

# Formation Kotlin I

by Labo Academy & Okono





# Introduction et installation des outils

## INTRODUCTION À KOTLIN

### 1. Présentation de Kotlin

- Historique
- Philosophie

### 2. Comparaison avec Java et autres langages

- Concision, Sécurité, Interopérabilité, Coroutines

### 3. Pourquoi apprendre Kotlin

- Adoption par Google
- Communauté croissante
- Polyvalence

BACK

NEXT



# Introduction et installation des outils

## INTRODUCTION À KOTLIN

### 1. Présentation de Kotlin

Kotlin est un langage de programmation moderne, introduit en 2011 par JetBrains, la société derrière IntelliJ IDEA, un IDE populaire pour Java. Kotlin est conçu pour être concis, sûr et compatible avec les bibliothèques Java existantes, ce qui permet aux développeurs d'intégrer facilement ce langage dans des projets Java existants.

Kotlin est désormais le langage privilégié pour le développement Android, ce qui en fait un choix incontournable pour les développeurs mobiles. Google l'a officiellement adopté comme langage principal pour Android en 2017, en raison de sa capacité à simplifier le développement et à améliorer la sécurité du code grâce à des fonctionnalités comme la null safety.

[BACK](#)[NEXT](#)



# Introduction et installation des outils

## INTRODUCTION À KOTLIN

### 2. Comparaison avec Java et autres langages

Kotlin offre plusieurs avantages par rapport à Java, notamment :

- **Concision** : Kotlin nécessite beaucoup moins de code pour accomplir les mêmes tâches.
- **Null Safety** : Kotlin élimine les erreurs liées aux objets nuls, couramment rencontrées en Java.
- **Interopérabilité** : Kotlin fonctionne parfaitement avec toutes les bibliothèques Java existantes.

[BACK](#)[NEXT](#)



# Introduction et installation des outils

## INTRODUCTION À KOTLIN

### 3. Pourquoi apprendre Kotlin

- **Adoption par Google:** Depuis 2017, Kotlin est le langage recommandé par Google pour le développement Android. De nombreuses entreprises ont déjà adopté Kotlin pour leurs applications, et la communauté de développeurs Kotlin est en pleine croissance. Ce livre vous guidera à travers tout le parcours pour devenir un expert Kotlin, depuis les bases du langage jusqu'à la création d'applications Android complètes.
- **Communauté croissante :** Support actif et nombreuses ressources (Des livres, formations, tuto, etc)
- **Polyvalence :** Utilisable pour le développement mobile, web, desktop et serveur.
- **UD :** C'est une langue principal de votre cours INF355.

[BACK](#)[NEXT](#)



# Introduction et installation des outils

## INSTALLATION DES OUTILS

### 1. Installer le JDK (Java Development Kit)

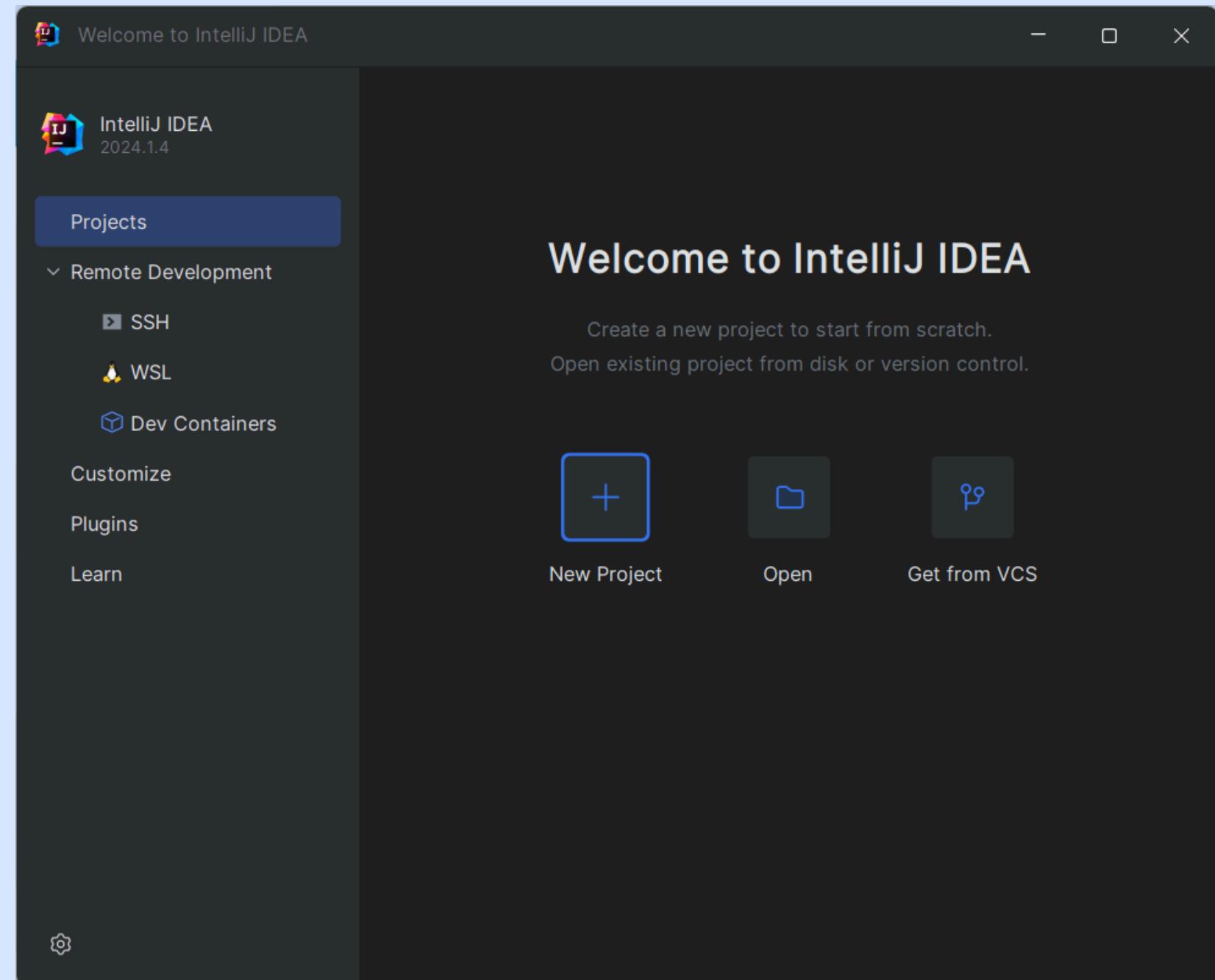
- Téléchargement
- Installation
- Configuration

### 2. Choisir un IDE (Environnement de Développement Intégré)

- IntelliJ IDEA
- Android Studio
- Gradle et Maven : Gestionnaires de build pour les projets Kotlin.

