Développement Android avec Kotlin

Séance 1 : Introduction à Kotlin

Jordan Hiertz

Contact

hiertzjordan@gmail.com

jordan.hiertz@al-enterprise.com

https://github.com/j-hiertz/Android-training

Organisation

- Objectifs pédagogiques
  - Acquérir les concepts du développement Android en Kotlin
  - Maitriser les bonnes pratiques du développement d'une application mobile
- Supports de cours envoyés
- Séances théoriques + Travaux pratiques
- Évaluation des TP + Examen final

Présentation de Kotlin

- Langage de programmation **orienté objet** et **fonctionnel.**
- Typage **statique** et **inféré.**
- **Interopérabilité** complète avec Java.
- **Multiplateforme** : Compilé pour la JVM, le JavaScript et de manière native pour iOS, Windows.
- Langage officiel d'Android depuis 2019.
- **Syntaxe concise et expressive**, réduisant la verbosité du code par rapport à Java

Avantages de Kotlin

Expressivité / concision => ~30% de code en moins que Java

https://pl.kotl.in/dpQA4g0SG?theme=darcula

Avantages de Kotlin

Orienté Objet

https://pl.kotl.in/X7iCdtdkR?theme=darcula

Avantages de Kotlin

- **Programmation fonctionnelle**
  - **Fonctions pures :** Pas d'effets secondaires
  - **Immutabilité :** Les données ne changent pas
  - **Fonctions de première classe :** Les fonctions sont traitées comme des valeurs
  - **Expressions** plutôt qu'instructions (`if` qui retourne un résultat)
  - **Récursion** plutôt que boucles pour traiter des structures répétitives

https://pl.kotl.in/CvavAvyV-?theme=darcula

Hello, world!

```kotlin
fun main() {
    println("Hello, world!")
}

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

fun main() = println("Hello, world!")
```

- **Point d'entrée** : La fonction `main` est le point d'entrée d'une application Kotlin.
- **Arguments** : Accepte un nombre variable d'arguments `String` (optionnels).
- **Affichage** : `println` affiche son paramètre et ajoute un saut de ligne.

Variables

```kotlin
val a: Int = 1 // Immutable (non modifiable après affectation)

var b = 2 // Mutable (modifiable après affectation) le type est inféré
b = a

val c: Int // Le type est requis si pas d'initialisation
c = 3 // Affectation différée

a = 4 // Error: Val cannot be reassigned

const val NAME = "Kotlin" // Calculé pendant la compilation
```

- Le type peut être inféré dans la plupart des cas.
- L'assignation peut être différée.
- Convention de nommage : majuscule pour les constantes.

Fonctions

```
1  fun sum(a: Int, b: Int): Int {
2      return a + b
3  }
4
5  fun mul(a: Int, b: Int) = a * b
6
7  fun greet(name: String = "world") {
8      println("Hello $world")
9  }
10
11 fun String.removeSpaces() = this.replace(" ", "")
```

- `fun` : Mot-clé pour définir une fonction
- Type de retour après `:` (peut être omis pour `Unit` ou inféré)
- Les arguments peuvent avoir des valeurs par défaut
- Utilisation des **arguments nommés** pour plus de clarté
- **Fonctions d'extension** : Ajout de méthodes à des classes existantes

Fonctions

https://pl.kotl.in/fYjfOcY9y?theme=darcula

Limites des Fonctions d'Extension en Kotlin

- **Pas d'accès aux membres privés ou protégés.**
- **Méthodes existantes prioritaires** : Les méthodes d'une classe ont la priorité.
- **Pas de polymorphisme** : Basé sur le type statique de l'objet, pas le type réel.
- **Limitées au scope** : Les extensions ne sont disponibles que dans le contexte où elles sont définies ou importées.

