KOTLIN CHEAT SHEET

“Un résumé compact et efficace de Kotlin, couvrant les bases essentielles aux fonctionnalités avancées, conçu comme un aide-mémoire pour les développeurs.”

superdeveloppeur.com

1 Syntaxe de base

- **Variables (val, var) :**
  - **val** : pour une variable immuable (constante).

```
val languageName: String = "Kotlin"
```

  - **var** : pour une variable mutable.

```
var version: Double = 1.4
```

- **Types de données (Int, String, etc.) :**
  - Types primitifs : Int, Double, Float, Boolean, etc.
  - Chaînes de caractères : String

```
val year: Int = 2021
val name: String = "Kotlin Cheat Sheet"
```

- **Structures de contrôle (if, when, for, while) :**
  - **if** : Utilisé pour les conditions.

```
if (year > 2020) {
    println("Bienvenue dans le futur !")
}
```

  - **for** : Boucle sur des plages ou collections.

```
for (i in 1..5) {
    println(i)
}
```

  - **while / do-while** : Boucles classiques.

```
var i = 1
while (i <= 5)  {
    println(i)
    i++
}
```

```
var compteur = 1
do {
   println("Compteur est à : $compteur")
   compteur++
} while (compteur <= 5)
```

  - **when** : Remplace le switch-case de Java.

```
when (version) {
    1.4 -> println("Kotlin version 1.4")
    else -> println("Autre version")
}
```

2 Classes et Objets

- **Déclaration de Classes :**
  - Les classes en Kotlin sont déclarées en utilisant le mot-clé 'class'.

```
class Voiture {
    var marque: String = "Inconnue"
    fun démarrer() {
        println("La voiture démarre!")
    }
}
// Création d'une instance
val maVoiture = Voiture()
```

- **Héritage et Interfaces :**
  - Kotlin supporte l'héritage simple. Toutes les classes ont une super classe commune Any.

```
open class Vehicule {
 // 'open' est nécessaire pour hériter
   open fun demarrer() {
      println("Le véhicule démarre!")
   }
}

class Moto : Vehicule() {
 override fun demarrer() {
 println("La moto vroom vroom!")
 }
}
```

- **Propriétés et Méthodes :**
  - Les propriétés sont déclarées comme des variables. Les méthodes sont déclarées avec fun.

```
class Personne(val nom: String) {
    var age: Int = 0
    fun sePresenter() {
        println("_e m'appelle $nom et j'ai $age ans.")
    }
}
```

3 Fonctions

- **Définition et Appel :**
  - Les fonctions sont définies avec le mot-clé fun et ont un type de retour spécifié.

```
fun additionner(a: Int, b: Int): Int {
    return a + b
}
val resultat = additionner(5, 3)
```

- **Paramètres et Retour :**
  - Les fonctions peuvent prendre des paramètres et retourner des valeurs.

```
fun saluer(nom: String): String {
    return "Bonjour, $nom!"
}
println(saluer("Alice"))
```

- **Fonctions d'Extension :**
  - Kotlin permet d'étendre une classe avec de nouvelles fonctionnalités sans en hériter.

```
fun String.exclamer() {
    println(this + "!")
}
"Bonjour".exclamer() // Affiche: Bonjour!
```

4 Gestion des Exceptions :

- **Try, Catch, Finally :**
  - Kotlin gère les exceptions de manière similaire à Java.

```
try {
    val division = 10 / 0
} catch (e: ArithmeticException) {
    println("Division par zéro!")
} finally {
    println("Ceci est exécuté quoi qu'il arrive.")
}
```

- **Gestion Personnalisée des Erreurs :**
  - Vous pouvez créer vos propres exceptions en héritant de la classe 'Exception'.

```
class MaException(message: String) : Exception(message)

fun verifierAge(age: Int) {
    if (age < 18) {
        throw MaException("Accès refusé.")
    }
}
```

5 Collections

- **List :** Les listes en Kotlin peuvent être mutables (MutableList) ou immuables (List). Elles sont ordonnées et peuvent contenir des éléments dupliqués.

```
val immuableList = listOf("Kotlin", "Java", "C++")
val mutableList = mutableListOf("Kotlin", "Java", "C++")
mutableList.add("Python")
println(mutableList)
 // Affiche [Kotlin, Java, C++, Python]
```

Parcours et manipulation de listes avec des lambdas :

```
val nombres = listOf(1, 2, 3, 4, 5)
val doubles = nombres.map { it * 2 }
// Multiplie chaque élément par 2
val pairs = nombres.filter { it % 2 == 0 }
// Garde seulement les nombres pairs
println(doubles)  // Affiche [2, 4, 6, 8, 10]
println(pairs)  // Affiche [2, 4]
```