BACK

NEXT



The screenshot shows the IntelliJ IDEA 2024.1.4 welcome screen. The title bar says "Welcome to IntelliJ IDEA". The main area has a dark background with the text "Welcome to IntelliJ IDEA" and "Create a new project to start from scratch." Below this are three buttons: "New Project" (with a plus sign icon), "Open" (with a folder icon), and "Get from VCS" (with a gear icon). On the left, there's a sidebar with the "Projects" tab selected, showing "Remote Development" with options for "SSH" and "WSL", "Dev Containers", "Customize", "Plugins", and "Learn". At the bottom of the sidebar is a gear icon.

Premier projet Kotlin

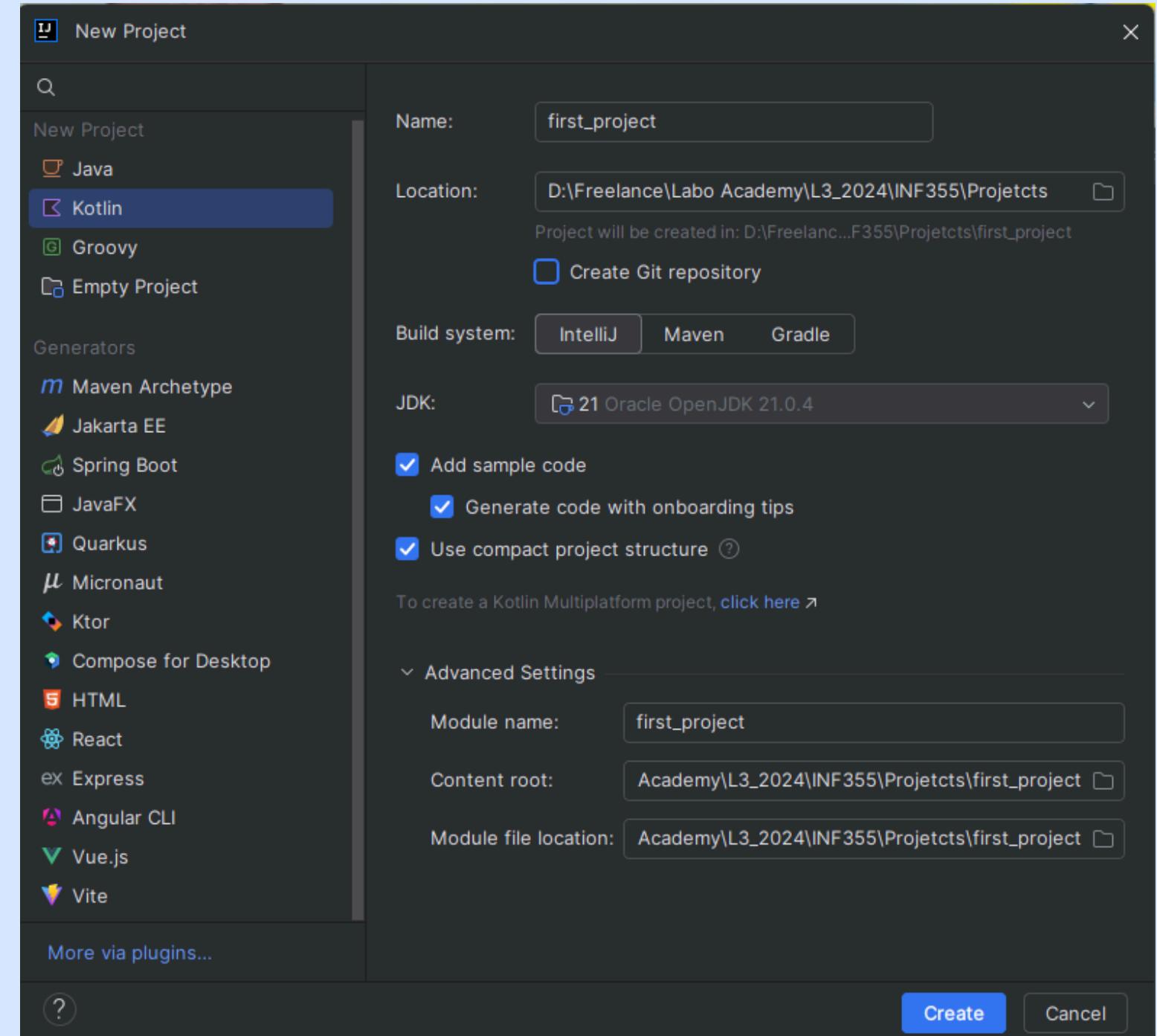
Welcome to IntelliJ IDEA

BACK

NEXT

# Premier projet Kotlin

## New Project



The screenshot shows the 'New Project' dialog in IntelliJ IDEA. The 'Kotlin' option is selected in the left sidebar under 'New Project'. The project name is set to 'first\_project' and the location is 'D:\Freelance\Labo Academy\L3\_2024\INF355\Projetccts'. The 'Create Git repository' checkbox is unchecked. Under 'Build system', 'IntelliJ' is selected. The 'JDK' dropdown shows '21 Oracle OpenJDK 21.0.4'. In the 'Generators' section, 'Add sample code' is checked. Advanced settings show the module name as 'first\_project' and content root as 'Academy\L3\_2024\INF355\Projetccts\first\_project'. The 'Create' button is at the bottom right.