Flux de contrôle

`if/else` et `when`

- **Peut être une expression** : retourne une valeur
- **Peut être une instruction** : exécute simplement du code

```kotlin
val max = if (a > b) a else b

if (a > b) {
    println("a est plus grand que b")
} else {
    println("b est plus grand ou égal à a")
}

val result = when (value) {
    1 -> "Un"
    2 -> "Deux"
    else -> "Autre"
}

when (value) {
    1 -> println("Un")
    2 -> println("Deux")
    else -> println("Autre") // Peut être omis
}
```

Flux de contrôle

Boucles

- **`for`** : boucle sur une collection ou une range
- **`while`** et **`do-while`** : répète tant qu'une condition est vraie

```
1  val items = listOf("apple", "banana", "kiwifruit")
2
3  for (item in items) { // Boucle sur une collection
4      println(item)
5  }
6
7  for (i in 1..10) { // Boucle for avec une range
8      println(i)
9  }
10
11 while (x > 0) {
12     println(x)
13     x--
14 }
15
16 do { // Boucle do-while (exécute au moins une fois)
17     println(x)
18     x++
19 } while (x < 5)
```

Flux de contrôle

Boucles

- **Label**: permet de contrôler spécifiquement la boucle affectée par un **`break`** ou un **`continue`**
- **`continue`** : saute l'itération actuelle de la boucle
- **`break`** : termine la boucle entièrement

https://pl.kotl.in/D9OoeXUAI?theme=darcula

Null safety en Kotlin

Kotlin introduit un système permettant de **gérer explicitement les valeurs nulles**, réduisant les erreurs courantes comme les `NullPointerException`

```kotlin
val notNullText: String = "Definitely not null" // Ne peut jamais être nul
val nullableText1: String? = "Might be null"    // Peut être nul
val nullableText2: String? = null               // Est nul
```

En ajoutant **`?`**, on indique au compilateur que la variable peut contenir `null`

L'opérateur **`?:`** (appelé Elvis Operator) permet de fournir une valeur par défaut si l'expression est `null`

```kotlin
fun log(text: String?) {
    val toPrint = text ?: "Nothing to print :("
    println(toPrint)
}
```

Gestion des valeurs nulles

Opérateur `?.` (**Safe call**)

- Permet de **consommer une valeur nullable** sans lancer une exception si elle nulle.
- Si la valeur est nulle, l'expression entière devient `null`.

```
1 val nullableText: String? = "Hello"
2
3 // Si nullableText est non nul, renvoie sa longueur, sinon renvoie null
4 println(nullableText?.length)
```

Opérateur `!!` (**Assertion de Non-Null**)

- Force la conversion d'une valeur nullable en non nullable.
- Lève une **`NullPointerException`** si la valeur est nulle.

```
1 val nullableText: String? = "Hello"
2
3 // Lance une exception si nullableText est null
4 println(nullableText!!.length)
```

CheatSheet

Mutability

```
var mutableString: String = “Toto”
val immutableString: String = “Toto”
val inferredString = “Toto”
```

Numbers

```
var intNum = 10
val doubleNum = 10.0
val longNum = 10L
val floatNum = 10.0F
```

Booleans

```
var trueBoolean = true
val falseBoolean = false
```

Strings

```
var mutableString: String = “Toto”
val immutableString: String = “Toto”
val inferredString = “Toto”
```

Null Safety

```
val cannotBeNull: String = null // Compile erro
val canBeNull: String? = null // Valid

val cannotBeNull: Int = null // Compile error
val canBeNull: Int? = null // Valid
```

Safe Operator

```
val nullableLength: Int? = nullableString?.length
val chiefName: String? = person?.department?.head?.nam
```

List

```
val numbers: List<Int> = listOf(1, 2, 3)
val firstNumber = numbers[0] // Premier élément
```

Elvis Operator

```
val nonNullLength: Int = nullableString?.length ?: 0
val chiefName: String = person?.name ?: "Toto"
```

Set

```
// 3 est unique
val uniqueNumbers: Set<Int> = setOf(1, 2, 3, 3)
val containsOne = 1 in uniqueNumbers // true
```

