

Introduction à Kotlin

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-01

Table des matières

1	Introduction à Kotlin	1
2	Kotlin en Action	1
2.1	Différences Clés avec Java	1
2.2	Expressions bodies	2
3	Getters et Setters en Kotlin	3
4	Membres Statiques en Kotlin	3
4.1	Null Safety	4
5	Extensions Functions	4
6	Smart Casts	4
7	Coroutines	5
8	Classe en Kotlin	5
8.1	Exemple Complet	6
8.2	Points Clés	6
9	Interface en Kotlin	6
10	Genericité en Kotlin	6
11	Propriétés en Kotlin	7
12	Interopérabilité avec Java	7
12.1	Principes Fondamentaux	7
12.2	De Java vers Kotlin	7
12.3	De Kotlin vers Java	8
12.4	Considérations Importantes	8
13	Apprendre Kotlin	8

1 Introduction à Kotlin

- Langage moderne créé par JetBrains
- 100% interopérable avec Java
- Fonctionne sur la JVM
- Adopté officiellement pour Android
- Réduit significativement le boilerplate code

2 Kotlin en Action

```
// Java
public class Person {
    private String name;
    private int age;

    public Person(String name, int
age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }
}
```

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

// Usage
val person = Person("John", 30)
println(person.name)
```

2.1 Différences Clés avec Java

1. Déclaration de Variables

- Type après le nom : `val name: String` (Kotlin) vs `String name` (Java)

- Variables immuables : `val` (final) vs mutables : `var`

```
val name: String = "John" //
Immutable (comme final en Java)
var age: Int = 25 //
Mutable (peut être modifié)
```

2. Syntaxe Simplifiée

- Point-virgule optionnel en fin de ligne
- Inférence de type avec `val` et `var`

```
val message = "Hello" //
Type String inféré
var count = 0 // Type
Int inféré
```

3. Data Classes

- Génération automatique des méthodes utilitaires

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

val user = User("John", 30)
val copy = user.copy(age = 31) //
Copie avec modification
val (name, age) = user //
Destructuration
```

4. Null Safety

- Types non-nullables par défaut
- Opérateurs de sécurité pour les nullables

```
var name: String = "John" //
Non-nullable
var nullableName: String? =
null // Nullable
val length
= nullableName?.length ?: 0 //
Safe call avec Elvis
```

5. Expression Bodies

- Fonctions en une ligne
- Propriétés calculées

```
fun double(x: Int) = x * 2
val isAdult get() = age >= 18
```

2.2 Expressions bodies

```
class Calculator {
    // Expression body avec type de
    retour explicite
    fun add(a: Int, b: Int): Int = a
    + b

    // Expression body avec type inféré
    fun multiply(a: Int, b: Int) = a
    * b

    // Expression body pour une
    propriété en lecture seule
    val version = "1.0"

    // Expression body pour une
    propriété calculée
    val isReady: Boolean
    get() = true

    // Expression body
    pour une propriété avec accesseurs
    personnalisés
    var lastOperation: String = ""
    private set
    get() = "Last: $field"
}

fun main() {
    val calc = Calculator()
    println("2 + 3 = ${calc.add(2,
3)}")
    println("2 * 3 =
${calc.multiply(2, 3)}")
    println("Version:
${calc.version}")
    println("Ready: ${calc.isReady}")
}
```

```
main()
```

3 Getters et Setters en Kotlin

En Kotlin, la gestion des propriétés de classe est grandement simplifiée par rapport à Java. Par défaut, chaque propriété déclarée génère automatiquement un getter et un setter (pour les propriétés mutables `var`). Kotlin permet également de personnaliser ces accesseurs pour ajouter de la logique métier, des validations, ou des calculs. Les propriétés peuvent être en lecture seule (`val`), avoir des accesseurs personnalisés, utiliser des backing fields, ou même être calculées à la demande. Cette approche réduit considérablement le code boilerplate tout en maintenant un niveau élevé d'encapsulation.

```
class Temperature {
    // Propriété avec getter/setter
    par défaut
    var celsius: Double = 0.0

    // Propriété avec getter/setter
    personnalisé
    var fahrenheit: Double = 32.0
    get() = celsius * 9/5 + 32
    set(value) {
        celsius = (value - 32)
        * 5/9
    }

    // Propriété en lecture seule
    (getter uniquement)
    val kelvin: Double
    get() = celsius + 273.15

    // Backing field protégé avec
    getter public
    private var _data: String = ""
    val data: String
    get() = _data
}
```

```
}

// Usage
fun main() {
    val temp = Temperature()
    temp.celsius = 25.0
    println("Fahrenheit:
    ${temp.fahrenheit}") // 77.0
    println("Kelvin:
    ${temp.kelvin}") // 298.15
}
```