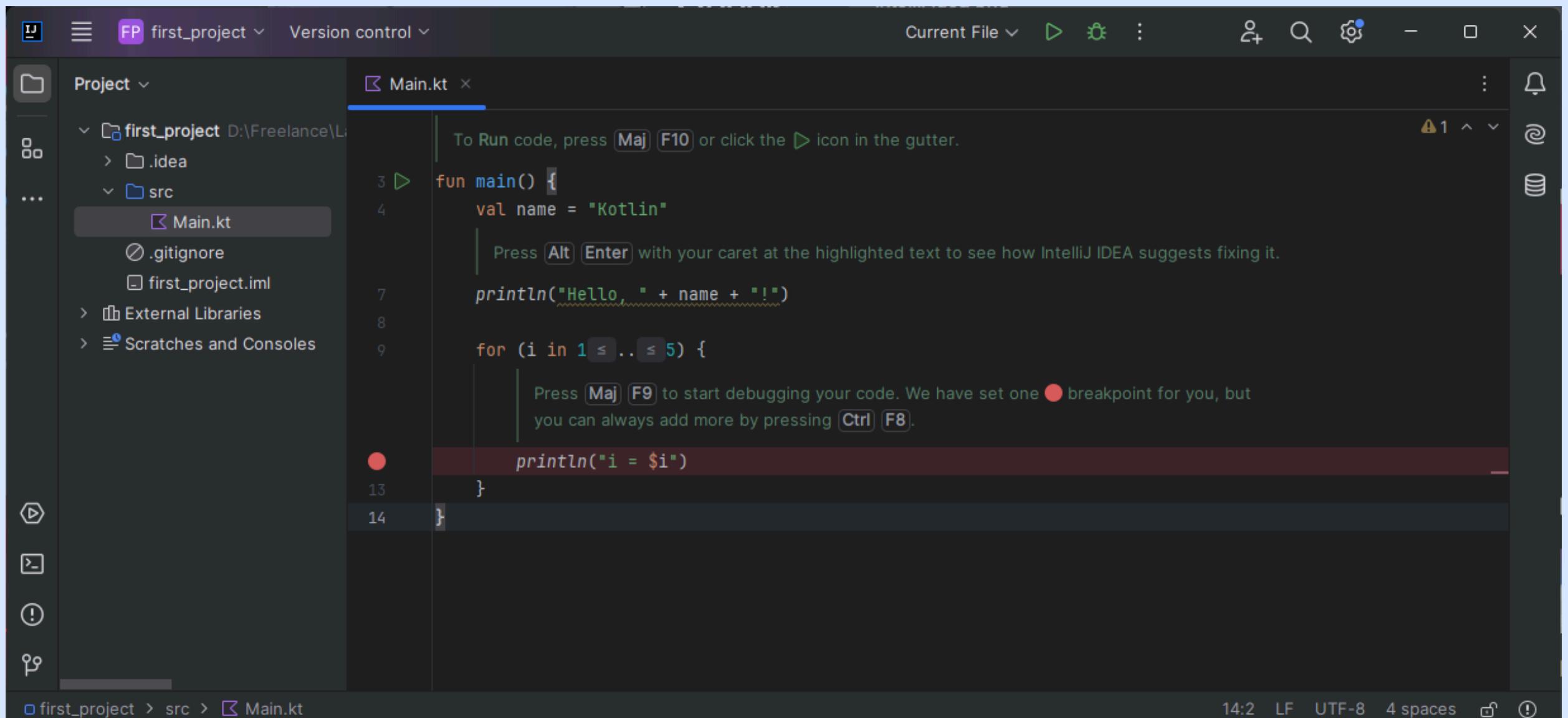
BACK

NEXT



# Premier projet Kotlin

## “Hello Kotlin”



The screenshot shows the IntelliJ IDEA interface with a project named "first\_project". The "Main.kt" file is open in the editor. The code contains a simple "Hello World" application:

```
fun main() {
    val name = "Kotlin"
    println("Hello, " + name + "!")
    for (i in 1 .. 5) {
        println("i = $i")
    }
}
```

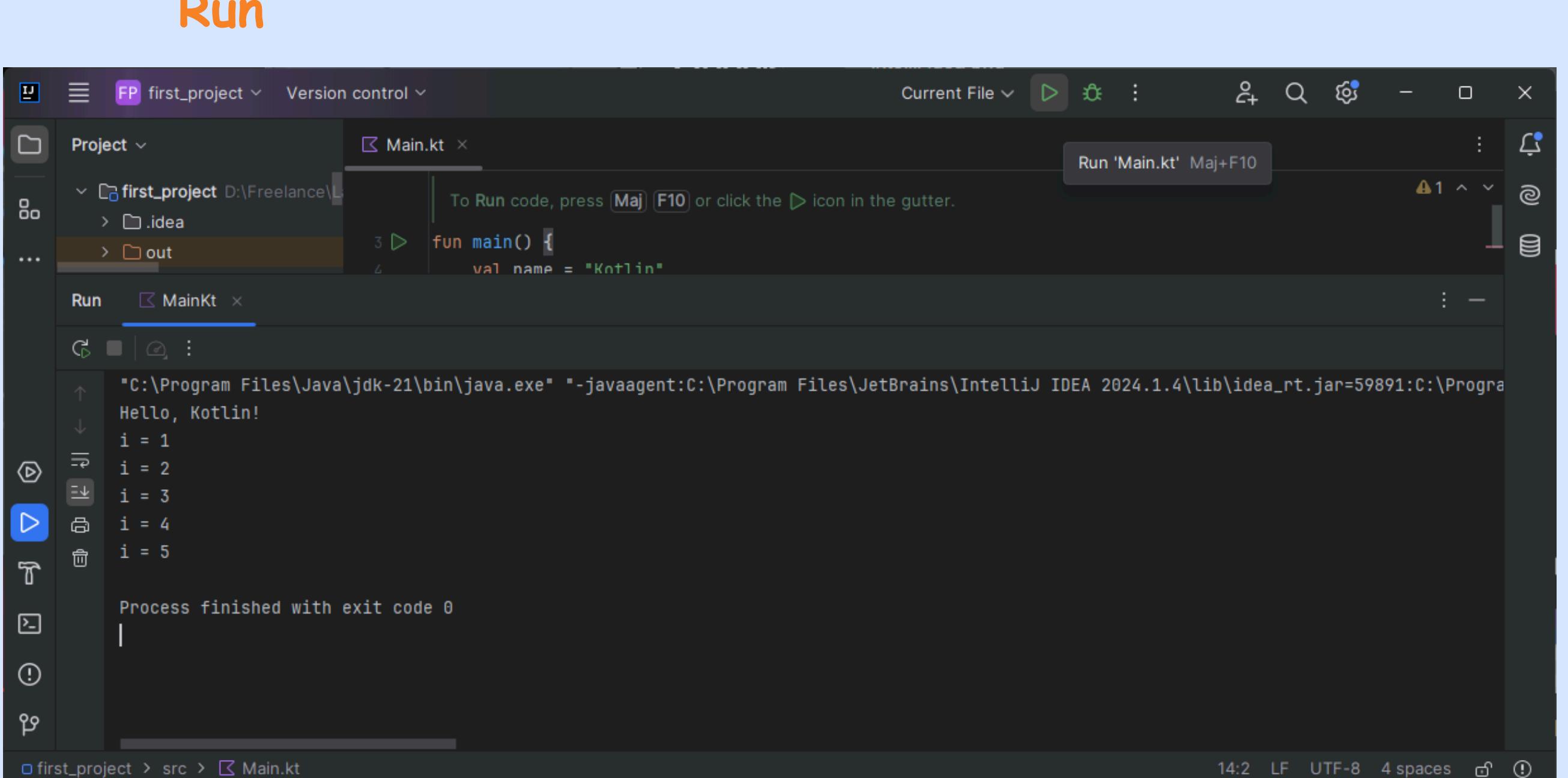
The code editor includes various annotations and hints from the IDE, such as run instructions and debugging tips.

