Kotlin

Introduction

- Statically typed programming language for the JVM, Android and the browser.
- 100% interoperable with Java.
- Created by JetBrains, the company behind IntelliJ IDEA and other (sweet) tools.
- Intended for industry use.
- Open source.

Features

- *Concise* to reduce the amount of boilerplate code you need to write.
- *Expressive* to make your code more readable and understandable.
- *Safe* to avoid entire classes of errors such as null pointer exceptions.
- *Versatile* for building server-side applications, Android apps or frontend code running in the browser.
- *Interoperable* to leverage existing frameworks and libraries of the JVM with 100 percent Java interoperability.

Hello World!

```
fun main(args: Array<String>): Unit {
    println("Hello, World!")
}
```

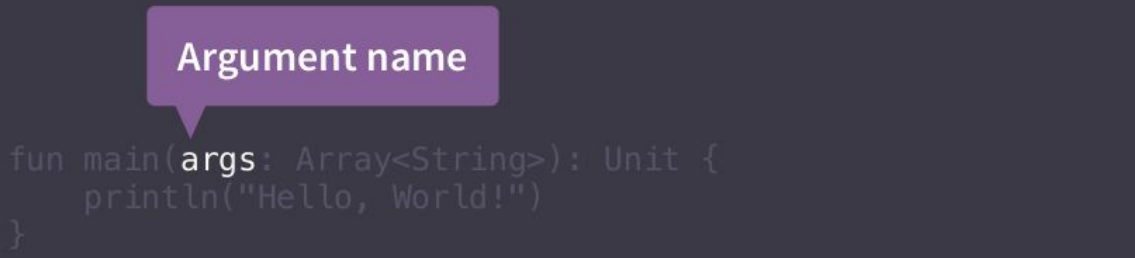
```
> Hello, World!
```

Function keyword

```
fun main(args: Array<String>): Unit {
    println("Hello, World!")
}
```

Function name

```
fun main(args: Array<String>): Unit {
    println("Hello, World!")
}
```


Argument name
fun main(args: Array<String>): Unit {
println("Hello, World!")
}

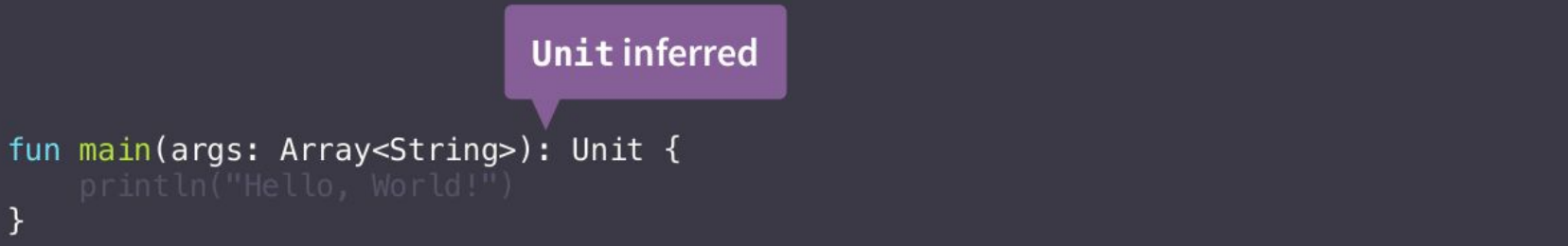
Argument type

```
fun main(args: Array<String>): Unit {
    println("Hello, World!")
}
```

Return type

```
fun main(args: Array<String>): Unit {
    println("Hello, World!")
}
```

```
fun main(args: Array<String>): Unit {
    println("Hello, World!")
}
```


Unit inferred
fun main(args: Array<String>): Unit {
println("Hello, World!")
}

```
fun main(args: Array<String>) {
    println("Hello, World!")
}
```

```
fun main(args: Array<String>) {
    println("Hello, World!")
}
```

```
fun main(args: Array<String>) {
    var name = "World"
    println("Hello, $name!")
}
```

