

# IHM et Programmation Mobile

L3 MIAGE

2024-2025

Leo Donati

# IHM et Programmation Mobile

- Partie I : le langage Kotlin

1. **Kotlin et Jetbrains**
2. Variables et types
3. Collections
4. Instructions
5. Fonctions
6. Objets



# Kotlin

## Histoire de Kotlin

- Crée par JetBrains en 2011
- Version 1.0 en 2016
- Annoncé en 2017 comme langage de programmation officiel d'Android par Google
- Version 1.5 en 2021
- Version actuelle : 2.0.21

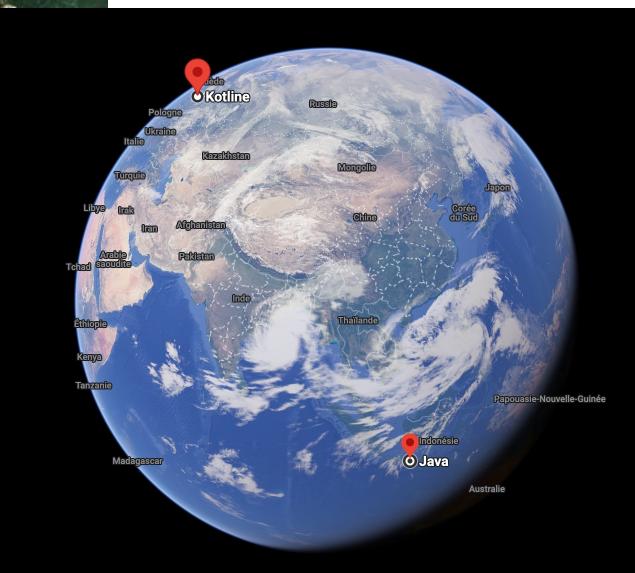


# JetBrains

- Entreprise fondée en 2000 à Prague par 3 développeurs russes :
  - Sergey Dmitriev,
  - Valentin Kipyatkov
  - Eugene Belyaev.
- Leur spécialité est le développement de IDE dont le plus connu est IntelliJ IDEA pour Java (2001) qui est écrit en Java et en Kotlin



# Ile de Kotlin



# Hello World en Kotlin



```
1 fun main() {  
2     println("Hello, World!")  
3 }
```



```
1 kotlinc helloworld.kt -include-runtime -d helloworld.jar  
2 java -jar helloworld.jar
```

# Objectifs

- Améliorer la productivité des développeurs
- Combler les lacunes de Java
- Offrir une syntaxe plus concise et expressive
- Prise en charge des fonctionnalités modernes de programmation
- Interopérabilité avec le code Java existant
- Faciliter le développement d'applications Android

# Caractéristiques du langage

- Orienté objet
- Fonctionnel
- Statiquement typé
- Inférence de type
- Types nullables
- Interopérable avec Java
- Développement classique (VM Java)
- développement Android
- développement côté Serveur (pour produire des pages HTML, des services web, des API, des micro-services, etc.) : KTOR
- Développement multiplateforme (iOS)

# IHM et Programmation Mobile

- Partie I : le langage Kotlin

1. Kotlin et Jetbrains
2. Variables et types
3. Collections
4. Instructions
5. Fonctions
6. Objets

# Déclaration d'une variable

- **Immutable** avec `val`



```
1 val a : Int
2 val b = 3
3 a = 2 * b
4 // a et b ne peuvent plus changer
5 // de valeur (ni de type)
```

- **Mutable** avec `var`



```
1 var x : Double
2 var y = 3.0
3 x = 2 *y
4 // x et y peuvent changer
5 // de valeur mais pas de type
```

- Le langage déduit le type de la variable du contexte si possible
  - Lors d'une affectation
  - D'après la valeur de retour d'une fonction
  - D'après les arguments d'une fonction

## Inférence de type



```
1 val s = "Bonjour"
2 // le type de s est String
3 var b = False
4 // le type de b est Boolean
```

- Int : entier sur 32 bits
- Long : entier long sur 64 bits
- Short : entier court sur 16 bits
- Byte : octet (8 bits)
- Float : flottant sur 32 bits
- Double : double sur 64 bits
- Char : caractère Unicode (16 bits)
- Boolean : booléen (8 bits)