BACK

NEXT

# Premier projet Kotlin

## Run



To Run code, press Maj+F10 or click the ▶ icon in the gutter.

```
fun main() {
    val name = "Kotlin"
}

Hello, Kotlin!
i = 1
i = 2
i = 3
i = 4
i = 5

Process finished with exit code 0
```

14:2 LF UTF-8 4 spaces

BACK NEXT



# Syntaxe et Types de Données

## Les commentaires

1. Pourquoi commenter son code

2. Par ligne

```
// Ceci est un commentaire  
// println("Hello Kotlin")
```

3. Par Bloc

```
/*  
 * Ceci est un commentaire  
 * val name = "Kotlin"  
 */
```



BACK

NEXT



# Syntaxe et Types de Données

## Déclaration des Variables

### 1. Syntaxe générale :

`var nomVariable: Type = valeur`

**Exemple:**

```
val name: String = "Kotlin"  
var age: Int = 18
```

### 2. Variables et Constantes : `val` vs `var`

- **val (valeur)** : immuable, c'est-à-dire que vous ne pouvez pas modifier la valeur une fois qu'elle a été initialisée.
- **var (variable)** : mutable, la valeur peut être modifiée après l'initialisation

```
name = "Java"  
age = 19
```

BACK

NEXT



# Syntaxe et Types de Données

## Les types de donné

1. Numbers: Byte, Short, Int, Long, Float, Double
2. Boolean: true, false
3. Characters
4. Arrays
5. Strings

[BACK](#)[NEXT](#)



# Syntaxe et Types de Données

## Les types de donné

| Type   | Taille | Plage  |
|--------|--------|--|
| Byte   | 8 bit  | -128 à 127   |
| Short  | 16 bit | -32 768 à 32 767                                       |
| Int    | 32 bit | -2 147 483 648 à 2 147 483 647                         |
| Long   | 64 bit | -9 223 372 036 854 775 808 à 9 223 372 036 854 775 807 |
| Float  | 32 bit | Nombre à virgule flottante simple précision            |
| Double | 64 bit | Nombre à virgule flottante double précision            |

[BACK](#)
[NEXT](#)



# Syntaxe et Types de Données

## Les types de donné

```
val byte: Byte = 1          // 1 0c
val short: Short = 2        // 2 0c
val int: Int = 3            // 4 0c
val long: Long = 4L         // 8 0c

val float: Float = 3.0f     // 4 0c
val double: Double = 5.0    // 8 0c

val char: Char = 'c'
val string: String = "Kotlin"

val bool: Boolean = true
```

Les types sont colorés en gris parce que Kotlin peut les déterminés

```
val float2 = 3.0f
val double2 = 100.0
val int2 = 3

println("Le type de float2 est : " + float2::class)
// Le type de float2 est : float (Kotlin reflection is not available)
println("Le type de double2 est : " + double2::class.simpleName)
// Le type de double2 est : Double
```

```
println("La valeur max d'un Int est : " + Int.MAX_VALUE)
// La valeur max d'un Int est : 2147483647
println("La valeur min d'un Int est : " + Int.MIN_VALUE)
// La valeur min d'un Int est : -2147483648
```

```
val array: Array<String> = arrayOf("a", "b", "c", "d", "e", "f", "g", "h")
```

BACK

NEXT



# Syntaxe et Types de Données

## La conversion de type

~~val num1: Int = 101~~

~~val num2: Long = num1~~

**val num2: Long = num1.toLong()**

```
val int3: Int = double.toInt()
val long3: Long = int3
val double3: Double = float
```

Type mismatch.  
Required: Double  
Found: Float

```
val long4: Long = 10000000000000000L
val int4: Int = long4.toInt()
println("int4 = " + int4) // Concatenation
println("int4 = $int4") // Template
// int4 = 1569325056
```

```
val int5 = 100
val char2 = int5.toChar()
println("char2 = $char2")
// char2 = d
```

BACK

NEXT



# Syntaxe et Types de Données

## Les opérateurs arithmétiques

- + Opérateur d'addition
- Opérateur de Soustraction
- \* Opérateur de Multiplication
- / Opérateur de Division
- % Opérateur Modulo

[BACK](#)[NEXT](#)



# Syntaxe et Types de Données

## Les opérateurs relationnels

- > Supérieur Strictement à
- < Inférieur strictement à
- >= Supérieur ou égal à
- <= Inférieur ou égal à
- == Egal
- != Différent

BACK

NEXT



# Syntaxe et Types de Données

## Les Opérateurs d'affectation

| Opérateur | Exemple   | Forme développée |
|-----------|-----------|------------------|
| =         | $x = 10$  | $x = 10$         |
| $+=$      | $x += 10$ | $x = x + 10$     |
| $-=$      | $x -= 10$ | $x = x - 10$     |
| $*=$      | $x *= 10$ | $x = x * 10$     |
| $/=$      | $x /= 10$ | $x = x / 10$     |
| $%=$      | $x %= 10$ | $x = x \% 10$    |

[BACK](#)
[NEXT](#)



# Syntaxe et Types de Données

## Les Opérateurs logiques

| Opérateur               | Nom         | Forme développée            |
|-------------------------|-------------|-----------------------------|
| <code>&amp;&amp;</code> | Et logique  | <code>a &amp;&amp; b</code> |
| <code>  </code>         | Ou Logique  | <code>a    b</code>         |
| <code>!</code>          | Nom Logique | <code>!a</code>             |

[BACK](#)[NEXT](#)



# Syntaxe et Types de Données

## Les entrées / sorties

```
println("Quel est votre nom ?")
```

```
val nom = readLine()
```

```
println("Vous avez écrit: $nom")
```

## 2e Méthode

```
val console = Scanner(System.`in`) // import java.util.*
```

```
println("Quel est votre âge?")
```

```
val age = console.nextInt()
```

BACK

NEXT



# Syntaxe et Types de Données

## Manipuler les chaînes de caractères

- La concaténation
- La taille d'une chaîne de caractères
- Comparer deux chaîne de caractères
- Accès à un caractère dans une chaîne à un index spécifique
- Substring (sous chaîne de caractères)

[BACK](#)[NEXT](#)



# Syntaxe et Types de Données

## Manipuler les chaînes de caractères

### 1- Concaténation et la taille d'une chaîne de caractères

```
// Concaténation
val str = "Test" + " 1"
val str2 = "Je fais : $str"
val str3 = "Taille de str2: ${str.length}"

val phrase = """Bonjour,
| Je suis Wilfried Okono.
| Dev Web et Mobile (Laravel, Vuejs, Reactjs et Flutter)
""".trimMargin()
```

BACK

NEXT



# Syntaxe et Types de Données

## Manipuler les chaînes de caractères

### 2- Comparer deux chaîne de caractères

```
val str4 = "Douala"  
val str5 = "Duala"  
println("str4 == str5 ? ${str4.equals(str5)}")  
  
println("str4 == str5 ? ${str4 == str5}")
```

### 3- Accès à un caractère dans une chaîne à un index spécifique

```
val str4 = "Douala"  
val str5 = "Duala"  
  
println("str4[0] = ${str4.get(0)}")  
  
println("str4[0] = ${str4[0]}")
```