Variable declaration

```
fun main(args: Array<String>) {
    var name = "World"
    println("Hello, $name!")
}
```

String interpolation

```
fun main(args: Arr
    var name = "Worl"
    println("Hello, $name!")
}
```

```
fun main(args: Array<String>) {
    var name = "World"
    if (args.isNotEmpty()) {
        name = args[0]
    }

    println("Hello, $name!")
}
```

```
fun main(args: Array<String>) {
    var name = "World"
    if (args.isNotEmpty()) {
        name = args[0]
    }

    println("Hello, $name!")
}
```

```
fun main(args: Array<String>) {
    val name = "World"
    if (args.isNotEmpty()) {
        name = args[0]
    }

    println("Hello, $name!")
}
```

Constant declaration

```
fun main(args: Array<String>) {
    val name = "World"
    if (args.isNotEmpty()) {
        name = args[0]
    }

    println("Hello, $name!")
}
```

```
fun ma
    va
    if (args.isNotEmpty()) {
        name = args[0]
    }

    println("Hello, $name!")
}
```

Val cannot be reassigned

```
fun main(args: Array<String>) {
    val name = "World"
    if (args.isNotEmpty()) {
        name = args[0]
    }

    println("Hello, $name!")
}
```

```
fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) {
        args[0]
    } else {
        "World"
    }

    println("Hello, $name!")
}
```

```
fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) {
        args[0]
    } else {
        "World"
    }

    println("Hello, $name!")
}
```

Conditional assignment block

```
fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) {
        args[0]
    } else {
        "World"
    }

    println("Hello, $name!")
}
```

```
fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) { args[0] } else { "World" }
    println("Hello, $name!")
}
```

```
fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) args[0] else "World"
    println("Hello, $name!")
}
```

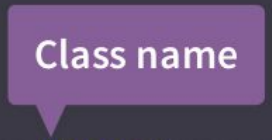
```
class Person(var name: String)

fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) args[0] else "World"
    println("Hello, $name!")
}
```

Class keyword

```
class Person(var name: String)

fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) args[0] else "World"
    println("Hello, $name!")
}
```




```
class Person(var name: String)

fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) args[0] else "World"
    println("Hello, $name!")
}
```

Primary constructor

```
class Person(var name: String)

fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) args[0] else "World"
    println("Hello, $name!")
}
```

Non-final class member

```
class Person(var name: String)

fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) args[0] else "World"
    println("Hello, $name!")
}
```

```
class Person(var name: String)

fun main(args: Array<String>) {
    val name = if (args.isNotEmpty()) args[0] else "World"
    println("Hello, $name!")
}
```

```
class Person(var name: String)

fun main(args: Array<String>) {

    println("Hello, $name!")
}
```

```
class Person(var name: String)

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, $name!")
}
```

```
class Person(va

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, $name!")
}
```

Instance declaration

```
class Person(var name: String)

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, $name!")
}
```

```
class Person(var name: String)

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, ${person.name}!")
}
```

```
> Hello, Michael!
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String)

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, ${person.name}!")
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String)

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, ${person.name}!")
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language)

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, ${person.name}!")
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN)

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, ${person.name}!")
}
```

Default value

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN)

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, ${person.name}!")
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, ${person.name}!")
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    val person = Person("Michael")
    println("Hello, ${person.name}!")
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    val person = Person("Michael")

}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    val person = Person("Michael")
    person.greet()
}
```

```
> Hello, Michael!
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {


}
```

```kotlin
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    val people = listOf(
        Person("Michael"),
        Person("Miguel", Language.SP),
        Person("Michelle", Language.FR)
    )
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    val people = listOf(
        Person("Michael"),
        Person("Miguel", Language.SP),
        Person("Michelle", Language.FR)
    )

    for (person in people) {
        person.greet()
    }
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    val people = listOf(
        Person("Michael"),
        Person("Miguel", Language.SP),
        Person("Michelle", Language.FR)
    )

    people.forEach { person ->
        person.greet()
    }
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    val people = listOf(
        Person("Michael"),
        Person("Miguel", Language.SP),
        Person("Michelle", Language.FR)
    )

    people.forEach { it.greet() }
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    listOf(
        Person("Michael"),
        Person("Miguel", Language.SP),
        Person("Michelle", Language.FR)
    ).forEach { it.greet() }
}
```