- Type Unit



```
1 val x = 3.5.plus(4.5).times(2)
2 val y = println(x)
3 println(y)
4 // x vaut 16.0
5 // y vaut kotlin.Unit
```

## Types primitifs

- Les Strings sont des chaînes de caractères Unicode
  - Immuables
  - Peuvent contenir des escape chars comme « \n »
  - Peuvent être définies sur plusieurs lignes avec les triples guillemets
- Méthodes innombrables pour travailler sur les chaines

# String



```
1 val chaine = """  
2 Bonjour,  
3 Je suis une chaîne de caractères  
4 sur plusieurs lignes.  
5 """  
6 // extraire les mots de la chaîne  
7 val mots = chaine.split(" ", "\n")
```

# Interpolation et concaténation

- La **concaténation** de chaînes se fait avec l'opérateur +
- L'**interpolation** est le fait d'inclure la valeur d'une variable dans la chaîne en la convertissant
  - Se fait avec  **`${var}`**



```
1 println("la variable x vaut ${x} " + " et la variable y = ${y}")  
2 //la variable x vaut 16.0et la variable y = kotlin.Unit  
3 println("la longueur de chaine est ${chaine.length}")  
4 //la longueur de chaine est 77
```

- Null safety

- Par défaut dans Kotlin une variable ne peut pas être null
- On peut spécifier qu'une variable est d'un type nullable
- On peut tester pour null avec le Elvis Operator
- On peut gérer des appels de fonctions qui renvoie null de façon sûre
  - Avec ?.
  - Avec !!.

# Types nullables



```
1 var nomClient: String?  
2 var prenom: String?  
3 nomClient = "Leo Donati"  
4 prenom = nomClient?.split(" ")?.get(0)  
5 println(prenom)  
6 nomClient = null  
7 prenom = nomClient?.split(" ")?.get(0)  
8 println(prenom)
```

- On utilise !! sur un type nullable ou une méthode qui peut renvoyer null lorsqu'on est **sûrs que null ne peut pas arriver !**
- Risque :
  - En cas de null il y a une exception de type NullPointerException qui est levée
  - Réservé à des cas exceptionnels

# Opérateur !!



```
1 var nomClient: String? = "Leo Donati"
2 val len = nomClient!!.length
3 // len est de type Int et vaut 10
4 // val len = nomClient.length pas accepté à cause de la Null Safety
5
```



- L'opérateur `:?` Appelé *Elvis Operator* permet de gérer le cas particulier où une variable nullable est null en proposant une valeur alternative

## Opérateur Elvis `:?`



```
1 var nomClient: String?  
2 var prenom: String  
3 nomClient = null  
4 prenom = nomClient?.split(" ")?.get(0) ?: "Inconnu"  
5 print(prenom)
```

# IHM et Programmation Mobile

- Partie I : le langage Kotlin

1. Kotlin et Jetbrains
2. Variables et types
3. Collections
4. Instructions
5. Fonctions
6. Objets

# Listes

- Les listes sont des collections ordonnées d'éléments de même type
- Chaque élément peut apparaître plus d'une fois à des endroits différents
- On peut accéder aux éléments de la liste par leur index qui commence à 0

# Listes immuables

- On crée une liste immuable avec la commande `listOf` suivi par les éléments de la liste
- Le type déduit est `List<T>` où `T` est le type des éléments



```
1 val profs = listOf("Leo", "Michel", "Michel", "Michel", "Greg")
2 print(profs)
3 print(profs.size)    // 5
4 print(profs[0])      // Leo
5 print(profs.indexOf("Michel")) // 1
```

# Listes mutables

- On crée une liste Mutable avec mutableListOf
- Les éléments peuvent être modifiés; effacés et nouveau x éléments peuvent être insérés



```
1 val etudiants = mutableListOf("Julien", "Khalid", "Emma")
2 etudiants.add("Christophe")
3 etudiants.remove("Julien")
4 etudiants[0] = "Sofiane"
5 print(etudiants)
6 // [Sofiane, Emma, Christophe]
```