BACK

NEXT



# Syntaxe et Types de Données

## Manipuler les chaînes de caractères

### 4- Substring (sous chaîne de caractères)

```
// Substring
val nomComplet = "Wilfried Okono"

println("Prénom = ${nomComplet.subSequence(0,8)}")
// Prénom = Wilfried
println("okono exist ? ${nomComplet.contains( other: "okono")}")
//okono exist ? false
println("okono exist ? ${nomComplet.contains( other: "okono", ignoreCase: true)})"
//okono exist ? true
```

BACK

NEXT



# Syntaxe et Types de Données

## Structures de contrôle (if, when, for, while, do while)

### 1- if et else

```
if (int == int2){  
    println("Ok")  
}else{  
    println("Not Ok")  
}  
  
val rest = if (int == int3){  
    "Equals"  
} else{  
    "Not Equals"  
}
```

BACK

NEXT



# Syntaxe et Types de Données

## 2- when

```
when(int){  
    0, 1, 2, 3, 4 -> println("is 4")  
    5 -> println("is 5")  
    6 -> println("is 6")  
    7 -> println("is 7")  
    else -> println("orther value")  
}  
  
val reslt2 = when(int2){  
    in 1 .. 4 -> 4.0  
    5 -> 5.0  
    6 -> 6.0  
    7 -> 7.0  
    else -> 8.0  
}
```

BACK

NEXT



# Syntaxe et Types de Données

## 3- for, while et do while

```
for (i in 1 .. 10 step 2){  
    println(i)  
}  
  
for (index in array.indices){  
    println("$index : $index")  
}  
  
for ((index, value) in array.withIndex()){  
    println("$index : $value")  
}  
  
for ((index, value) in (1 .. 10).withIndex()){  
    println("$index : $value")  
}  
  
while (int != 0){  
    int %= 2  
}  
  
do {  
    int %= 2  
}while (int != 0)
```

BACK

NEXT



# Syntaxe et Types de Données

## Les Fonctions

```
fun salutation(): Unit { ↗ OkonoWil
    println("Bonjour")
}

fun salutation2(name: String, age: Int = 100): String { ↗ OkonoWil *
    return if (age > 20){
        "Bonjour grand $name"
    } else {
        "Bonjour petit $name"
    }
}
```

## Les Fonctions Lambda

**Syntaxe :** {variable : Type -> corps de la fonction}

```
val somme = {num1: Int, num2: Int -> num1 + num2}
println("10 + 5 = ${somme(10, 5)}")
```

BACK

NEXT



# Syntaxe et Types de Données

## Déclaration et initialisation d'un tableau

Syntaxe:

**val tab = arrayOf<Type>(val1, val2, val3...)**

```
val ints = arrayOf(1, 2, 3, 4, 5)
val ints2 = intArrayOf(1, 2, 3, 4, 5)
val ints3 = arrayOf<Int>(1, 2, 3, 4, 5)
val ints4 = Array<Int>(size: 15) {elt -> elt + 1}

ints4.forEach { elt -> print(" $elt") }
// 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
println(ints.get(3))
println(ints[3])
// 4
ints.set(3, 15)
ints[3] = 15
println(ints[3])
// 15
```

BACK

NEXT



# Syntaxe et Types de Données

## Manipulation des tableaux

- La Taille d'un tableau:

```
var numbers = arrayOf<String>("un", "deux", "deux", "trois", "quatre")
numbers.size // 5
numbers.count() // 5
```

- Vérifier l'existence d'un élément dans un tableau

```
numbers.contains("cinq") // false
```

- Connaître le premier et le dernier élément d'un tableau

```
numbers.first // un
numbers.last // quatre
```

- Valeurs distinctes d'un tableau

```
val result = numbers.distinct() // un, deux, trois, quatre
```

BACK

NEXT



# Syntaxe et Types de Données

## Manipulation des tableaux

- Suppression d'éléments d'un tableau

```
val result = numbers.drop(2) // deux trois quatre  
val result = numbers.dropLast(2) // un deux deux
```

- Vérification si un tableau est vide

```
numbers.isNotEmpty() // true  
numbers.isEmpty() // false
```

[BACK](#)[NEXT](#)



# Syntaxe et Types de Données

## Null safety en Kotlin

- Le Null Safety en Kotlin est une procédure visant à éliminer le risque de référence nulle du code
- Le compilateur Kotlin lève NullPointerException immédiatement s'il trouve qu'un argument nul est passé.

### Nullable Types

```
// déclarer une variable comme nullable  
var text: String? = "Hello"  
text = null // OK  
println(text)
```

#### Output

null

BACK

### Not Nullable Types

```
// déclarer une variable comme non nullable  
var text: String = "Hello"  
text = null // Error  
println(text)
```

#### Output

Kotlin: Null can not be a value of a not-null String

NEXT



# Syntaxe et Types de Données

## Conversion de type sécurisé et risqué

### UnSafe Cast

```
val any: Any? = null  
val text: String = any as String  
println(text)
```

### Output

Exception in thread "main"  
Kotlin.TypeCastException: null cannot  
be cast to not null type Koylin.String

### Null check ?.

### Opérateur Elvis ?:

### Safe Cast

```
val any: Any? = null  
val text: String = any as? String  
println(text)
```

### Output

null

BACK

NEXT



# La programmation orientée objet

## Les constructeurs

- Une classe Kotlin peut avoir deux type de constructeurs:  
**Constructeur principal et Constructeurs secondaires**
- Une classe Kotlin peut avoir un constructeur principal et un ou plusieurs constructeurs secondaires supplémentaires.
- Le constructeur principal Kotlin initialise la classe, tandis que le constructeur secondaire aide à inclure une logique supplémentaire lors de l'initialisation de la classe.

[BACK](#)[NEXT](#)



# La programmation orientée objet

## Le constructeur principal

```
class Person2 constructor(name: String, age: Int) {  
    var _name: String = name  
    var _age: Int  
    init {; new *  
        _age = age + 1  
    }  
}
```

```
class Person3(var nom: String, var age: Int) { new *
```

```
class Person3(private var nom: String, protected var age: Int) {
```

[BACK](#)[NEXT](#)



# La programmation orientée objet

## Les constructeurs secondaires

```
class Person(var name : String, protected var age: Int) { lateinit OkonoWil *  
    var email: String = ""  
    constructor(name: String, age: Int, email: String, ) : this(name, age){ new *  
        this.email = email  
    }  
    constructor(name: String) : this(name, age: 0) lateinit OkonoWil  
    constructor(): this( name: "Toto") lateinit OkonoWil  
  
    fun salution() { lateinit OkonoWil  
        println("$name vous dit bonjour")  
    }  
}
```

