - These insights can be captured in a "Kotlin playbook" that new teams can pick up.

- **Give your own developers the stage**
    - Run lightning-talk sessions where anyone can demo a neat trick, library, or failure they overcame in 5–10 minutes.
    - Celebrate contributors publicly – shout-outs in all-hands or internal newsletters boost engagement and knowledge-sharing.

- **Keep feedback loops tight**
    - After each session, collect quick polls on clarity and usefulness, then tweak future agendas accordingly.
    - Rotate organizational duties so the community doesn't hinge on a single champion and stays resilient long-term.

Note: Many of the suggestions above sound straightforward. However, the effort needed to keep a community vibrant and alive should not be underestimated.

4. Be patient…

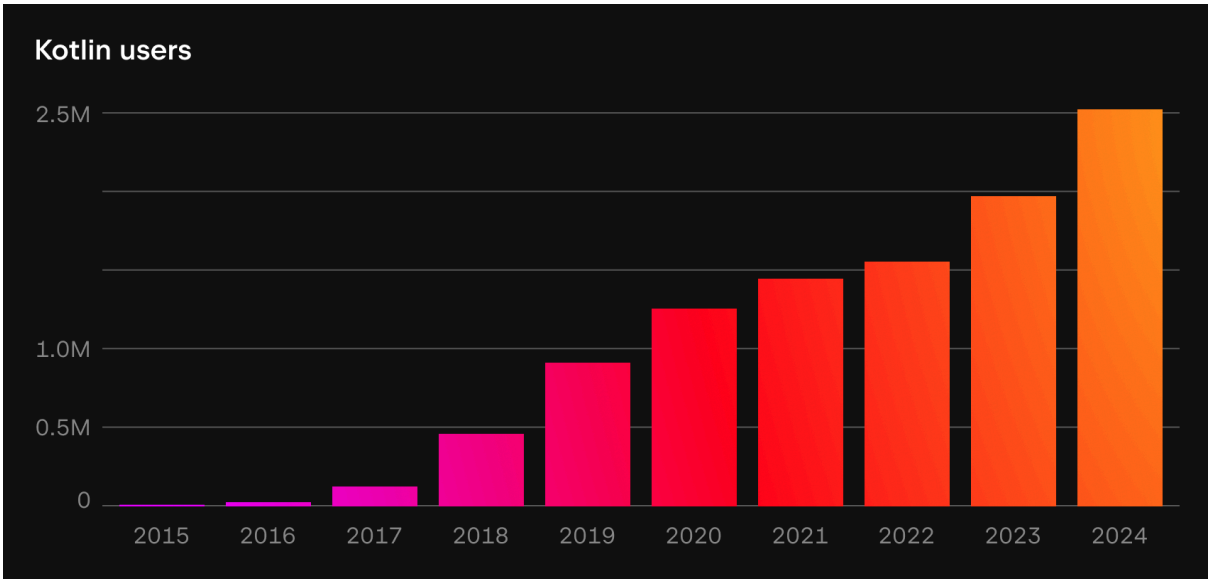
Cultural change takes time. The danger of being enthusiastic about a tool that makes a difference for you is pushing things, which can be counterproductive. An effective approach is to nurture the adoption process with all the activities discussed above, with *show, don't tell* as prior guidance.

5. Persuading Management: Building the Business Case for Kotlin

Once you have achieved a critical mass of Kotlin supporters, presenting a compelling business case to management requires moving beyond code demonstrations to hard data and strategic considerations.

Market momentum and industry validation

Kotlin has demonstrated remarkable growth, with 2.5 million developers worldwide now using the language. This represents sustained adoption since Google's pivotal 2017 announcement declaring Kotlin the official language for Android development, followed by Spring Boot's embrace of Kotlin as a first-class citizen.



Source: [Keynote KotlinConf 2025]

The momentum has accelerated significantly. Spring Boot, which commands approximately 90% of the backend market share, formalized a strategic partnership with JetBrains in 2025 – a clear signal of Kotlin's enterprise viability and long-term trajectory.

Framing the strategic decision

While these adoption metrics demonstrate Kotlin's market validation, management decisions require a comprehensive cost-benefit analysis. Technology adoption at scale involves significant investments in training, tooling, and potential migration efforts. The key is presenting both the tangible benefits and realistic implementation costs, allowing leadership to make an informed strategic decision rather than a purely technical one.