Map

```
val map = mapOf("one" to 1,"two" to 2)
val oneValue = map["one"] // 1
```

CheatSheet

If/Else

```
val guests = 30

if (guests == 0) {
    println(“No guests”)
} else if (guests < 20) {
    println(“Small group”)
} else {
    println(“Large group”)
}
```

```
val isEven = if (num % 2 == 0) true else false
```

For

```
val pets = arrayOf("dog", "cat", "canary")
for (element in pets) {
    print(element + " ")
}
```

Range

```
 // Inclusif
for (i in 1..10) { println(i) }

// Exclusif
for (i in 1 until 10) { println(i) }
```

When

```
when (results) {
    0  -> println("No results")
    in 1..39 -> println("Got results!")
    else -> println("That's a lot of results!
}
```

```
val sign = when(x) {
    0 -> “Zero”
    in 1..4 -> “Four or less”
    else -> “Other numbers”
}
```

Fonctions

```
fun log(str: String = "") = println("log: $str")

// Extension function
fun String.isEvenLength(): Boolean = this.length

// Fonction lambda qui additionne deux nombres
val sum = { a: Int, b: Int -> a + b }
```

Expression Lambda en Kotlin

Une expression lambda est une fonction anonyme que l'on peut assigner à une variable.

```kotlin
val sum: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
val mul = { x: Int, y: Int -> x * y }
```

Convention Kotlin : Lambda en dernier paramètre

Quand le dernier paramètre d'une fonction est une fonction (lambda), la lambda peut être placée **en dehors des parenthèses** de l'appel de fonction

```kotlin
// Avec les parenthèses
val badProduct = items.fold(1, { acc, e -> acc * e })

// Sans les parenthèses
val goodProduct = items.fold(1) { acc, e -> acc * e }
```

Avantage : Améliore la lisibilité du code en réduisant le nombre de parenthèses.

Expression Lambda en Kotlin

Les expressions lambda sont très utiles dans Kotlin pour plusieurs raisons :

Fonctions anonymes : Crée des fonctions sans nom pour des utilisations temporaires.

```kotlin
val add = { x: Int, y: Int -> x + y }
```

Fonctions de haut niveau: Passe des comportements comme arguments ou retourne des fonctions.

```kotlin
fun operateOnNumbers(x: Int, y: Int, operation: (Int, Int) -> Int): Int
```

Syntaxe concise: Simplifie les opérations sur des collections.

```kotlin
val doubled = numbers.map { it * 2 }
```

Programmation réactive et fonctionnelle: Simplifie la manipulation de données

```kotlin
val sum = items.fold(0) { acc, item -> acc + item }
```

Programmation orientée objet

La **programmation orientée objet (POO)** est un paradigme de programmation basé sur la représentation d'un programme comme un ensemble d'**objets** et les **interactions** entre eux.

Classe et Objet

```
1 class Person(val name: String, var age: Int) {
2     fun introduce() {
3         println("Je m'appelle $name et j'ai $age ans.")
4     }
5 }
6
7 // Création d'un objet
8 val person = Person("Alice", 30)
9 person.introduce()  // Output : Je m'appelle Alice et j'ai 30 ans.
```

Classe : Une classe représente une entité abstraite avec des **attributs** (données, propriétés) qui stockent l'état, et des **méthodes** (fonctions) qui définissent le comportement.

Objet : Une instance concrète d'une classe. Chaque objet possède un état spécifique.

Invariant de classe

```
1  class BankAccount(private var balance: Int) {
2      init {
3          require(balance >= 0) { "Le solde initial ne peut pas être négatif." }
4      }
5
6      fun withdraw(amount: Int) {
7          require(amount >= 0) { "Le montant du retrait doit être positif." }
8          if (balance - amount >= 0) {
9              balance -= amount
10         } else {
11             throw IllegalArgumentException("Fonds insuffisants.")
12         }
13     }
14 }
```

Un **invariant** impose des **règles** sur l'état d'un objet, qui doivent toujours être respectées durant sa vie.