BACK

NEXT



# La programmation orientée objet

## Les Getters et Setters

- **Setter** est utilisé pour définir la valeur d'un attribut
- **Getter** est utilisé pour obtenir la valeur

Les getters et les setters sont générés automatiquement dans le code en Kotlin

```
var email: String = "" new *  
|  
get() = field  
set(value) {field = "Email : $value" }
```

BACK

NEXT



# La programmation orientée objet

## Les modificateurs de visibilité

Les modificateurs de visibilité Kotlin sont les mots clés qui définissent la visibilité des classes, des objets, de l'interface, des constructeurs, des fonctions ainsi que des propriétés et de leurs setters. **public, private, protected, internal**

Syntaxe :

modificateur val/var variable = valeur

Exemple:

**public val** text = "Hello"

BACK

NEXT



# Syntaxe et Types de Données

## Les modificateurs de visibilité

| Modificateur | Description   |
|--------------|---|
| public       | visible partout   |
| private      | visible à l'intérieur de la classe uniquement             |
| protected    | visible à l'intérieur de la classe et de ses sous-classes |
| internal     | visible à l'intérieur du module                           |

[BACK](#)[NEXT](#)



# Syntaxe et Types de Données

## Heritage

L'héritage est un concept fondamental en programmation orientée objet, y compris en Kotlin. Il permet à une classe de dériver d'une autre classe, d'hériter de ses propriétés et méthodes, et de les réutiliser ou de les modifier. Cela favorise la réutilisation du code et permet d'étendre les fonctionnalités d'une classe existante.

### 1. Déclaration d'une classe parent et d'une classe enfant

En Kotlin, toutes les classes sont finales par défaut, c'est-à-dire qu'elles ne peuvent pas être héritées. Si vous voulez qu'une classe puisse être héritée, vous devez la déclarer avec le mot-clé `open`.

### 2. Le mot-clé `open` et `override`

- `open` : Utilisé pour indiquer qu'une classe ou une méthode peut être héritée ou redéfinie. En Kotlin, les classes et méthodes sont finales par défaut, donc vous devez explicitement les rendre héritables.
- `override` : Utilisé pour redéfinir une méthode ou une propriété d'une classe parent dans une classe enfant.

[BACK](#)[NEXT](#)



# La programmation orientée objet

## Exemple

```
// Classe Parent
open class Animal { new*
    open fun sound() { new*
        println("L'animal fait un bruit.")
    }
}

// Classe Enfant qui hérite d'Animal
class Dog : Animal() { new*
    override fun sound() { new*
        println("Le chien aboie.")
    }
}

fun main() { new*
    val myDog = Dog()
    myDog.sound() // Affiche: Le chien aboie.
}
```

- La classe **Animal** est une classe parent, et elle est marquée avec le mot-clé **open** pour permettre l'héritage.
- La méthode **sound()** est également marquée avec **open**, ce qui signifie qu'elle peut être redéfinie par les classes dérivées (comme **Dog**).
- La classe **Dog** hérite de **Animal** et **redéfinit** (**override**) la méthode **sound()**.

[BACK](#)
[NEXT](#)



# La programmation orientée objet

## Constructeur dans l'héritage

Lorsqu'une classe enfant hérite d'une classe parent, elle doit appeler le constructeur de la classe parent. Cela se fait avec les paramètres du constructeur de la classe parent lors de la déclaration de la classe enfant.

```
// Classe Parent avec un constructeur
open class Animal2(val name: String) { new *
    open fun sound() { new *
        println("$name fait un bruit.")
    }
}

// Classe Enfant avec un constructeur
class Dog2(name: String) : Animal2(name) { new *
    override fun sound() { new *
        println("$name aboie.")
    }
}
```

[BACK](#)[NEXT](#)



# La programmation orientée objet

## Heritage

```
open class Person3(protected var nom: String, protected var age: Int) { new *
    fun eat(){ new *
        println("$nom $age - eat")
    }
    open fun walk(){ new *
        println("$nom $age - walk")
    }
}

class Student(nom: String, age: Int) : Person3(nom, age) { new *
    private fun learn(){ new *
        println("Learn")
    }
}

class Teacher(nom: String, age: Int, private var salary: Double) : Person3(nom, age) { new *
    private fun teach(){ new *
        println("Teacher")
    }
    override fun walk() { new *
        println("$nom $age - walk - Teacher")
    }
}

fun main(args: Array<String>) { new *
    val person = Teacher( nom: "Adam", age: 18, salary: 20000.0 )
    person.walk()
}
```

BACK

NEXT



# La programmation orientée objet

## Classes abstraites et héritage

Les classes abstraites jouent un rôle important dans l'héritage en Kotlin. Elles définissent des fonctionnalités communes aux classes dérivées, mais ne peuvent pas être instanciées directement. Elles peuvent contenir des méthodes abstraites (sans implémentation) qui doivent être redéfinies dans les classes dérivées.

## Appel des méthodes de la classe parent

Il est possible d'appeler les méthodes ou les constructeurs de la classe parent à partir de la classe enfant en utilisant le mot-clé super.

## Héritage multiple avec les interfaces

Kotlin ne supporte pas l'héritage multiple de classes, mais permet d'implémenter plusieurs interfaces. Les interfaces peuvent contenir des définitions de méthodes abstraites (sans implémentation) ainsi que des méthodes avec une implémentation par défaut. Une classe peut hériter d'une autre classe tout en implementant plusieurs interfaces.

[BACK](#)[NEXT](#)



# La programmation orientée objet

## Résumé de l'héritage en Kotlin

- **open** : Les classes et méthodes doivent être marquées comme open pour pouvoir être héritées ou redéfinies.
- **override** : Utilisé pour redéfinir une méthode ou une propriété d'une classe parent dans une classe enfant.
- **Constructeur** : Les classes enfants doivent appeler le constructeur de la classe parent lors de l'héritage.
- **super** : Utilisé pour appeler les méthodes de la classe parent dans une classe dérivée.
- **Abstraction** : Les classes abstraites et les interfaces permettent de définir des comportements communs, mais seuls les comportements définis (non abstraits) peuvent être directement utilisés.

[BACK](#)[NEXT](#)



# La programmation orientée objet

## Les classes abstraites

- une classe abstraite ne peut pas être instanciée, ce qui signifie que nous ne pouvons pas créer l'objet d'une classe abstraite
- Contrairement à d'autres classes, une classe abstraite est toujours ouverte, nous n'avons pas besoin d'utiliser le mot-clé open
- Une classe qui possède une méthode abstraite doit obligatoirement être abstraite
- Une classe qui hérite d'une classe abstraite doit redéfinir ses fonctions abstraites
- Contrairement aux interfaces, une classe abstraite peut avoir des constructeurs, stocker un état (des propriétés concrètes), et contenir des méthodes complètes.