The business gains unlocked by embracing Kotlin

These are the business gains when choosing Kotlin:

Advantage	Details	Business/Team impact
~30% code reduction / more code clarity	~30% less to write, read, fix, and maintain.	➜ Increase in productivity. ➜ Improved maintainability.
25% fewer bugs due to safety features	Kotlin reduces defects out of the box – no extra skills required – through features like null safety, safe APIs, and sensible defaults.	➜ Better quality – happier customers and developers ➜ More time for new features ➜ Faster time-to-market
Invest in just one new language: Kotlin	Keep existing frameworks (e.g. SpringBoot, Micronaut, Quarkus).	➜ Framework knowledge is preserved.
Better choice for AI	Kotlin is more AI-friendly due to noise reduction, clarity, fluency, and more, which aligns with how LLMs process code.	➜ Better productivity with AI, which is increasingly influencing coding.
Fully interoperable with Java	Kotlin is fully interoperable with Java; no rewrite of libraries required.	➜ Investments based on Java are preserved.
Well supported	Backed by strong players: JetBrains, Google, and Meta.	➜ Future-proof choice.
Gradual adoption	Kotlin supports incremental migration paths.	➜ No “Big Bang” needed.
Increase in developer happiness	In surveys and studies, Kotlin developers consistently report better developer experience and higher job satisfaction.	➜ Attract new developers. ➜ Better retention 😃😃😃.
Faster audit, safer compliance	Kotlin’s type safety, immutability, and clear data structures make systems easier to trace, validate, and certify. These qualities support transparent and reliable software processes across regulated domains. For a real-world example, see Kotlin in Payment Gateways and Fintech: A Strategic Fit for 2026 Architectures.	➜ Easier internal and external audits. ➜ Lower compliance effort and risk. ➜ Greater organizational transparency..

Some words on developer happiness

Looking at my career, I was greatly rewarded for letting go of my attachment to the seemingly *safe* career choice of Java. Daily, I'm empowered with a language that is a joy to work with and helps me express what I want to achieve in the best possible way, concisely, safely, and productively.

It's a fact that, personally, I've become a happier developer (human ;-)?) using Kotlin. Though this is subjective, developer satisfaction data tells the same unmistakable story:

Backend developers who migrate from Java to Kotlin report dramatically higher happiness levels across every major industry survey.

- Stack Overflow's Developer Survey data shows Kotlin consistently ranking among the top five most loved languages at ~63% satisfaction, significantly outpacing Java's ~54% rating.
- The Mercedes-Benz.io backend team documented "significantly more concise code than Java, reducing boilerplate and improving readability" while achieving measurable productivity gains in their microservices architecture.
- Tyler Russell, a backend developer with 15 years of Java experience, reports after four years with Kotlin: "I really like Kotlin... it feels like it enables productivity significantly more than it hinders it", and never wants to return to Java.

Corporate Kotlin to Java backend migrations demonstrate that these happiness improvements scale effectively across enterprise environments.

- N26's backend engineering team converted 60% of their microservices to Kotlin within two years, reporting enhanced developer satisfaction and reduced production issues.
- ING's five-year backend adoption journey shows sustained organic growth, with 8% of their 20,000+ technical staff repositories now using Kotlin, demonstrating that developers voluntarily choose Kotlin when given the option.

The cost / return on investment of embracing Kotlin:

These are the investments required for choosing Kotlin.

Investment	Description	Expected Return / Impact
Developer training	Training sessions, workshops, and onboarding materials.	➜ Experienced developers new to Kotlin are productive in 2–3 weeks. ➜ ROI in ~1–4 months.
Codebase migration (if applicable)	A gradual rewrite or a mixed Kotlin/Java code setup strategy is required.	➜ No need for big bang rewrite ➜ Maintains delivery speed.
Knowledge sharing	Internal Kotlin champions, brown bag sessions, shared repos with examples.	➜ Faster team-wide adoption. ➜ Reuse of best practices.
Hiring / upskilling	Hire Kotlin-experienced devs or upskill existing Java devs.	➜ Higher retention. ➜ Broader talent pool.

One of the most common concerns when considering Kotlin adoption is the perceived shortage of experienced Kotlin developers. But rather than a roadblock, this can be a **strategic advantage**:

- ✅**Fast upskilling for Java developers**

  Kotlin is designed to be intuitive for Java developers. With the right guidance, most can become productive in just **2–3 weeks**.

- ✅**Kotlin attracts high-quality developers**

  Those who actively pursue Kotlin are often **curious, modern, and quality-driven** – traits you want in your engineering team.

