HISTORY OF KOTLIN

- JetBrains builds developer tools.

- Kotlin was announced by JetBrains in 2011 during its early development.

It targeted Java 6.

- Android devs were stuck on Java 7 by that time. Java 8 was released in 2014.

- Android official support for Kotlin in 2017.

- Android went "Kotlin first" in 2019.

- Support in Spring 5+ and Spring Boot 2+

Kotlin can provide modern features on older JVM targets.

Share code across Android, iOS, backend, web, desktop.

```kotlin
/*
  Kotlin does not need ";" to end statements
  No wrapping class
  No arguments
  No static modifier
 */
fun main() {
    println("Welcome to the Madrid JUG")
}

/*
  Kotlin has no primitive data types. EVERYTHING is an object.
 */
fun sumNumbers(firstNumber: Int, secondNumber: Int): Int {
    return firstNumber + secondNumber
}

/*
 Assignments are NOT expressions in Kotlin
*/
var n = m = getNumber()      // Compile error
```

STRING INTERPOLATION

```kotlin
/*
 String interpolation
 An expression is evaluated inside the argument to println.
*/
fun greetUser() {
    val name = readln()
    println("Hello, ${if (name.isBlank()) "someone" else name}!")
}
```

```kotlin
fun greetUser() {
    val name = readln()
    println("Hello, ${name.ifBlank { "someone" }}!")
}
```

FUNCTIONS

```kotlin
/*
 A function that returns no meaningful value.
 "Unit" return type can be omitted. It is equivalent to "void" in Java.
*/
fun printSum(a: Int, b: Int): Unit {  // Warning: Redundant 'Unit' return type
    println("sum of $a and $b is ${a + b}")
}

fun welcomeUser(message: String): String {   // the function parameter can't be modified
    return "Hello, $message"
}

/* A function as an expression */
fun welcomeUserAsAnExpression(message: String) = "Hello, $message"

fun main() {
    println(welcomeUser("Welcome to the Madrid JUG"))
    printSum( a: 2,  b: 5)
}
```

COMPARISON AND EQUALITY

```
class Customer

fun comparisonAndEquality() {
    val num1 = 1000
    val num2 = 1000
    val str1 = "Hi"
    val str2 = "Hi"
    val customer1 = Customer()
    val customer2 = Customer() // Java: var customer2 = new Customer();

    println(num1 == num2) // true
    println(num1.equals(num2)) // true
    // we get a Sonarlint warning, because "equals" does not allow the operands to be null

    // If you do want to check for reference equality, use ===
    // This won't work for some of the basic types:
    println(num1 === num2)    // true
    println("================ Strings ================")
    println(str1 === str2)    // true
    println("================ Classes ================")
    println(customer1 === customer2)    // false
    println(customer1 == customer2)     // false
}
```

CONTROL FLOW

```kotlin
/*
 "If" can be a statement, like in Java ---> return (a > b) ? a : b;
 There is no ternary operator in Kotlin
*/
fun maxOfStatement(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}

/* if is an expression here */
fun maxOf(a: Int, b: Int): Int {
    return if (a > b) {
        a
    } else {
        b
    }
}

/* if expression in only one line */
fun maxOfOneLine(a: Int, b: Int) = if (a > b) a else b
```

CONTROL FLOW

In Kotlin, most control structures, except for the loops (for, while, and do / while) are expressions, which sets it apart from other languages like Java.

- while ----------> like in Java

- do…..while ---------> like in Java

- break ----------> like in Java

LOOPS and RANGES

```kotlin
val kotlinLovers = listOf("Juanjo", "Jose", "David")
for (j in kotlinLovers) {
    println(j)
}

/*
 Descending, inclusive range
*/
for (i in 6 downTo 0) {
    print("$i ")   // 6 5 4 3 2 1 0
}

/*
 Only even numbers
*/
for (i in 6 downTo 0 step 2) {
    print("$i ")   // 6 4 2 0
}

/*
 Exclusive range
*/
for (i in 0 until 6) {
    print("$i ")    // 0 1 2 3 4 5
}
```

"Kotlin para pia

https://githu

WHEN EXPRESSION

- Similar to “switch” in modern Java.

```
fun describe(obj: Any): String =
    when (obj) {
        in 1..3    -> "One, two or three"
        "Hello"    -> "Greeting"
        is Long    -> "Long"
        !is String -> "Not a string"
        else       -> "Unknown"   // It will not compile without an else branch
    }
```

Question: what if the parameter “obj” is null?