L'invariant est préservé avant et après chaque méthode.

Corollaire : Les **champs publics** peuvent briser les invariants en permettant une modification non contrôlée de l'état.

Abstraction

Les objets sont des **abstractions de données** avec des représentations internes, et des méthodes pour interagir avec ces représentations.

Il n’est pas nécessaire de **révéler** les détails de l’implémentation interne, qui peuvent rester cachés à l'intérieur de l'objet.

```
1 class Rectangle(private val width: Int, private val height: Int) {
2     fun area(): Int {
3         return width * height  // Détails internes cachés
4     }
5 }
6
7 val rectangle = Rectangle(5, 10)
8 println(rectangle.area())  // Output : 50
```

Encapsulation

- **Encapsulation** : Permet de regrouper des **données** avec des **méthodes** qui opèrent sur ces données, tout en cachant les détails d'implémentation à l'utilisateur.
- Un objet est une **boîte noire** : il accepte des messages et y répond d'une manière ou d'une autre.
- L'**interface** d'une classe et l'encapsulation sont liées : tout ce qui n'est pas dans l'interface est encapsulé.

```
1  class Car(private var fuel: Int) {
2      fun drive() { // Les détails internes (comme fuel) sont cachés
3          if (fuel > 0) {
4              fuel--
5              println("La voiture roule.")
6          } else {
7              println("Pas assez de carburant.")
8          }
9      }
10 }
```

Visibilité

Pour soutenir l'encapsulation, il existe des mécanismes permettant de **cacher l'état** et les méthodes du reste du programme.

La plupart des langages fournissent des **modificateurs d'accès** comme `public` et `private`.

Kotlin propose 4 modificateurs d'accès :

- `public` : accessible par tous
- `private` : accessible uniquement à l'intérieur de la classe
- `protected` : comme private, mais accessible aussi dans ses sous-classes
- `internal` : visible à l'intérieur du module (ensemble de fichiers sources compilés ensemble)

Héritage

L'**héritage** permet de définir une nouvelle classe basée sur une classe déjà existante, en conservant tout ou une partie des fonctionnalités de la classe de base (état/comportement).

- La classe héritée est appelée **classe de base** ou **classe parent**.
- La nouvelle classe est appelée **classe dérivée**, **classe enfant** ou **héritière**.

La classe dérivée respecte entièrement la spécification de la classe de base, mais peut avoir des **fonctionnalités supplémentaires** (état/comportement).

Motivation

- **Séparer le code partagé** dans la classe de base et le réutiliser dans les classes dérivées.
- **Hiérarchie des types et sous-typage** : Les classes dérivées appartiennent à la même hiérarchie que la classe de base
- L'héritage est parfois **redondant** et peut être remplacé par la **composition**

Sous-types

Héritage simple :

- Kotlin permet d'hériter d'une seule classe
- Évite les ambiguïtés liées à l'héritage multiple

D'autres langages (C++, Python) autorisent **l'héritage multiple**, cela peut entraîner des problèmes, comme le "*diamond problem*"

```
  A                 // Méthode show()
 / \
B   C               // Redéfinit show()
 \ /
  D                 // Quelle méthode show() doit-on utiliser ?
```

Polymorphisme

Le polymorphisme est un concept clé de la POO qui permet de travailler avec des objets via leurs interfaces, sans connaissance de leurs types spécifiques ni de leur structure interne.

- Les classes dérivées peuvent redéfinir et modifier le comportement hérité.