- ✅**Kotlin enhances your company image**

  Adopting Kotlin signals that you embrace **modern, forward-looking technology**, which helps attract top-tier talent.

- ✅**Kotlin boosts developer happiness and retention**

  Kotlin developers consistently report **higher job satisfaction**, making it easier to retain engaged and motivated team members.

  In short: Kotlin isn't just a smart tech choice – it's a **strategic people decision**.

6. Success Factors for Large-Scale Kotlin Adoption

Gaining developer buy-in and management support for Kotlin is a significant milestone, but it's not the finish line. The real challenge begins when you're faced with existing Java codebases that need to coexist with or transition to Kotlin. How do you navigate this hybrid world effectively?

The key to managing legacy codebases is developing a strategy that aligns with your organizational goals and operational realities. Here's a proven approach that has worked out well in practice.

Application lifecycle strategy

Different applications require different approaches based on their lifecycle phase. Let's examine three distinct categories:

End-of-life applications

Strategy: Leave them alone.

If an application is scheduled for retirement, there's no business case for migration. Keep these systems in Java and focus your energy where it matters most. The cost of migration will never be justified by the remaining lifespan of the application.

New systems

Strategy: Default to Kotlin.

In organizations where Kotlin adoption is complete, greenfield projects naturally start with Kotlin. If the adoption process is in progress, teams often get the choice between Java and Kotlin. Choose wisely ;-).

Active applications

Strategy: Pragmatic, feature-driven migration.

Active applications are the ones that require careful consideration: Rewriting for the sake of rewriting is a tough sell to your Product Owner. Instead, combine migration efforts with new feature development. This approach provides tangible business value while modernizing your codebase. The different attack angles we already discussed in the section: *Extend/Convert an existing Java application*.

Java-to-Kotlin conversion approaches

When converting Java to Kotlin, you have several options, each with distinct trade-offs:

1. Complete rewrite

Best for: Small codebases

Challenge: Time-intensive for larger systems

Rewriting a codebase from scratch gives you the cleanest, most idiomatic Kotlin code. This approach is suitable for small codebases, like a MicroService. For large codebases, this approach generally tends to be prohibitively expensive.

2. IDE auto-conversion with manual refinement

Best for: Medium codebases with dedicated refactoring time

Challenge: Manual refinements are mandatory

IntelliJ IDEA's *Convert Java to Kotlin* feature provides a literal translation that's far from idiomatic. Consider this example:

Take 1:

Java

```
record Developer(
       String name,
       List<String> languages,
       String email,
) {}
```

Raw auto-conversion result:

Kotlin

```
@JvmRecord
data class Developer(
   val name: String?,
   val languages: MutableList<String?>?
)
```

This conversion has several issues:

- Everything becomes nullable (overly defensive).
- Java Collections become Kotlin's `MutableList` instead of Kotlin's default read-only `List`.

Improving the conversion with jspecify annotations:

Luckily, for the conversion of all Java types to Nullable types in Kotlin, there is a remedy in the form of @NonNull/@Nullable annotations. Different options are available; the most modern one is jspecify, which has recently also been officially supported by Spring:

```
<dependency>
   <groupId>org.jspecify</groupId>
   <artifactId>jspecify</artifactId>
   <version>1.0.0</version>
</dependency>
```

```
implementation("org.jspecify:jspecify:1.0.0")
```

With jspecify, we can annotate the Java code with @Nullable and @NonNull.

Take 2:

Java

```
import org.jspecify.annotations.NonNull;
import org.jspecify.annotations.Nullable;

record Developer(
       @NonNull String name,
       @NonNull List<@NonNull String> languages,
       @Nullable String email
) {}
```

Now the auto-conversion produces much better results:

Kotlin

```
@JvmRecord
data class Developer(
   //😃 non-null as requested
   val name: String,
   //😃 both, collection and type are non-null
   val languages: MutableList<String>,
   //😃 nullable as requested
   val email: String?,
)
```

Limitations of the auto conversion approach:

Even with jspecify annotations, complex Java patterns don't translate well. Consider this auto-converted code example shown in section: *3. No more checked Exceptions, yet safer code in Kotlin*:

Kotlin

```
fun downloadAndGetLargestFile(urls: MutableList<String>): String? {
     //🤨 using Stream, instead of Kotlin Collections
     val contents = urls.stream().map { urlStr: String? ->
     //🤨 still using Optional, rather than Nullable types
     //🤨 var but we want val!
     var optional: Optional<URL>
     //🤨 useless try-catch, no need to catch in Kotlin
     try {
         optional = Optional.of(URI(urlStr).toURL())
     } catch (e: URISyntaxException) {
           optional = Optional.empty<URL>()
     } catch (e: MalformedURLException) {
           optional = Optional.empty<URL>()
     }
     optional
   }.filter { it!!.isPresent() }//🤨 discouraged !! to force conversion to
non-null
    .map {it.get() }
    .map { url: URL ->
      //🤨 useless try-catch, no need to catch in Kotlin
      try {
        url.openStream().use { `is` ->
           String(`is`.readAllBytes(), StandardCharsets.UTF_8)
        }
      } catch (e: IOException) {
         throw IllegalArgumentException(e)
       }
    }.toList()
   //🤨 usage of Java collections...
   return Collections.max(contents)
}
```

The autoconversion is far away from the desired, idiomatic result:

Kotlin

```
fun downloadAndGetLargestFile(urls: List<String>): String? =
   urls.mapNotNull {
       runCatching { URI(it).toURL() }.getOrNull()
   }.maxOfOrNull{ it.openStream().use{ it.reader().readText() } }
```

The auto-conversion provides a starting point, but significant manual refinement and knowledge of idiomatic Kotlin is required to achieve truly idiomatic Kotlin.

3. AI-assisted conversion

Best for: Larger codebases with robust testing infrastructure

Challenge: Manual review required

AI can potentially produce more idiomatic results than basic auto-conversion, but success requires careful preparation:

Prerequisites:

1. **Comprehensive testing coverage:** Since LLMs are unpredictable, you need reliable tests to catch AI hallucinations.
2. **Well-crafted system prompts:** Create detailed instructions for idiomatic Kotlin conversions that are aligned with your standards. You can use this system prompt as a starting point.
3. **Extensive code review:** AI output requires a thorough review for logical and idiomatic correctness, which can be mentally taxing for large codebases.

Using the proposed system prompt to guide the conversion, the result is quite satisfying, yet not perfect:

Kotlin

```
fun downloadAndGetLargestFile(urls: List<String>): String? {
   val contents = urls
      .mapNotNull {
            urlStr -> runCatching { URI(urlStr).toURL() }.getOrNull()
      }.mapNotNull { url -> runCatching {
            url.openStream().use { it.readAllBytes().toString(UTF_8)
         }
      }.getOrNull() }

   return contents.maxOrNull()
}
```

4. Auto-conversion at scale

Best for: Massive codebases requiring systematic transformation

Currently, there's no official tooling for converting Java codebases to idiomatic Kotlin at scale. However, both Meta and Uber have successfully tackled this challenge for their Android codebases using approaches that work equally well for backend applications. The following documentation and talks provide insights into how Meta and Uber approach this quest:

Meta's approach:

- **Strategy:** Rule-based, deterministic transformation
- **Resources:**
  - Engineering blog post
  - Conference talk

Uber's approach:

- **Strategy:** Rule-based, deterministic transformation using AI to generate conversion rules
- **Resource:** KotlinConf presentation

Both companies succeeded by creating systematic, repeatable processes rather than relying on manual conversion or simple automation. Their rule-based approaches ensure consistency and quality across millions of lines of code.

> **Important**: Converting Java to Kotlin at scale introduces a challenge on the social level: to be reliable, you still want 'the human in the loop' reviewing the generated code. However, if not planned carefully, Engineers can easily be overwhelmed by the flood of pull requests resulting from the automated conversion. Therefore, the social impact needs to be considered carefully.

Remember, successful large-scale Kotlin adoption isn't just about converting code – it's about building team expertise, establishing coding standards, and creating sustainable processes that deliver long-term value to your organization.

Short recap on when to use which approach:

- Small application or being rewritten anyway?
  → **Rewrite in Kotlin (1)**

- Medium codebases with dedicated refactoring time or team learning Kotlin?
  → **Intellij IDEA Auto-Conversion + refine** (start with tests first) **(2)**

- Mid/large code with repeated patterns and good tests?
  → **AI-assisted approach (3)**

- Org-level migration to Kotlin across many services?
  → **Auto-Conversion at scale with concrete plan** (platform-led) **(4)**