SMART or AUTOMATIC CASTS

```kotlin
/*
 Smart Casts
 */
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        // `obj` is automatically cast to `String` in this branch
        return obj.length
    }

    // `obj` is still of type `Any` outside the type-checked branch
    return null
}
```

CLASSES

Kotlin classes have properties, not fields

```
fun usingClasses() {
    /*
     There is no "new" keyword in Kotlin.
     Kotlin will generate a getter method for each attribute.
     And for mutable attributes also a setter method.
    */
    val onePerson = Person( name: "José",  age: 24)
    println("New Person: $onePerson")  // New Person: learningkotlin.examples.Person@17f052a3

    onePerson.age = 67

    println("New Person: $onePerson")  // New Person: learningkotlin.examples.Person@17f052a3

    println("New age: ${onePerson.age}")  // New age: 67

    onePerson.dateOfBirth = 2000
    println("${onePerson.dateOfBirth}")   // 2000
    println("${onePerson.nameInUpperCase}")
}
```

GETTER and SETTER METHODS

```kotlin
/*
 Custom setter
 - The property must be inside the class body.
 - Common use case: having a mutable property with some validations
 */
class Person(val name: String, var age: Int) {
    var dateOfBirth: Int = 1970   // default value
        set(value) {
            field = if (value >= 1970) value else 1970
        }
    val nameInUpperCase : String  = name
        get() = field.uppercase()
}
```

DATA CLASSES and JAVA RECORDS

```kotlin
/*
 equals, toString and hashCode functions are generated automatically.
*/
data class Person(val name: String, var age: Int)
```

Similar to Java records:

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

Differences

- Kotlin data classes also generates a **copy()** function. And *componentN()* functions, which are important for variable destructuring.

- In a record, all properties must be final.

- The data classes can inherit from other classes, while records do not allow that.

Accessing a data class from Java

```
@JvmRecord    // will NOT compile.
data class Person(val name: String, var age: Int)
```

Since Java records are immutable, we can't use var declarations for data classes annotated with @JvmRecord.

INHERITANCE

```kotlin
/*
 Kotlin classes are final by default
*/
open class Animal

class Dog: Animal()
```

Interfaces and abstract classes

```kotlin
/*
 Both interfaces and abstract members are always open, so the open modifier is not needed.
*/
interface Clickable {
    fun click()
}

class Button : Clickable {
    // The override modifier is mandatory
    // The overridden member is open by default, if not marked final
    override fun click() = println("Button was clicked")
}
```

SEALED HIERARCHY

```kotlin
/*
 A sealed hierarchy
*/
sealed class Expr
class Num(val value: Int) : Expr()
class Sum(val left: Expr, val right: Expr) : Expr()
class Mul(val left: Expr, val right: Expr): Expr()
fun eval(e: Expr): Int =
    when (e) {
        is Num -> e.value
        is Sum -> eval(e.right) + eval(e.left)
// ERROR: 'when' expression must be exhaustive,
// add necessary 'is Mul' branch or 'else' branch instead.
    }
```

- Similar to modern Java.

```kotlin
sealed class Expr
class Num(val value: Int) : Expr()
class Sum(val left: Expr, val right: Expr) : Expr()
class Mul(val left: Expr, val right: Expr): Expr()
fun eval(e: Expr): Int =
    when (e) {
        is Num -> e.value
        is Sum -> eval(e.right) + eval(e.left)
        is Mul -> TODO( reason: "As of 2024, not yet implemented")
        // TODO("...") is equivalent to NotImplementedError("...")
    }
```

Solution

EXCEPTIONS

- In Java, you have to declare all the checked exceptions that your function can throw, and they need to be handled explicitly.
- In Kotlin, the compiler does not require you to handle exceptions and the "throws" clause does not exist.

```kotlin
fun readNumber(reader: BufferedReader) {
    // "try" can be used as a statement or as an expression
    val aNumber = try {
        Integer.parseInt(reader.readLine())
    } catch (e: NumberFormatException) {
        return
    } finally {
        reader.close()
    }

    println(aNumber)
}
```

This is also valid in Kotlin

```kotlin
fun readNumberNoErrorHandling(reader: BufferedReader): Int {
    val line = reader.readLine()
    reader.close()
    return Integer.parseInt(line)
}
```

How to use in Java functions that do not declare exceptions:

```kotlin
@Throws(NumberFormatException::class)
fun readNumberNoErrorHandling(reader: BufferedReader): Int {
    val line = reader.readLine()
    reader.close()
    return Integer.parseInt(line)
}
```