4 Membres Statiques en Kotlin

En Kotlin, il n'existe pas de mot-clé `static` comme en Java. À la place, Kotlin propose deux mécanismes principaux pour implémenter des fonctionnalités similaires :

1. Le **companion object** qui permet de définir des membres liés à une classe (équivalent aux méthodes/propriétés statiques Java)
2. La déclaration **object** qui crée un singleton avec tous ses membres « statiques »

Ces deux approches offrent plus de flexibilité que le simple mot-clé `static` de Java, tout en permettant une meilleure encapsulation et la possibilité d'implémenter des interfaces. Pour l'interopérabilité avec Java, l'annotation `@JvmStatic` permet de générer de véritables membres statiques.

```
// Companion object - équivalent des
membres statiques
class MathUtils {
    companion object {
        const val PI = 3.14159

        fun square(x: Int): Int = x
        * x
    }
}
```

```

    @JvmStatic // Pour utilisation
    depuis Java
    fun cube(x: Int): Int = x *
x * x
    }
}

// Object declaration - Singleton
avec membres statiques
object Configuration {
    const val VERSION = "1.0.0"
    var apiKey: String = ""

    fun initialize() {
        println("Configuration
initialized")
    }
}

// Usage
fun main() {
    println(MathUtils.PI)
    println(MathUtils.square(4))

    Configuration.apiKey = "xyz"
    Configuration.initialize()
}

```

4.1 Null Safety

```

//| error: true
// Variables non-nullables par défaut
var name: String = "John"
//name = null // Erreur de compilation

```

```

// Variables nullables explicites
var nullableName: String? = "John"
nullableName = null // OK

// Safe calls
println(nullableName?.length)

```

```

// Elvis operator
val len = nullableName?.length ?: 0

```

Avantages

- NullPointerException évités à la compilation
- Gestion explicite des nulls
- Opérateurs de sécurité intégrés
- Vérification statique du null-safety

5 Extensions Functions

```

// Extension function
fun String.addHello(): String {
    return "Hello $this"
}

// Usage
val name = "John"
println(name.addHello())
// Output: Hello John

// Extension property
val String.lastChar: Char
    get() = this[length - 1]

```

Bénéfices

- Étendre des classes existantes
- Améliorer la lisibilité
- Organiser le code
- Réutilisation facilitée

6 Smart Casts

```

fun printLength(obj: Any) {
    if (obj is String) {
        // obj est automatiquement
        casté en String
        println(obj.length)
    }
}

// Pattern Matching avec when

```

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

Avantages

- Casts automatiques sûrs
- Pattern matching puissant
- Vérification exhaustive
- Code plus concis

7 Coroutines

```
// Suspend function
// Fonction asynchrone
suspend fun fetchUser(): User {
    delay(1000) // non bloquant
    return User("John")
}

// Usage
// Appel asynchrone
GlobalScope.launch {
    val user = fetchUser()
    println(user)
}

// Flow
// Flux de données asynchrone
flow {
    for (i in 1..3) {
        delay(100)
        emit(i)
    }
}.collect { value ->
    println(value)
}
```

Caractéristiques

- Asynchrone léger
- Séquentiel en apparence
- Annulation structurée
- Gestion des erreurs simplifiée

8 Classe en Kotlin

Types de Classes

```
// Classe standard
class User(val name: String)

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

// Classe abstraite
abstract class Shape {
    abstract fun area(): Double
}

// Object (singleton)
object Config {
    const val VERSION = "1.0"
}

// Companion object
class Factory {
    companion object {
        fun create() = Factory()
    }
}
```

Caractéristiques

- Classes `final` par défaut
- Constructeur primaire dans l'en-tête
- Initialiseurs avec `init`
- Propriétés dans le constructeur
- Héritage explicite avec `open`
- Délégation avec `by`
- Visibilité par défaut: `public`

8.1 Exemple Complet

```
open class Animal(val name: String) {
    open fun makeSound() = "..."
}

class Dog(
    name: String,
    val breed: String
) : Animal(name) {
    override fun makeSound() = "Woof!"

    init {
        println("Dog $name created")
    }
}

// Usage
fun main() {
    val dog = Dog("Rex", "Labrador")
    println("${dog.name} says: ${dog.makeSound()}")
}
```