# Arrays

- Les Arrays sont utilisés pour stocker des éléments accessibles via leur indice
  - Ils sont toujours mutables mais leur taille est fixe
  - Les types des éléments peuvent être différents
- Optimal pour
  - Des recherches
  - Des ordonnancements
  - Automatiser des tâches sur les éléments
- L'opérateur + concatène les Arrays

# Création d'un array



```
1 val tableau = arrayOf("Julien", 34, 3.14)
2 println(java.util.Arrays.toString(tableau))
3 //le type de tableau est Array<Any>
4 val personnes: Array<String> = arrayOf("Leo", "Marc")
5 personnes[0] = "Emma"
6 //le type de personnes est spécifié
7 val tab = intArrayOf(1, 3, 5)
8 tab[0] = 2
9 println(java.util.Arrays.toString(tab))
10 //le type de tab est IntArray
```

- Le type Pair représente un couple générique deux valeurs
  - Les deux types des membres du couple sont libres
  - first donne accès au premier
  - Second au deuxième
  - toList convertit la paire en une liste
  - Une paire est immuable
- Il existe aussi le type Triple



```
1 var exemple = Pair("leo", 1.73 )  
2 // le type est Pair<String, Double>  
3 val (nom, hauteur) = exemple  
4 print("${nom} mesure ${hauteur} cm")
```

# Ensembles

- Le type Set permet de définir les ensembles
  - Des collections non ordonnées d'objets qui ne peuvent apparaître qu'en un seul exemplaire
  - Le type Set est immuable mais il existe la version MutableSet
  - Les opérations ensemblistes sont implémentées

# Dictionnaires

- En fait toutes les collections Java existent en Kotlin avec les mêmes méthodes en version Mutable et immuable (par défaut)
  - HashMap
  - HashSet
  - LinkedList
  - ArrayList

- Un range permet de définir un intervalle borné de valeurs contigües de types comparables
  - first → last
  - Avec .. dernier élément est **inclus**
  - Avec ..< le dernier est exclu
- Utilisation
  - Dans les boucles *for*, *when*
  - Dans des tests d'appartenance
  - Méthode *random* renvoie un élément aléatoire de l'intervalle

# range



```
1  val exRange = 1..100
2  print(exRange.first) //1
3  print(exRange.last) //100
4  print(55 in exRange) //true
5  for (i in exRange)
6      print(i)
```

# IHM et Programmation Mobile

- Partie I : le langage Kotlin

1. Kotlin et Jetbrains
2. Variables et types
3. Collections
4. Instructions
5. Fonctions
6. Objets

- Le **if** de Kotlin est classique **sauf qu'il renvoie une valeur** :
  - Sa valeur est la valeur de la dernière expression évaluée dans le bloc if ou dans le else

if



```
1  var x = 5
2  val y =
3      if (x > 5) {
4          x += 1
5          x*x
6      }
7      else {
8          x -= 1
9          x*x
10     }
11 // y vaut 16 et x vaut 4
```

- Permet de remplacer de façon élégante une série de if/else
- Ressemble un peu à un switch mais en plus simple
- On peut mettre
  - Des valeurs fixes
  - Des ranges
  - Des

# when



```
1 val note = 13
2 val resultat =
3     when (note) {
4         in 0..<10 -> "Insuffisant"
5         in 10..<12 -> "Passable"
6         in 12..<14 -> "Mention AB"
7         in 14..<16 -> "Mention Bien"
8         else -> "Mention TB"
9     }
```

31

- Le **for** en Kotlin doit itérer sur un range ou sur les éléments d'une collection



```
1 for (x in 1..10 step 2) {  
2     print(x)  
3 }  
4 println()  
5 for (x in 9 downTo 0 step 3) {  
6     print(x)  
7 }
```

for

- On peut itérer sur l'élément et sur l'index d'une collection



```
1 val pets = arrayOf("dog", "cat", "canary")  
2 for (element in pets) {  
3     print(element + " ")  
4 }
```



```
1 for ((index, element) in pets.withIndex()) {  
2     println("Item at $index is $element\n")  
3 }
```