METHOD OVERLOADING

```java
/*
 Method overloading on Java
 */
void logger(String message, String level) {
    System.out.println(message + " - Level: " + level);
}

void logger(String message) {
    String level = "INFO";
    logger(message, level);
}
```

Kotlin has default parameter values:

```kotlin
@file:JvmName("Utils")

package learningkotlin.examples

import java.io.File

@JvmOverloads
fun logger(message: String = "", level: String = "info"): String {
    return "$message, level $level"
}
```

We can use the Kotlin function in Java:

```java
Utils.logger();
Utils.logger("Inside the main loop");
Utils.logger( message: "Inside the main loop",  level: "Info");
```

COLLECTIONS

```kotlin
fun collections() {

    println("========== Collections ============")
    // similar to Java's List.of() factory method
    val hobbits = listOf("Frodo", "Sam", "Pippin", "Merry")
    hobbits.add("Sam2")  // Compile error!
    println("List: $hobbits")

    // similar to Java's Set.of() factory method
    val uniqueHobbits = setOf("Frodo", "Sam", "Pippin", "Merry", "Frodo")
    println("Set: $uniqueHobbits")

    // similar to Java's Map.of() and Map.ofEntries() factory methods
    val movieBatmans = mapOf(
        "Batman Returns" to "Michael Keaton",
        "Batman Forever" to "Val Kilmer",
        "Batman & Robin" to "George Clooney"
    )
    println(movieBatmans)
}
```

Mutable collections

```
fun mutableCollections() {

    println("========== Mutable Collections ============")
    val editableHobbits = mutableListOf(
        "Frodo", "Sam",
        "Pippin", "Merry"
    )
    editableHobbits.add("Bilbo")
    editableHobbits + "Bilbo"
    println(editableHobbits)
}
```

READING A LIST AND ITS INDEXES

```java
/*
 This is a common task in Java.
 Some of the read data may be null.
 Keep track of the index whose element is null.
*/
List<String> customers = new ArrayList<>();
customers.add("22");
customers.add(null);
for (int i = 0; i < customers.size(); i++) {
    var elem = customers.get(i);
    if (elem == null) {
        elem = "NULL VALUE";
    }
    System.out.println(i + ": " + elem);
}
```

The Kotlin collection functions make this easier:

```kotlin
// "?" indicates it's a nullable list
val customers = mutableListOf<String?>()
customers.add(null)
customers.add("22")
// No need to think about the index bounds
for ((index, elem) in customers.withIndex()) {
    println("$index: $elem")
}
```

If-not-null shorthand or safe-call operator

```kotlin
val files = File("Test").listFiles()
println(files?.size) // size is printed if files is not null
```

If-not-null-else shorthand or Elvis operator

```kotlin
val files = File("Test").listFiles()

if (files != null) {
    println(files.size)
} else {
    println("empty")
}

// Shortcut provided by Kotlin
println(files?.size ?: "empty")
```

These two operators already existed in **Groovy**

```
def name = null
def length = name?.length() ?: 0
println length
```

NULLABLE TYPES

- Int? extends Int.

- **Any?** extends Any. Any is equivalent to Java's **Object** type.

FUNCTIONS RETURNING A VALUE OR NULL

- In Java, you need to be careful when working with list elements.

- You should always check whether an element exists at an index before you attempt to use the element:

```java
var numbers = new ArrayList<Integer>();
numbers.add(1);
numbers.add(2);

System.out.println(numbers.get(0));
int value = numbers.get(5); // Exception!
```

The Kotlin standard library often provides functions whose names indicate whether they can possibly return a null value.

This is especially common in the collections API:

```
val numbers = listOf(1, 2)
// Can throw IndexOutOfBoundsException if the collection is empty
println(numbers[0])
println(numbers.get(0))  // The indexing operator is preferred in Kotlin

println(numbers.firstOrNull())
println(numbers.getOrNull(5)) // null
```

STRINGS IN JAVA AND KOTLIN

Build a string

```java
StringBuilder countDown = new StringBuilder();
for (int k = 5; k > 0; k--) {
    countDown.append(k);
    countDown.append("\n");
}
System.out.println(countDown);
```

In Kotlin:

```kotlin
// Under the hood, the buildString uses Java's StringBuilder
val countDown = buildString {
    for (i in 5 downTo 1) {
        append(i)
        appendLine()
    }
}
println(countDown)
```