[BACK](#)[NEXT](#)



# La programmation orientée objet

## Les classes abstraites

```
class Animal(name: String, age: Int) : EtreVivant(name, age) { new *

    override fun eat() { new*
        println("Animal is eating...")
    }

    override fun run() { new*
        println("Animal is runing...")
    }
}

abstract class EtreVivant(var name: String, var age: Int) { new *

    constructor(name: String) : this(name, age: 0) {} new *

    abstract fun eat(); new *

    abstract fun run(); new *

    fun sayHello(){ new*
        println("Hello, is $name")
    }
}

fun main(args: Array<String>) { new*
    var animal = Animal( name: "Jack", age: 25)
    animal.eat()
    animal.sayHello()
}
```

BACK

NEXT



# La programmation orientée objet

## Points clés sur les classes abstraites en Kotlin :

- Héritage simple : Une classe ne peut hériter que d'une seule classe abstraite (comme en Java).
- Constructeurs : Une classe abstraite peut avoir un constructeur et des propriétés concrètes.
- Méthodes abstraites et concrètes : Une classe abstraite peut contenir des méthodes abstraites (à implémenter) et des méthodes concrètes (avec du code).

[BACK](#)[NEXT](#)



# La programmation orientée objet

## Les interfaces

- Une interface peut avoir à la fois une fonction abstraite et non abstraite
- Une classe peut implémenter plusieurs interfaces.
- Toutes les propriétés abstraites et les fonctions membres abstraites d'une interface doivent être redéfinies dans les classes qui l'implémentent.
- Une interface en Kotlin est une sorte de contrat qui définit un ensemble de méthodes et de propriétés qu'une classe doit implémenter. Contrairement à une classe abstraite, une interface ne peut pas stocker d'état (c'est-à-dire, pas de propriétés avec des valeurs concrètes), mais elle peut avoir des implementations par défaut de certaines méthodes.

[BACK](#)[NEXT](#)



# La programmation orientée objet

## Déclaration d'une interface

```
interface Clickable { new *
    fun click() new *
    fun show() { new *
        println("I'm clickable!")
    }
}
```

- **click()** : Méthode abstraite, elle doit être implémentée par toute classe qui hérite de l'interface.
- **show()** : Méthode avec une implémentation par défaut, elle peut être utilisée directement ou redéfinie par la classe qui implémente l'interface.

## Implémentation d'une interface :

```
class Button : Clickable { new *
    override fun click() { new *
        println("Button clicked!")
    }
}

fun main() { new *
    val button = Button()
    button.click() // Affiche: Button clicked!
    button.show() // Affiche: I'm clickable!
}
```

BACK

NEXT



# La programmation orientée objet

## Points clés sur les interfaces en Kotlin :

- **Multiples interfaces** : Une classe peut implémenter plusieurs interfaces.
- **Méthodes avec implémentation par défaut** : Les interfaces peuvent définir des méthodes avec du code par défaut
- **Propriétés** : Une interface peut également déclarer des propriétés, mais elles ne peuvent pas contenir de valeurs concrètes. Seul un getter ou setter peut être défini par défaut.

```
interface Shape { new *
    val area: Double
    fun draw() new *
}

class Circle(val radius: Double) : Shape { new *
    override val area: Double new *
        get() = Math.PI * radius * radius

    override fun draw() { new *
        println("Drawing a circle")
    }
}
```

BACK

NEXT



# La programmation orientée objet

## Comparaison entre Interfaces et Classes Abstraites

| Caractéristique           | Interface               | Classe Abstraite      |
|---------------------------|-------------------------|-----------------------|
| Instanciabilité           | Non                     | Non                   |
| Multiples implémentations | Oui                     | Non (héritage simple) |
| Méthodes abstraites       | Oui                     | Oui                   |
| Méthodes concrètes        | Oui (depuis Kotlin 1.2) | Oui                   |
| Propriétés concrètes      | Non                     | Oui                   |
| Constructeurs             | Non                     | Oui                   |

[BACK](#)
[NEXT](#)



# La programmation orientée objet

## Les classes imbriquée (nested) et intérieure (inner)

En Kotlin, les classes imbriquées (nested classes) et les classes intérieures (inner classes) permettent de définir des classes à l'intérieur d'autres classes. Elles servent à structurer le code, à regrouper des éléments logiquement liés, et à faciliter la manipulation de certaines données.

### 1. Classes Imbriquées (Nested Classes)

Une classe imbriquée en Kotlin est une classe déclarée à l'intérieur d'une autre classe. Par défaut, une classe imbriquée n'a pas accès aux membres de la classe qui la contient. Elle se comporte comme une classe totalement indépendante.

```
class ClassExterieure { new *
    var name: String = "Toto"

    class ClassImbriquee{ new *
        var age: Int = 15

        fun afficher(){ new *
            println("$name $age")
        }
    }
}

fun main(args: Array<String>) { new *
    val classImbriquee = ClassExterieure.ClassImbriquee()
}
```

BACK

NEXT



# La programmation orientée objet

## Les classes imbriquée (nested) et intérieure (inner)

### Points importants sur les classes imbriquées

- Par défaut, une classe imbriquée n'a pas accès aux membres de la classe externe (comme `name` dans l'exemple ci-dessus).
- Elle peut être utilisée comme une classe statique en Java.

### 2. Classes Intérieures (Inner Classes)

Une classe intérieure est une classe imbriquée qui a accès aux membres de la classe externe. Pour créer une classe intérieure, on utilise le mot-clé `inner`. Cela permet d'accéder aux propriétés et méthodes de la classe externe via une instance de la classe extérieure.

```
class ClassExterieure2 { new *
    var name: String = "Toto"

    inner class ClassInterieure{ new *
        var age: Int = 15

        fun afficher(){ new *
            println("$name $age")
        }
    }
}

fun main(args: Array<String>) { new *
    val classInterieure = ClassExterieure2().ClassInterieure()
}
```

BACK

NEXT



# La programmation orientée objet

## Quand utiliser une classe imbriquée ou une classe intérieure ?

- **Classe imbriquée (Nested Class)** : Utilisez-la si vous avez une classe qui est logiquement liée à une autre mais qui n'a pas besoin d'accéder aux membres de la classe externe. Par exemple, pour des utilitaires ou des composants statiques.
- **Classe intérieure (Inner Class)** : Utilisez-la lorsque vous avez besoin d'une classe qui doit accéder à l'état ou au comportement de la classe externe. Cela peut être utile pour les implémentations fortement couplées, où la classe externe et interne doivent collaborer.

BACK

NEXT



# La programmation orientée objet

## Les classes de données (Data Class)

En Kotlin, **une data class** (ou classe de données) est une classe spécialement conçue pour contenir des données. Elle fournit automatiquement des fonctionnalités pratiques comme l'égalité, la copie, et une représentation lisible sous forme de chaîne, tout en réduisant le besoin d'écrire beaucoup de code standard. Les data classes sont très utiles lorsqu'on travaille avec des modèles de données ou des structures qui encapsulent des valeurs.