- Permet de répéter un bloc de code un nombre fixé de fois
  - Espèce de for simplifié lorsqu'on n'a pas besoin de l'indice de boucle

## repeat



```
1 var x = 2
2 repeat(3) {
3     x = x*x
4 }
5 // x vaut 256
```

# IHM et Programmation Mobile

- Partie I : le langage Kotlin

1. Kotlin et Jetbrains
2. Variables et types
3. Collections
4. Instructions
5. Fonctions
6. Objets

- On définit une fonction avec le mot-clé **fun**
- Les arguments peuvent
  - Être nommés
  - Avoir des valeurs par défaut
  - Requis si pas de valeur par défaut
- Valeur de retour
  - Unit si pas de commande return

# Fonctions



```
1 fun bonjour(qui : String = "tout le monde") : String {  
2     return "Bonjour ${qui}"  
3 }  
4 bonjour()          // => Bonjour tout le monde  
5 bonjour("Leo")    // => Bonjour Leo  
6 bonjour(qui = "Toi")// => Bonjour Toi
```

- Il est conseillé que les arguments qui ont une valeur par défaut aient des noms clairs (en camelCase) qui explique leur usage
- Dans la définition on place d'abord les arguments positionnels requis et ensuite les arguments nommés optionnels

## Arguments nommés



```
1 fun dites(message: String, à: String = "tout le monde") {  
2     println(message + " " + à)  
3 }  
4 dites("Bonjour")    // => Bonjour tout le monde  
5 dites("Au revoir", à = "Leo") // => Au revoir Leo
```

- L'écriture compacte consiste à remplacer le code entre {.....} par un = qui donne le résultat de l'appel
  - Possible lorsqu'il y a une seule expression dans le corps de la fonction

## Fonctions compactes



```
1 fun somme(a: Int, b: Int) = a + b
2 // Le type de retour est Int
3 // la fonction est de type (Int, Int) -> Int
4 val resultat = somme(2, 3) // resultat vaut 5
```



```
1 fun pgcd(a: Int, b: Int) : Int =
2   if (b == 0) a
3   else pgcd(b, a % b)
4
5 val resultat = pgcd(12, 18) // resultat vaut 6
```

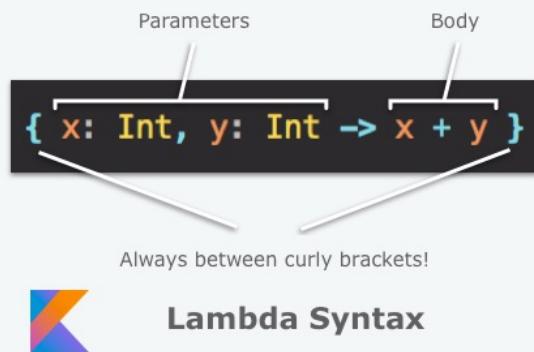
# Langage fonctionnel

- Dans Kotlin les fonctions sont des objets de première classe :
  - On peut les passer en argument à d'autres fonctions
  - On peut les utiliser comme valeur de retour
  - On peut stocker des fonctions dans des structures de données

```
● ● ●  
1 typealias FonctionReelle = (Double) -> Double  
2  
3 fun main() {  
4     val f : FonctionReelle = { x -> x*x + 1 }  
5     val g : FonctionReelle = { it * it - 1 }  
6  
7     val tab : Array<FonctionReelle> = arrayOf(f, g, ::sin, ::cos, ::tan)  
8     for (fonction in tab) {  
9         println(fonction(0.0))  
10    }  
11 }
```