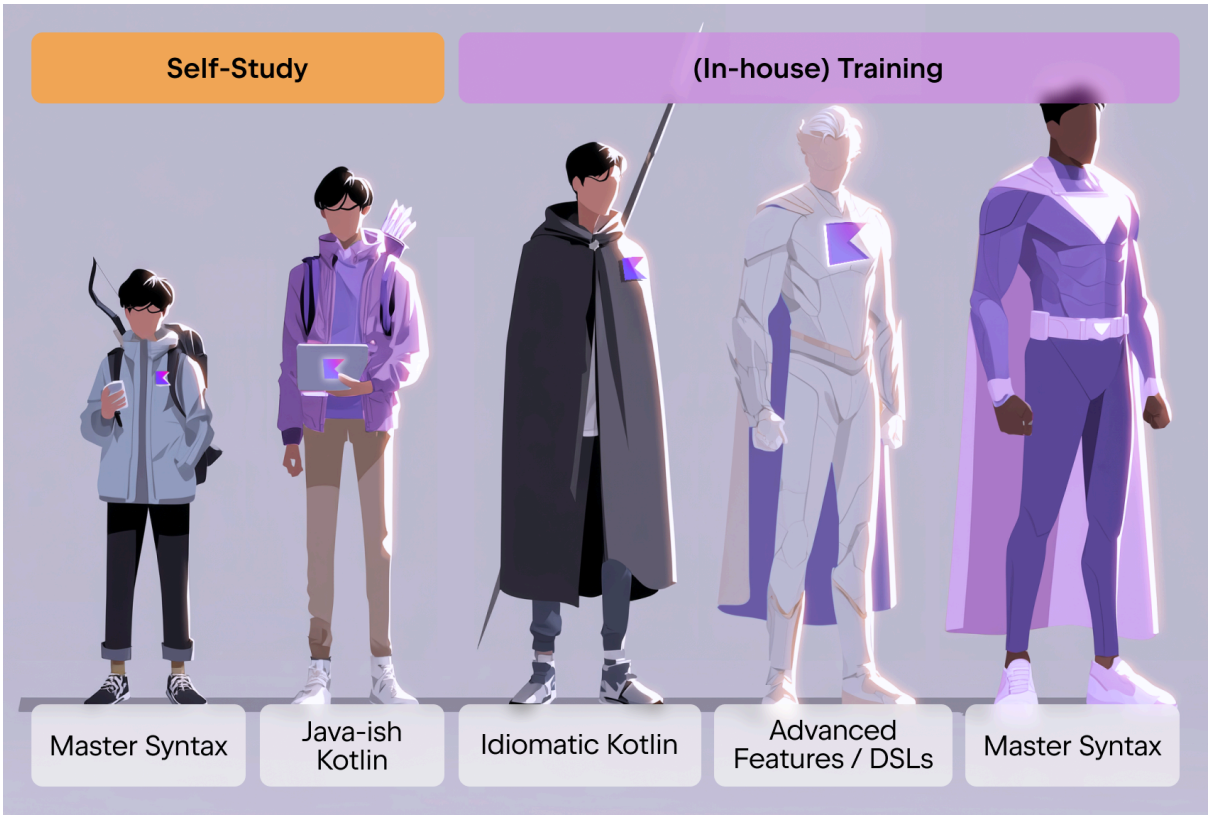
Unlock Kotlin's full potential

Kotlin makes developers productive fast. Its concise syntax, safety features, and rich standard library help many developers write better code within weeks – often through self-study or on-the-job learning. But without guidance, many fall into the trap of using Kotlin in a *Java-like way*: sticking to mutable structures, verbose patterns, and missing out on idiomatic features.

Reaching the next level

Reaching the *next level* – embracing immutability, expression-oriented code, DSLs, and structured concurrency with Coroutines (with or without Virtual Threads) – is where many developers get stuck.

At this stage, I've found that external training makes a far greater difference than self-study. Even developers with years of Kotlin experience often benefit from focused coaching to adopt idiomatic patterns and unlock the language's full potential.



7. To Kotlin or Not to Kotlin: What Kind of Company Do You Want to Be?

Ultimately, the decision to adopt Kotlin reflects your engineering culture. Do you value the:

- **Traditional approach (Java)**: Conservative, ceremonial, stable.
- **Progressive approach (Kotlin)**: Pragmatic, modern, adaptive.

Both have their merits. Java will get the job done. Kotlin will likely get it done better, with happier developers and fewer bugs. The question isn't whether Kotlin is better – it's whether your organization is ready to invest in being better.

The journey from that first Kotlin test to organization-wide adoption isn't always smooth, but with the right approach, it's remarkably predictable. Start small, prove value, build community, and scale thoughtfully.

Your developers – current and future – will thank you.