- Les objets peuvent être manipulés à travers les interfaces de leurs classes parentes
  - Le code client ne sait pas (et ne se soucie pas) s'il travaille avec la classe de base ou une sous-classe, ni ce qui se passe "à l'intérieur"

Polymorphisme

Principe de substitution de Liskov (LSP)

> *Si un objet d'un sous-type peut remplacer un objet de type parent sans changer le comportement d'un programme, alors ce sous-type respecte le LSP*

```
// Base class
open class Animal {
    open fun makeSound() {
        println("I am an animal!")
    }
}

// Subclass Dog
class Dog : Animal() {
    override fun makeSound() {
        println("Woof!")
    }
}

// Subclass Cat
class Cat : Animal() {
    override fun makeSound() {
        println("Meow!")
    }
}
```

```
// Function using Animal
fun makeAnimalSound(animal: Animal) {
    animal.makeSound()
}

val myDog = Dog()
val myCat = Cat()

makeAnimalSound(myDog) // Prints "Woof!"
makeAnimalSound(myCat) // Prints "Meow!"
```

Programmation objet en Kotlin

```kotlin
class SomeClass

fun main() {
    val someObject = SomeClass() // () here is constructor invocation
}
```

```kotlin
class Person(val name: String, val surname: String, private var age: Int) {

    init {
        findJob()
    }

    constructor(name: String, parent: Person) : this(name, parent.surname, 0)
}
```

Le constructeur primaire utilisé par défaut. S'il est vide on peut omettre les parenthèses.

Le constructeur secondaire

L'ordre d'initialisation : le constructeur primaire -> le bloc init -> le constructeur secondaire

Init blocs

https://pl.kotl.in/gUWRL45O9?theme=darcula

Abstraction

```
interface RegularCat {
    fun pet()
    fun feed(food: Food)

    // Doesn't compile
    // Property initializers
    // in interfaces are prohibited.
    val legsCount = 4
}

interface SickCat {
    fun checkStomach()
    fun giveMedicine(pill: Pill)
}
```

Les interfaces ne peuvent pas avoir d'état

```
abstract class RegularCat {
    abstract val name: String

    abstract fun pet()
    abstract fun feed(food: Food)

    val legsCount = 4 // ok
}

abstract class SickCat {
    abstract val location: String

    abstract fun checkStomach()
    fun giveMedicine(pill: Pill) {}
}
```

Les classes abstraites ne peuvent pas avoir d'instance mais peuvent avoir un état

```
interface UiObject {
    val title: String
}
```

Dans cet exemple, `title` n'est pas un état ; c'est une exigence pour toute classe implémentant cette interface d'avoir la propriété title.

Héritage

- Pour qu'une classe puisse être héritée, elle doit être marquée avec le mot clé `open`. (Les classes abstraites sont toujours ouvertes par défaut)
- Lors de l'héritage d'une classe, le constructeur de la classe parente doit être appelé.

```
1 open class ParentClass(val name: String)
2
3 class ChildClass(name: String, val age: Int) : ParentClass(name) {
4     // Constructeur de ParentClass est appelé
5 }
```

Pourquoi Kotlin utilise *final by default ?*

- Si une classe doit être utilisée par d'autres développeurs, elle doit être **conçue correctement** pour l'héritage et marquée explicitement avec `open`.

> *Design and document for inheritance, or else prohibit it.*
>
> *Joshua Bloch, Effective Java, Item 19*

Propriétés

Propriétés dans le constructeur :

- Utilisation rapide pour initialiser les propriétés lors de la création d'un objet.
- Pratique pour les valeurs immuables (`val`) ou les paramètres obligatoires.

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

Exemple d'utilisation :

```
1 val person = Person("John", 25)
2 println(person.name)  // Affiche "John"
```

Propriétés

Propriétés dans le corps de la classe :

- Idéale pour des valeurs par défaut ou pour des propriétés qui peuvent évoluer après l'initialisation de l'objet.

```
1 class Person {
2     var age: Int = 0
3 }
```

Exemple d'utilisation :

```
1 val person = Person()
2 person.age = 30
```

Propriétés

Getters/Setters personnalisés :