8.2 Points Clés

1. Constructeur Primaire

- Partie de la déclaration de classe
- Initialiseurs avec `init`
- Paramètres peuvent être propriétés

2. Héritage

- Classes `final` par défaut
- Mot-clé `open` pour permettre l'héritage
- `override` explicite obligatoire

3. Visibilité

- `public` par défaut
- `private`, `protected`, `internal`
- S'applique aux classes et membres

9 Interface en Kotlin

```
// Interface basique
interface Movable {
    fun move()
```

```
    val speed: Int

    // Méthode avec implémentation
    // par défaut
    fun describe() = "Moving at $speed"
}

// Interface avec propriété par
// défaut
interface Named {
    val name: String
    get() = "Unknown"
}

// Implémentation
class Car : Movable, Named {
    override val speed = 100
    override val name = "Ferrari"
    override fun move() =
        println("Vrooom!")
}
```

Caractéristiques

- Peuvent contenir des propriétés abstraites
- Méthodes par défaut possibles
- Implémentation multiple autorisée
- Pas d'état (pas de champs)
- Délégation possible avec `by`

10 Genericité en Kotlin

```
// Classe générique basique
class Box<T>(var content: T)

// Variance déclarée
interface Source<out T> {
    fun next(): T
}

// Contraintes de type
class NumberBox<T> : Number<>(var value: T)
```

```
// Type projection
fun copy(from: Array<out Any>, to:
Array<Any>) {
    from.forEachIndexed { i, e ->
to[i] = e }
}

// Fonctions génériques
fun <T> printBox(box: Box<T>) {
    println(box.content)
}
```

Concepts Clés

- Variance: in/out
- Contraintes de type
- Projections de type
- Star projections *
- Reified type parameters

11 Propriétés en Kotlin

```
class User {
    // Propriété simple
    var name: String = ""

    // Propriété en lecture seule
    val id: Int = 1

    // Propriété avec accesseurs
    var age: Int = 0
        get() = field
        set(value) {
            if (value >= 0) field
= value
        }

    // Propriété calculée
    val isAdult: Boolean
        get() = age >= 18

    // Propriété avec délégation
    val lazy: String by lazy {
        println("Computing...")
    }
```

```
"Lazy Value"
}
}
```

Fonctionnalités

- Backing field avec `field`
- Accesseurs personnalisables
- Délégation de propriétés
- Late initialization `lateinit`
- Propriétés calculées

12 Interopérabilité avec Java

12.1 Principes Fondamentaux

i Interopérabilité Bidirectionnelle

- Kotlin et Java peuvent coexister dans le même projet
- Appels transparents entre les deux langages
- Pas besoin d'adaptateurs ou de wrappers
- Compilation vers le même bytecode Java

12.2 De Java vers Kotlin

💡 Utilisation de Java en Kotlin

1. Accès aux Classes Java

- Import direct des classes Java
- Utilisation naturelle des API Java
- Support complet des frameworks Java
- Conversion automatique des collections

2. Adaptations Automatiques

- Getters/Setters → Propriétés Kotlin
- Interfaces SAM → Lambdas Kotlin
- Checked Exceptions → Gestion optionnelle
- Types primitifs → Types Kotlin

12.3 De Kotlin vers Java

! ☒ Utilisation de Kotlin en Java

1. Fonctionnalités Kotlin en Java

- Properties → Getters/Setters
- Extension functions → Méthodes statiques
- Data classes → Classes Java standard
- Named parameters → Arguments standards

2. Annotations Spéciales

- `@JvmStatic` pour méthodes statiques
- `@JvmField` pour champs publics
- `@JvmOverloads` pour paramètres optionnels
- `@JvmName` pour noms personnalisés

- Kotlin Playground
- IntelliJ IDEA
- SDKMan: `sdk install kotlin`
- Learning by exemple

12.4 Considérations Importantes

⚠ ⚠ Points d'Attention

1. Null Safety

- Types nullable Kotlin → Types Java standards
- Types Java → Platform Types en Kotlin
- Vérifications de nullité à runtime

2. Collections

- Collections Kotlin → Collections Java
- Mutabilité préservée
- Conversions automatiques

3. Spécificités

- Extensions Kotlin → Méthodes statiques Java
- Coroutines → Compatibilité avec futures/callbacks
- Propriétés → Pattern JavaBean

13 Apprendre Kotlin

- Pour travailler :