STRINGS IN JAVA AND KOTLIN

Take a substring

```java
// Java
String input = "What is the best JVM language? Groovy";
String answer = input.substring(input.indexOf("?") + 1);
System.out.println(answer);
```

```kotlin
val input = "What is the best JVM language? Kotlin hands down"
val answer = input.substringAfter("?")  // or input.substringAfterLast("?")
println(answer)
```

STRINGS IN JAVA AND KOTLIN

Set default value if the string is blank

Java

```java
String nameValue = getName();
String name = nameValue.isBlank() ? "MISSING NAME" : nameValue;
System.out.println(name);
```

Kotlin

```kotlin
val name = getName().ifBlank { "MISSING NAME" }
println(name)
```

STRINGS IN JAVA AND KOTLIN

Split a string

```java
// Java
System.out.println(Arrays.toString("Once.upon.a.time".split("\\.")));
```

In Kotlin

- A List is returned, not an Array.

- The argument is just a String, not a regular expression.

```kotlin
// Kotlin
println("Once.upon.a.time".split("."))   // [Once, upon, a, time]
```

STRINGS IN JAVA AND KOTLIN

Build multiline strings

The difference with Java is that Java automatically trims indents, and in Kotlin you should do it explicitly.

```java
// Java
String result = """
            Kotlin
               Java
            """;
System.out.println(result);
```

```kotlin
// Kotlin
val result = """
    Kotlin
       Java
""".trimIndent()
println(result)
```

STRINGS IN JAVA AND KOTLIN

Create a string from collection items

```java
// Java
List<Integer> allNumbers = List.of(1, 2, 3, 4, 5, 6);
String invertedOddNumbers = allNumbers
        .stream()
        .filter(it -> it % 2 != 0)
        .map(it -> -it)
        .map(Object::toString)
        .collect(Collectors.joining("; "));
System.out.println(invertedOddNumbers);
```

In Kotlin you do not have to use Streams.

```kotlin
val numbers = listOf(1, 2, 3, 4, 5, 6)
val invertedOddNumbers = numbers
    .filter { it % 2 != 0 }
    .joinToString(separator = "; ", prefix = "== ", postfix = " ==") {"${-it}"}
println(invertedOddNumbers)   // == -1; -3; -5 ==
```

EXTENSION FUNCTIONS

```
fun String.lastChar(): Char = get(this.length - 1)
```

- The String class is called a method receiver.
- Even though you may not even have the source code to that class, you can still extend it with the methods you need in your project.
- Unlike methods defined in the class, extension functions do not have access to private or protected members of the class.
- Extension functions are effectively **syntactic sugar over static method calls**.
- Many extension functions are declared in the **Kotlin standard library**.

```
fun Int.esPar() = if (this % 2 == 0) "$this es PAR" else "$this es IMPAR"
```

EXTENSION PROPERTIES

```
val String.lastChar22: Char
    get() = this.get(length - 1)
```

Usage

```
println("Last letter: ${message.lastChar22}")
println("Last letter: ${message.lastChar()}")
```

From Java, we can check an extension function is not a regular method:

```
MyfirstkotlinFileKt.esPar(3);
```

UTILITY CLASS VS. EXTENSION FUNCTIONS

```kotlin
// A Utility class, like in Java
class Utils {
    companion object {
        fun lowercase(str: String) =
            str.lowercase()

        fun lastChar(str: String) =
            str[str.length - 1]
    }
}
```

```kotlin
val str = "ada"
Utils.lastChar(Utils.lowercase(str))

// This is the common way to do it in Java
val strInLowercase = Utils.lowercase(str)
val result = Utils.lastChar(strInLowercase)
```

With extension functions, it is much easier to read

```kotlin
val result = str.lowercase().lastChar()
```

- This makes it a very good language to build DSLs (Domain-Specific Languages).

LAMBDA FUNCTIONS

Lambdas and anonymous functions are known as function literals.