- Contrôler la manière dont les propriétés sont lues ou modifiées.
- Utilisation du **backing field** (`field`) pour gérer les comportements internes.

```
var name: String = "John"
    get() = field.uppercase()  // Personnaliser l'accès
    set(value) {
        field = value.trim()   // Modifier le comportement d'affectation
    }
```

Exemple d'utilisation :

```
val person = Person()
person.name = "   alice   "
println(person.name)  // Affiche "ALICE"
```

Propriétés

Visibilité des getters et setters :

- Protéger certaines propriétés de la modification tout en autorisant leur lecture.

```
var password: String = "secret"
    private set  // Le setter est privé, le getter est public
```

Exemple d'utilisation :

```
val account = Account()
println(account.password)  // Accessible en lecture
// account.password = "newpassword" -> Erreur, setter privé
```

Propriétés

Surcharge des getters/setters :

- Les propriétés peuvent être ouvertes ou abstraites, ce qui signifie que leurs getters et setters peuvent ou doivent être surchargés par les héritiers.

```
1 open class OpenBase(open val value: Int)
2
3 class Child(value: Int) : OpenBase(value) {
4     override var value: Int = 1000
5         get() = field - 7
6 }
```

Exemple d'utilisation :

```
1 fun main() {
2     val child = Child(500)
3     println("Initial value: ${child.value}") // Affiche 993 (1000 - 7)
4 }
```

Data class

Les **data classes** en Kotlin sont des classes dont le principal but est de stocker des données. Elles simplifient le code en générant automatiquement des méthodes utiles.

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

Fonctionnalités dérivées par le compilateur :

`toString()`

```kotlin
val user = User("John", 42)
println(user) // Affiche: User(name=John, age=42)
```

`equals()` et `hashcode()`

```kotlin
val user1 = User("John", 42)
val user2 = User("John", 42)
println(user1 == user2) // Affiche: true
```

`componentN()`

```kotlin
val (name, age) = user
println("Name: $name, Age: $age") // Affiche: Name: John, Age: 42
```

`copy()`

```kotlin
val updatedUser = user.copy(age = 43)
println(updatedUser) // Affiche: User(name=John, age=43)
```

Enum class

- Les **enum classes** en Kotlin sont utilisées pour définir un ensemble fixe de constantes. Elles sont particulièrement utiles pour représenter des valeurs qui ont un nombre limité d'options.

```
enum class Direction {
    NORTH, SOUTH, EAST, WEST
}
```

Les enum classes peuvent contenir des propriétés et des méthodes.

```
enum class Direction(val degrees: Int) {
    NORTH(0), EAST(90), SOUTH(180), WEST(270);

    fun isNorth(): Boolean {
        return this == NORTH
    }
}
```

Singleton en Kotlin

Le **singleton** est un patron de conception qui garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.

> *“ En général, un patron de conception est une solution typique à des problèmes courants dans la conception de logiciels.*