```
> Hello, Michael!
> Hola, Miguel!
> Bonjour, Michelle!
```

Non-final

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

open class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

fun main(args: Array<String>) {
    listOf(
        Person("Michael"),
        Person("Miguel", Language.SP),
        Person("Michelle", Language.FR)
    ).forEach { it.greet() }.
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

open class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

class Hispanophone(name: String) : Person(name, Language.ES)
class Francophone(name: String) : Person(name, Language.FR)

fun main(args: Array<String>) {
    listOf(
        Person("Michael"),
        Person("Miguel", Language.SP),
        Person("Michelle", Language.FR)
    ).forEach { it.greet() }
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

open class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

class Hispanophone(name: String) : Person(name, Language.ES)
class Francophone(name: String) : Person(name, Language.FR)

fun main(args: Array<String>) {
    listOf(
        Person("Michael"),
        Hispanophone("Miguel"),
        Francophone("Michelle")
    ).forEach { it.greet() }
}
```

```
enum class Language(val greeting: String) {
    EN("Hello"), ES("Hola"), FR("Bonjour")
}

open class Person(var name: String, var lang: Language = Language.EN) {
    fun greet() = println("${lang.greeting}, $name!")
}

class Hispanophone(name: String) : Person(name, Language.ES)
class Francophone(name: String) : Person(name, Language.FR)

fun main(args: Array<String>) {
    listOf(
        Person("Michael"),
        Hispanophone("Miguel"),
        Francophone("Michelle")
    ).forEach { it.greet() }
}
```

Features

Type inference

```
class Dog {}

fun main(args: Array<String>) {

    var string: String = ""
    var inferredString = ""

    var int = 0
    var long = 0L
    var float = 0F
    var double = 0.0
    var boolean = true

    var dog: Dog = Dog()
    var inferredDog = Dog()

}
```

Null safety

```java
// java
String a = null;
System.out.println(a.length());
```

Method invocation 'length' may produce 'java.lang.NullPointerException' more... (⌘F1)

```java
// java
String a = null;
int length = a != null ? a.length() : -1;
```

```kotlin
// kotlin
val a: String = null
```

Null can not be a value of a non-null type String

```kotlin
val b: String? = null
println(b?.length) // safe, returns null if b is null
println(b!!.length) // unsafe, throws exception if b is null
```

```kotlin
// kotlin
val a: String? = null
var length = if (a != null) a.length else -1 // note that "a?" is not needed
var lengthElvisOperator = a?.length ?: -1 // "elvis operator"
```

Null safety

```
class Employee(val name: String, var department: Department? = null)
class Department(var head: Employee? = null)  // default value
```

```
val finances = Department()
val sales = Department()

val boss = Employee("boss", finances)
val john = Employee("john", finances)
val bob = Employee("bob", department = sales) // named parameter

finances.head = boss

println(john.department?.head?.name) // boss
println(bob.department?.head?.name) // null
```

Smart casts

```java
// java
Object word = "word";
if (word instanceof String) {
    System.out.println(((String) word).length());
}
```

```kotlin
// kotlin
val word: Any = "word"
if (word is String) {
    println(word.length)
}
```

Smart cast to kotlin.String

String templates

```
val apples = 4
println("I have " + apples + "apples.")
println("I have $apples apples.")

val bananas = 3
println("I have $apples and " + (apples + bananas) + " fruits. ")
println("I have $apples and ${apples + bananas} fruits. ")

class Dog(val color: String)
val dog = Dog("black")
println("The dog is ${dog.color}")
```

Range expressions

```
val i = 5

if (1 <= i && i <= 10) {
    println(i)
}

if (IntRange(1, 10).contains(i)) {
    println(i)
}

if (1.rangeTo(10).contains(i)) {
    println(i)
}

if (i in 1..10) {
    println(i)
}
```

```
for (j in 1..4) {
    print(j) // output: 1234
}

for (j in 1..10 step 2) {
    print(j) // output: 13579
}

for (j in 10 downTo 1 step 2) {
    print(j) // output: 108642
}
```

Higher-order functions & lambdas

A higher-order function is a function that:

- takes functions as parameters, or
- returns a function.