Java’s function types are a bit confusing.

```
Function<String, Boolean> lambdaExpr = s -> s != null;
System.out.println(lambdaExpr.apply("hello"));   // true
```

Kotlin syntax in more clear and coherent:

```
val isPalindrome = { s: String -> s == s.reversed() }
println(isPalindrome("dabalearrozalazorraelabad"))  // true
```

Summary

```kotlin
/*
 A function without arguments that does not return anything:
 () -> Unit
 (String) -> ((Int) -> Boolean) is the same as
 (String) -> (Int) -> Boolean
*/
val lambdaExpr = { s: String -> s != null }

val lambdaExpr2: (String) -> Boolean = { it != null }

val anonymousFunction = fun(s: String): Boolean = s != null
```

```kotlin
/* A function reference */
fun containsLetterB(s: String): Boolean = s.lowercase().contains("b")
val functionRef = ::containsLetterB
```

```kotlin
val doubleLambda: (String) -> (String) -> Unit = { s1 ->
    { s2 -> println("$s1, $s2") }
}
doubleLambda("Hola")("mundo")   // Hola, mundo
```

Important

Lambda argument can be moved out of parentheses

```kotlin
    val evenNumbers = listOf(2, 3, 4)
//                .filter({ it % 2 == 0 })
                .filter { it % 2 == 0 }
```

DELEGATION OR PROXYING PATTERN

In Java

```java
public interface BankCard {
    boolean isValid(LocalDate expirationDate);
}

public final class CreditCard implements BankCard {

    private static final Logger LOGGER =
            LoggerFactory.getLogger(CreditCard.class);

    @Override
    public boolean isValid(LocalDate expirationDate) {
        return true;
    }
}
```

```java
public final class DebitCard implements BankCard {

    private static final Logger LOGGER =
            LoggerFactory.getLogger(DebitCard.class);

    @Override
    public boolean isValid(LocalDate expirationDate) {
        return true;
    }
}
```

```java
public class BankCardController implements BankCard {

    private final BankCard bankCard;

    public BankCardController(BankCard bankCard) {
        this.bankCard = bankCard;
    }

    @Override
    public boolean isValid(LocalDate expirationDate) {
        return bankCard.isValid(expirationDate);
    }
}
```

```java
BankCardController debitCardController =
        new BankCardController(new DebitCard());
BankCardController creditCardController =
        new BankCardController(new CreditCard());

LocalDate localDate = LocalDate.from(Instant.now());
debitCardController.isValid(localDate);
creditCardController.isValid(localDate);
```

What about adding another kind of BankCard ?

DELEGATION IN KOTLIN

```kotlin
interface BankCard {
    fun isValid(expirationDate: LocalDate): Boolean
}

class DebitCard(val num: Int): BankCard {
    override fun isValid(expirationDate: LocalDate): Boolean {
        return true
    }
}

class CreditCard(val num: Int): BankCard {
    override fun isValid(expirationDate: LocalDate): Boolean {
        return true
    }
}
```

```
class PointsCard(val num: Int): BankCard by DebitCard(num)
```

- The class PointsCard also implements the interface, but not the method.
- Instead, it delegates the method call to an existing implementation.
- The delegate object is defined after the ***by*** keyword.
- No boilerplate code is required.

STANDARD DELEGATES

- The Kotlin standard library contains a number of useful delegates, like `lazy`, `observable`, and others.
- Lazy is used for lazy initialization.

L A Z Y EVALUATION

```
val lazyValue by lazy {
    println("Evaluating.....only now the field is initialized")
    18
}
```

- The first call to get() executes the lambda expression passed to lazy() as an argument and saves the result.
- Further calls to get() return the saved result.
- Lazy<T> instances can only be applied to read-only properties (val).
- Initialization by lazy { … } is thread-safe by default.

Java

```java
public class LambdaSupplier<T> {

    protected final Supplier<T> expensiveData;

    public LambdaSupplier(Supplier<T> expensiveData) {
        this.expensiveData = expensiveData;
    }

    public T getData() {
        return expensiveData.get();
    }
}
```

Making it thread-safe is not easy.

```java
public class LazyLambdaThreadSafeSupplier<T> extends LambdaSupplier<T> {

    private final AtomicReference<T> data;

    public LazyLambdaThreadSafeSupplier(Supplier<T> expensiveData) {
        super(expensiveData);
        data = new AtomicReference<>();
    }

    @Override
    public T getData() {
        if (data.get() == null) {
            synchronized (data) {
                if (data.get() == null) {
                    data.set(expensiveData.get());
                }
            }
        }
        return data.get();
    }
}
```

LATE EVALUATION

```
class MyClass {
    // Property must be initialized
    val name: String
}
```

Use cases:

- Unit tests.

- Dependency Injection frameworks, like Spring.

SOLUTION ===> ***lateinit*** keyword.

```
class MyClass {
    // It must be a non-nullable and mutable data type
    lateinit var name: String
}
```

An *UninitializedPropertyAccessException* is thrown if trying to access a lateinit variable without initializing it.

THE SINGLETON PATTERN

- Well-known Java implementation.