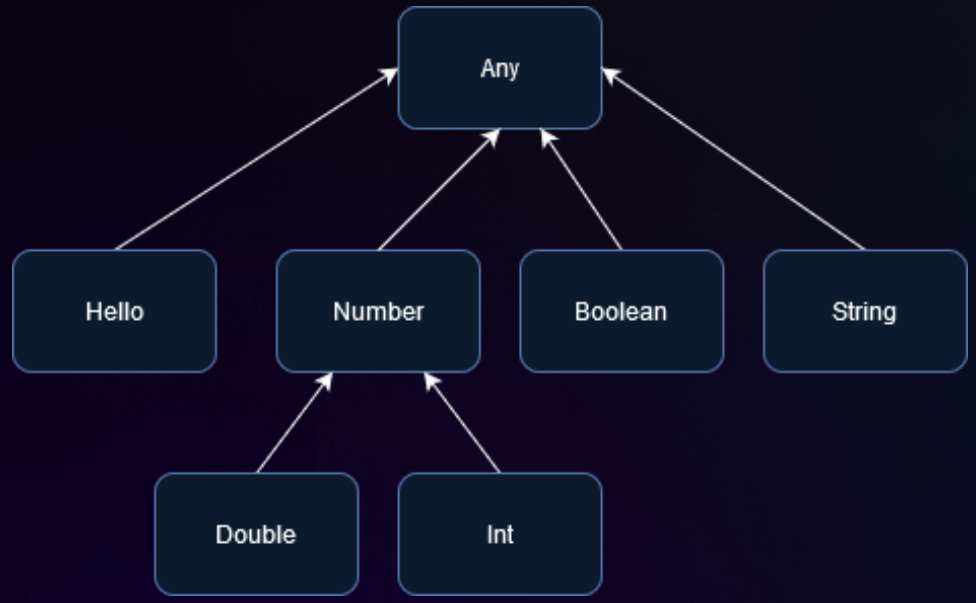
Déclaration d'un Singleton en Kotlin

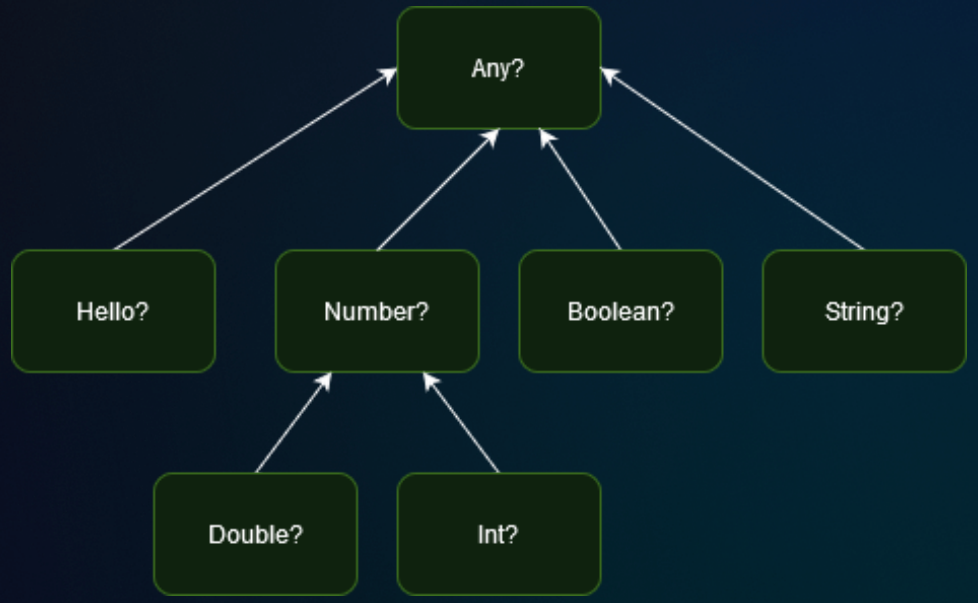
```kotlin
object DatabaseConnection {
    init {
        // Initialisation de la connexion à la base de données
    }

    fun connect() {
        println("Connection à la base de données établie.")
    }
}
```

Exemple d'utilisation :

```kotlin
DatabaseConnection.connect() // Affiche: Connection à la base de données établie.
```

Hiérarchie des types en Kotlin





Explorez les concepts avancés de Kotlin

- **Infix functions** : Utilisez une syntaxe lisible pour les appels de fonction. En savoir plus
- **Inline value classes** : Optimisez la gestion des données avec des classes légères En savoir plus
- **Selead classes** : Gérer des types restreints avec exhaustivité En savoir plus
- **Functional interfaces (SAM)** : Simplifiez l'utilisation des interfaces avec une seule méthode En savoir plus
- **Companion objects** : Créez des membres statiques dans des classes En savoir plus

Conclusion

- **Les bases du langage Kotlin** : Syntaxe, types de données et structures de contrôle
- **La programmation orientée objet** : Classes, objets, héritage et polymorphisme
- **Les spécificités de Kotlin** : Propriétés, data classes, enum...