# Fonctions lambda

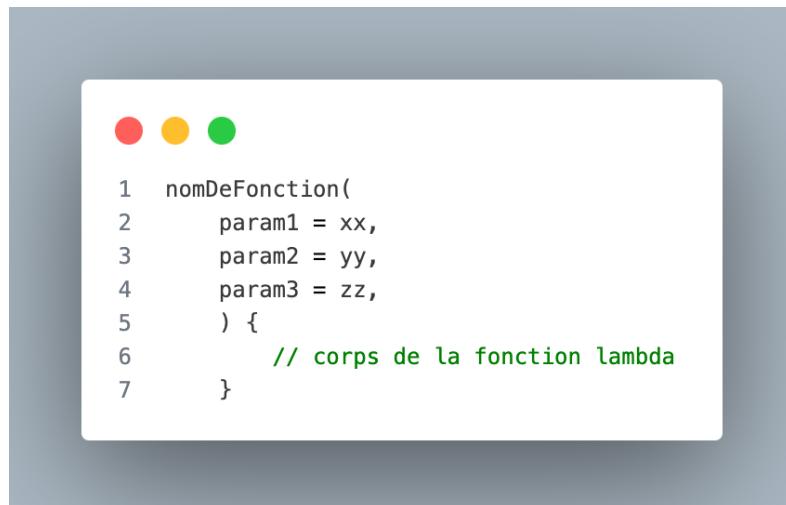
- Les fonctions lambda sont des expressions qui représentent des fonctions qui n'ont pas de noms
- Syntaxe : { params -> body }



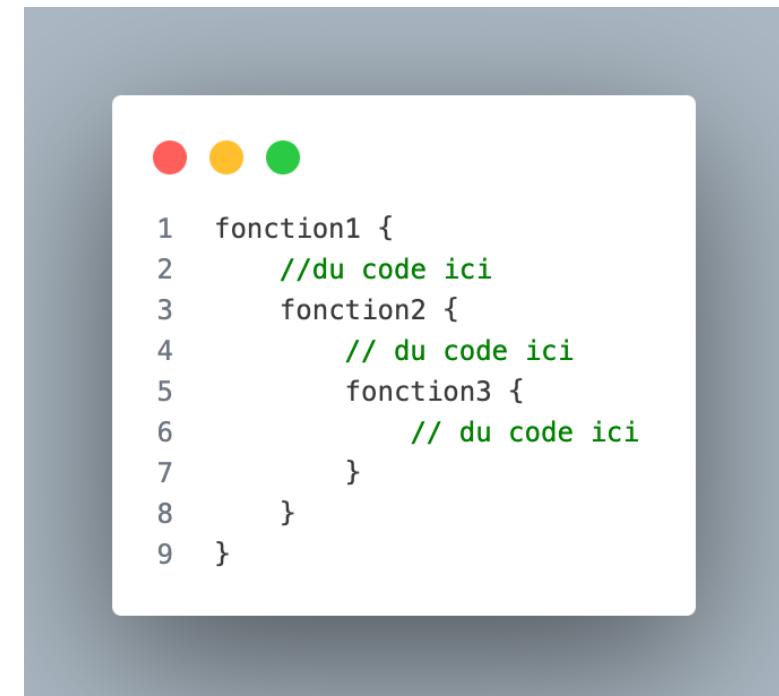
```
1  val noms = listOf("Leo", "Michel", "Stéphane", "Greg")
2  val plusLong = max(noms) {
3      a: String, b: String -> (a.length - b.length)
4  }
5  println(plusLong) // Stéphane
```

# Trailing lambda

- Lorsque le dernier argument d'une fonction est une fonction, et que l'on veut lui passer une fonction lambda, alors on a le droit d'écrire le code de la lambda à l'extérieur des parenthèses qui donnent les autres arguments de la fonction
  - Exemples dans gradle
  - Exemples dans Jetpack Compose



```
1 nomDeFonction(  
2     param1 = xx,  
3     param2 = yy,  
4     param3 = zz,  
5     ) {  
6         // corps de la fonction lambda  
7     }
```



```
1 fonction1 {  
2     //du code ici  
3     fonction2 {  
4         // du code ici  
5         fonction3 {  
6             // du code ici  
7         }  
8     }  
9 }
```

# Filtrage des listes

- La méthode *filter* du type **List** permet de créer une nouvelle liste qui ne contient que les éléments qui passent le filtre
  - utilise des lambda
- Le filtrage peut être :
  - Lazy : il n'a lieu que quand on veut vraiment lire le résultat (au runtime)
  - Eager : il a lieu tout de suite (par défaut)



```
1 val books = listOf("nature", "biology", "birds")
2 println(books.filter { it[0] == 'b' })
```