- It is not thread-safe.

```java
public final class Connection {

    private static Connection INSTANCE;
    private String name = "my database";

    private Connection() { }

    public static Connection getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new Connection();
        }

        return INSTANCE;
    }
}
```

In Kotlin

```kotlin
/* Singleton pattern */
object Connection2 {
    val name: String = "my database"
}
```

```kotlin
val connection = Connection2
val connection2 = Connection2
println("Only one object ?: ${connection === connection2}")          // true
println("Only one name?: ${connection.name === connection2.name}")   // true
```

Access the Kotlin singleton from Java

```java
// only way if Kotlin code is not annotated
var name = Connection2.INSTANCE.getName();
System.out.println(name); // my database

// possible only if Kotlin code is annotated
var name2 = Connection2.getName();
System.out.println(name2); // my database
```

Annotation @JvmStatic ===> // static getter and setter methods are generated

```kotlin
object Connection2 {
    @JvmStatic
    val name: String = "my database"
}
```

STATIC FACTORY METHODS

```
class Player private constructor(val id: Long, val name: String) {
    // Consistency of the next id generation is guaranteed
    // because a companion object is a singleton.
    companion object {
        private var currentId = 0L;
        fun newInstance(name: String) = Player(currentId++, name)
    }
}
```

```
val juan = Player.newInstance("Juan")
```

- A companion object is an instance of a real class called "Companion".
- Each class can have only one companion object.

ACCESS FROM JAVA

```kotlin
class Player private constructor(val id: Long, val name: String) {

    companion object {

        @JvmField
        val magicWord = "Please"

        private var currentId = 0L;

        @JvmStatic
        fun newInstance(name: String) = Player(currentId++, name)
    }
}
```

```java
// only way if Kotlin code is not annotated
var player = Player.Companion.newInstance("Juan");
// possible only if Kotlin code is annotated
var player2 = Player.newInstance("Juanito");

var word = Player.magicWord;
System.out.println(word);   // Please
```

THE BUILDER PATTERN

```java
public record Programmer(String name, String lastName, String preferredLanguage) {

    // Builder
    public static final class Builder {
        String name;
        String lastName;
        String preferredLanguage;

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

        public Programmer build() {
            return new Programmer(name, lastName, preferredLanguage);
        }
    }
}
```

```java
var programmer = new Programmer.Builder("Juan")
        .lastName("Pérez")
        .preferredLanguage("Python")
        .build();
```

In Kotlin, default parameter values and named arguments make the builder pattern not needed.

```kotlin
data class Programmer(
    // All optional fields must be initialized
    val name: String = "programmer",
    val lastName: String? = null,
    val preferredLanguage: String = "Kotlin"
)
```

INLINE FUNCTIONS

- When using inline functions, the compiler inlines the function body.
- That is, it substitutes the body directly into places where the function gets called.

```kotlin
inline fun <T> Collection<T>.each(block: (T) -> Unit) {
    for (e in this) block(e)
}
```

```kotlin
val list = listOf(2, 3, 4)
val anotherList = list + 5
anotherList.each { print("$it, ") }  // 2, 3, 4, 5,
```

In Java

- Inlining is a way to optimize compiled source code at runtime by replacing the invocations of the most often executed methods with its bodies.

- It’s not performed by the traditional javac compiler, but by the JVM itself. To be more precise, it’s the responsibility of the Just-In-Time (JIT) compiler, which is a part of the JVM.

KEEPING GENERIC TYPE INFORMATION

- Java and Kotlin erases the generic type information at runtime.
- The compiler can reify generic type information for inline functions.
- Just mark the type parameter with the ***reified*** keyword:

```
inline fun <reified T> Any.isA(): Boolean = this is T
```

```
println("hola".isA<Int>())     // false
println("hola".isA<String>())  // true
```

KOTLIN PRINCIPLES

- Concise, flexible

- Powerful language features

- One of its main goals is to increase developer productivity and

- Compatibility with existing Java tools (Maven, Gradle and other JVM tools).

- Easy integration with Java code.

MISSING FEATURES IN THIS TALK

- Type-safe builders
- Infix Functions
- Operator Overloading
- Domain Specific Languages
- Coroutines
- "Real" immutability
- Scope functions (let, also, apply, run)

GRACIAS

- ……..

CLASSES

Nested classes aren’t inner by default: they don’t contain an implicit reference to their outer class.

I've described four Kotlin features that I miss in Java in this post: immutable references, null safety, extension functions, and reified generics