- **Map** : Les maps (Map pour les immuables et MutableMap pour les mutables) associent des clés uniques à des valeurs. Utiles pour la recherche rapide de valeurs basée sur des clés.

```
val immutableMap = mapOf(1 to "Kotlin", 2 to "Java")
val mutableMap = mutableMapOf("Kotlin" to 1, "Java" to 2)
mutableMap["Python"] = 3  // Ajoute "Python" avec la clé 3
println(mutableMap)  // Affiche {Kotlin=1, Java=2, Python=3}
```

Itération sur une map :

```
for ((key, value) in mutableMap) {
    println("Clé: $key, Valeur: $value")
}
// Affiche chaque paire clé-valeur
```

- **Set :** Les sets (Set pour les immuables et MutableSet pour les mutables) stockent des éléments uniques. Ils sont utiles pour éliminer les doublons et effectuer des opérations d'ensemble.

```
val immutableSet = setOf("Kotlin", "Java", "Kotlin")
println(immutableSet)
 // Affiche [Kotlin, Java] car les doublons sont éliminés
val mutableSet = mutableSetOf("Kotlin", "Java")
mutableSet.add("C++")
println(mutableSet)
// Affiche [Kotlin, Java, C++]
```

Utilisation d'opérations d'ensemble :

```
val setA = setOf(1, 2, 3)
val setB = setOf(2, 3, 4)
val union = setA.union(setB)
val intersection = setA.intersect(setB)
println(union)  // Affiche [1, 2, 3, 4]
println(intersection)  // Affiche [2, 3]
```

6 Lambdas et Fonctions de Haute Ordre

- **Lambdas :** Les lambdas sont des fonctions anonymes qui peuvent être utilisées comme expression.

```
val sum = { a: Int, b: Int -> a + b }
println(sum(5, 3))  // Affiche 8
```

- **Fonctions de Haute Ordre :** Des fonctions qui prennent des fonctions comme paramètres ou retournent des fonctions.

```
fun operate(a: Int, b: Int, op: (Int, Int) -> Int): Int {
    return op(a, b)
}
val result = operate(2, 3, sum)
println(result)  // Affiche 5
```

- **Fonctions Inline, Noinline, et Crossinline :**
  - **inline :** Pour éviter l'overhead de l'allocation mémoire lors de l'utilisation de fonctions de haute ordre.
  - **noinline :** Pour empêcher l'inlining d'une lambda particulière.
  - **crossinline :** Pour garantir qu'une lambda passée à une fonction inline n'est pas utilisée dans un contexte non local (comme une inner function).

```
inline fun inlineOperation(a: Int, b: Int,
crossinline op: (Int, Int) -> Int): () -> Int {
    return { op(a, b) }
}
val inlineResult = inlineOperation(4, 2, sum)()
println(inlineResult)  // Affiche 6
```

7 Génériques

- **Définition et Utilisation :** Kotlin permet de définir des fonctions et des classes avec des types génériques.

```
class Box<T>(t: T) {
    var value = t
}
val box: Box<Int> = Box(1)
```

- **Contraintes et Variance (in, out) :**
  - in pour un type générique contraint à être consommé, mais jamais produit.
  - out pour un type générique contraint à être produit, mais jamais consommé.

```
class Producer<out T>(private val value: T) {
    fun get(): T = value
}
class Consumer<in T> {
    fun accept(t: T) { /* ... */ }
}
```

8 Interopérabilité avec Java

- **Appel de Code Java depuis Kotlin :**

Kotlin est conçu pour être compatible avec Java, permettant l'utilisation de bibliothèques Java dans le code Kotlin.

```
val list = java.util.ArrayList<String>()
list.add("Kotlin")
println(list.get(0))  // Affiche "Kotlin"
```

- **Utilisation de Bibliothèques Java :**

Vous pouvez importer et utiliser des classes Java de manière transparente.

```
import java.util.Date

val now = Date()
println(now.toString())
 // Affiche la date et l'heure actuelles
```

9 Coroutines

- **Concepts de Base :** Les coroutines sont une façon de gérer les opérations asynchrones de manière plus efficace et plus lisible.

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch {
        delay(1000L)
        println("Coroutine!")
    }
    println("Hello,")
}
// Affiche "Hello," puis "Coroutine!"  après une seconde
```

- **Utilisation dans la Gestion de l'Asynchronisme :** Les coroutines permettent d'écrire du code asynchrone qui ressemble à du code synchrone standard.

```
suspend fun longRunningTask(): Int {
    delay(2000L) // Simule un travail long
    return 42
}

fun main() = runBlocking {
    val result = async { longRunningTask() }
    println("Le résultat est ${result.await()}")
}
// Affiche "Le résultat est 42" après deux secondes
```