### 1. Déclaration d'une Data Class

Pour déclarer une data class en Kotlin, on utilise le mot-clé **data** avant le mot-clé **class**. La classe doit avoir au moins un paramètre dans le constructeur primaire.

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

Dans cet exemple, **User** est une classe qui stocke des informations sur un utilisateur, notamment son nom et son âge.

BACK

NEXT



# La programmation orientée objet

## 2. Fonctionnalités automatiques d'une Data Class

Lorsqu'une classe est déclarée comme une data class, Kotlin génère automatiquement plusieurs méthodes utiles :

1. `equals()` : Pour comparer les objets basés sur leurs propriétés.
2. `hashCode()` : Génère un code de hachage cohérent avec `equals()`.
3. `toString()` : Renvoie une représentation lisible sous forme de chaîne des données de l'objet.
4. `copy()` : Crée une copie de l'objet avec la possibilité de modifier certaines de ses propriétés.
5. `componentN()` : Génère des fonctions déstructurantes pour accéder aux propriétés individuelles de l'objet.

## 4. Conditions pour une Data Class

- Avoir un constructeur primaire avec au moins un paramètre.
- Tous les paramètres du constructeur primaire doivent être marqués comme `val` ou `var`, sinon ils ne seront pas inclus dans les fonctionnalités automatiques comme `equals()`, `hashCode()`, etc.
- Une data class ne peut pas être abstraite, ouverte, scellée (`sealed`), ou interne (`inner`).

BACK

NEXT



# La programmation orientée objet

## Les classes de données (Data Class)

```
data class Personne(var name: String, var age: Int) { new *  
}  
fun main(args: Array<String>) { new *  
    val person1 = Personne( name: "John", age: 18)  
    val person2 = Personne( name: "John", age: 18)  
  
    println("person1 == person2 = ${person1 == person2}")  
    // person1 == person2 = true  
  
    val person3 = person1.copy()  
    println("person3 : $person3")  
    // person3 : Personne(name=John, age=18)  
  
    val person4 = person1.copy(name = "Toto")  
    println("person4 : $person4")  
    // person4 : Personne(name=Toto, age=18)  
  
    println("hashCode : preson1 => ${person1.hashCode()} , preson2 => ${person2.hashCode()}")  
    // hashCode : preson1 => 71750727 , preson2 => 71750727  
  
    println("person1.name = ${person1.component1()}")  
    // person1.name = John  
    println("person1.age = ${person1.component2()}")  
    //person1.age = 18  
}
```

BACK

NEXT



# La programmation orientée objet

## La classe Objet

- En Kotlin, la classe object permet de définir un singleton, c'est-à-dire une classe dont on ne peut créer qu'une seule instance. Cette instance est créée dès que la classe est appelée et elle est accessible partout où cette classe est visible.

```
object DatabaseConfig { new *
    val url = "jdbc:mysql://localhost:3306/mydb"
    val user = "admin"
    val password = "password"

    fun connect() { new *
        println("Connecting to the database...")
    }
}

fun main() { new *
    println(DatabaseConfig.url)
    DatabaseConfig.connect()
}
```

[BACK](#)[NEXT](#)



# La programmation orientée objet

## Companion Object

- Les **companion objects** en Kotlin sont une manière de définir des membres statiques associés à une classe. Contrairement à Java, où l'on déclare des champs et des méthodes comme **static**, Kotlin ne supporte pas directement les membres statiques dans une classe. À la place, on utilise un companion object pour encapsuler ce comportement.
- Un **companion object** est un objet unique qui est partagé entre toutes les instances d'une classe. Il peut contenir des fonctions et des propriétés qui peuvent être accédées sans instancier la classe.

## Pourquoi utiliser un Companion Object ?

- Fonctionnalité semblable aux membres statiques : Il permet de regrouper des méthodes et des propriétés qui n'ont pas besoin d'être liées à une instance particulière de la classe.
- Interaction avec d'autres fonctions : Un companion object peut être utilisé pour fournir des méthodes d'usine (factory methods), des configurations par défaut, ou des valeurs constantes partagées entre toutes les instances de la classe.

BACK

NEXT



# La programmation orientée objet

## Déclaration d'un Companion Object

Un companion object est défini à l'intérieur d'une classe en utilisant le mot-clé `companion`. Il permet de créer des membres qui peuvent être accédés de manière similaire aux membres static en Java.

```
class CompanionObject { new *
    companion object { new *
        fun sayHello() { new *
            println("Hello from companion object!")
        }

        val CONSTANT = 42
    }
}

fun main() { new *
    CompanionObject.sayHello() // Affiche: Hello from companion object!
    println(CompanionObject.CONSTANT) // Affiche: 42
}
```

BACK

NEXT



# La programmation orientée objet

## Déclaration d'un Companion Object

Un companion object est défini à l'intérieur d'une classe en utilisant le mot-clé `companion`. Il permet de créer des membres qui peuvent être accédés de manière similaire aux membres static en Java.

```
class CompanionObject { new *
    companion object { new *
        fun sayHello() { new *
            println("Hello from companion object!")
        }

        val CONSTANT = 42
    }
}

fun main() { new *
    CompanionObject.sayHello() // Affiche: Hello from companion object!
    println(CompanionObject.CONSTANT) // Affiche: 42
}
```

BACK

NEXT



# La programmation orientée objet

## Factory Methods avec Companion Object

Un usage classique des companion objects est de créer des méthodes d'usine qui permettent d'instancier des objets de manière plus flexible.

```
class User(val name: String, val age: Int) { new *
    companion object { new *
        fun create(name: String): User { new *
            return User(name, age: 18) // Par défaut, on donne l'âge 18 à tous les nouveaux utilisateurs
        }
    }
}

fun main() { new *
    val user = User.create("Alice")
    println("Nom: ${user.name}, Âge: ${user.age}") // Affiche: Nom: Alice, Âge: 18
}
```

BACK

NEXT



# La programmation orientée objet

## Nommer un Companion Object

Même si ce n'est pas obligatoire, un companion object peut être nommé pour plus de clarté. Cela peut être utile lorsque vous avez plusieurs objets ou quand vous voulez rendre le code plus expressif.

```
class User(val name: String, val age: Int) { new *
    companion object Factory { new *
        fun create(name: String): User { new *
            return User(name, age: 18) // Par défaut, on donne l'âge 18 à tous les nouveaux utilisateurs
        }
    }
}

fun main() { new *
    val user = User.Factory.create("Alice") // Pas obligatoire d'utiliser le nom (Factory)
    println("Nom: ${user.name}, Âge: ${user.age}") // Affiche: Nom: Alice, Âge: 18
}
```

BACK

NEXT

# THANK YOU

by Labo Academy & Okono