A lambda expression or an anonymous function is a "function literal", i.e. a function that is not declared, but passed immediately as an expression.

Higher-order functions & lambdas

```
fun <T> filter(items: Collection<T>, f: (T) -> Boolean): List<T> {
    val filteredArray = arrayListOf<T>()
    for (item in items)
        if (f(item)) filteredArray.add(item)
    return filteredArray
}

filter(listOf(8, 2, 5, 9), { number ->
    number > 6
})

filter(listOf(8, 2, 5, 9), {
    it > 6
})

filter(listOf(8, 2, 5, 9)) {
    it > 6
}

filter(listOf(8, 2, 5, 9)) { it > 6 }
```

Extension functions

```
fun String.removeWhitespaces(): String {
    return this.replace(" ", "")
}
println("Using Kotlin Extensions".removeWhitespaces()) // "UsingKotlinExtensions"
```

```
fun Date.chechTalkToday(): Boolean {
    return day == 5
}
if (Date().chechTalkToday()) {
    println("today is friday")
}
```

Properties

```
class User {
    var name: String = "" // getter & setter
    val email: String // only getter
        get() = "${name.removeWhitespaces()}@infor.com" // custom getter
}

val user = User()
user.name = "john doe"
println(user.email) // johndoe@infor.com
```

Singletons & companion objects

```
object Logger {

    private val logger = LoggerFactory.getLogger("SingletonLogger")

    fun i(message: String) {
        logger.info(message)
    }
}

Logger.i("log")
```

```
class StartBusinessDayView: AuthorizedView() {

    companion object {
        val NAME = "Start Business Day"
    }
}

println(StartBusinessDayView.NAME)
```

Operator overloading

`+a`	`a.unaryPlus()`
`-a`	`a.unaryMinus()`
`!a`	`a.not()`
`a++`	`a.inc()`
`a--`	`a.dec()`
`a + b`	`a.plus(b)`
`a - b`	`a.minus(b)`

`a in b`	`a.contains(b)`
`a !in b`	`!a.contains(b)`
`a[i]`	`a.get(i)`
`a[i] = b`	`a.set(b)`
`a += b`	`a.plusAssign(b)`
`a -= b`	`a.minusAssign(b)`
`a > b`	`a.compareTo(b) > 0`

Data classes

We frequently create a class to do nothing but hold data.

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

// toString() of the form "User(name=John, age=1)"
// equals()
// hashCode()
// copy()

val bob = User(name = "Bob", age = 1)
val olderBob = bob.copy(age = 2)
```

Delegated properties

```
// lazy
val lazyValue: String by lazy {
    println("computed!")
    "lazy computed value"
}

println(lazyValue)
println(lazyValue)

// computed!
// lazy computed value
// lazy computed value
```

```
// observable
class User {
    var name: String by observable("initial value") { _, old, new ->
        println("$old -> $new")
    }
}

val user = User()
user.name = "first name"
user.name = "second name"

// initial value -> first name
// first name -> second name
```

Resources

https://kotlinlang.org/docs/reference/

https://kotlinlang.org/docs/reference/idioms.html

https://kotlinlang.org/docs/reference/java-interop.html

https://kotlinlang.org/docs/reference/java-to-kotlin-interop.html

http://try.kotlinlang.org/

https://kotlinlang.org/docs/tutorials/koans.html

https://kotlinlang.org/docs/reference/using-maven.html

¿?