# Filtres paresseux

- Pour mettre en place un filtrage paresseux on passe par des séquences qui permettent l'évaluation paresseuse des éléments d'une collection



```
1 val instruments = listOf("viola", "cello", "violin")
2 val filtered = instruments.asSequence().filter { it[0] == 'v'}
3 val newList = filtered.toList()
```

# IHM et Programmation Mobile

- Partie I : le langage Kotlin

1. Kotlin et Jetbrains
2. Variables et types
3. Collections
4. Instructions
5. Fonctions
6. Objets

# Classes et objets

- Comme en Java il y a des classes dont on crée des instances qui sont des objets, il y a des classes abstraites et des interfaces.
  - Héritage simple
  - Implémentation de plusieurs interfaces
- Ce qui est simplifié
  - La syntaxe des constructeurs, des accesseurs
- Ce qui est ajouté
  - Les data classes
- Ce qu'il n'y a plus
  - Méthodes statiques

- Syntaxe typique

- Le constructeur principal est défini dans l'en-tête de la classe
- Les attributs sont marqués val ou var
- Peuvent avoir des valeurs par défaut
- Les méthodes sont des fonctions internes à la classe qui ont accès aux attributs

# Définir une classe



```
1 class Pizza(  
2     val name: String,  
3     val prix: Double  
4 ) {  
5     fun print() {  
6         println("Pizza $name : $prix")  
7     }  
8 }  
9  
10 val myPizza = Pizza("4 fromages", 12.5)  
11 myPizza.print() // => Pizza 4 fromages : 12.5
```

- Les arguments du constructeur sans val ou var sont passés au constructeur et à l'initialiseur
  - Mais ils n'existent plus dans l'objet
- Le bloc init() peut être utile pour mettre en place un travail d'initialisation de l'objet

# initialiseur



```
1  class Personne(  
2      nom: String,  
3      prenom: String,  
4      val age: Int  
5  ) {  
6      val fullName = "$prenom $nom"  
7      init {  
8          println("Création de la personne $nom $prenom")  
9      }  
10     fun print() {  
11         println("$fullName, $age ans")  
12     }  
13 }  
14  
15 val p = Personne("Doe", "John", 30)  
16 p.print() // => John Doe, 30 ans
```

# Constructeur secondaire

- On peut ajouter autant de constructeurs secondaires que l'on veut qui doivent appeler le constructeur primaire avec `this`
- Syntaxe
  - `constructor(args) : this(something) { ... du code }`



```
1 class Circle(val radius:Double) {
2     constructor(name:String) : this(1.0)
3     constructor(diameter:Int) : this(diameter / 2.0) {
4         println(" in diameter constructor")
5     }
6     init {
7         println("Area: ${Math.PI * radius * radius}")
8     }
9 }
10 val c = Circle(2.0)      // Area: 12.566370614359172
11 val c2 = Circle("circle")// Area: 3.141592653589793
12 val c3 = Circle(diameter = 4) // Area: 12.566370614359172 in diameter constructor
```

- Les classes et les objets
- Les méthodes et les propriétés
- On va voir ça avec un exemple

```
1 class Rectangle(var width: Double) {  
2     var color: String = "white"  
3     val isSquare: Boolean  
4         get() {  
5             return width == height  
6         }  
7     var area: Double  
8         get() {  
9             return width * height  
10        }  
11        set(value) {  
12            val ratio = Math.sqrt(value / area)  
13            width *= ratio  
14            height *= ratio  
15        }  
16 }
```

```
1 val r = Rectangle(10.0, 20.0)  
2 println(r.isSquare) // => false  
3 println(r.area) // => 200  
4 r.area = 300  
5 println(r.width) // => 12.24744871391589  
6 println(r.height) // => 24.49489742783178  
7 r.color = "blue"  
8 println(r.color) // => blue
```

- En Kotlin par défaut les classes sont **final**
  - On ne peut pas créer des classes filles
- Si on veut autoriser les sous-classes il faut ajouter le mot-clé **open**
- Les méthodes et propriétés redéfinies doivent être marquées **override**

# Héritage



```
1 open class A
2
3 class B : A()
```



```
1 open class A(val x: Int)
2
3 class B(val a: Int, val b: String) : A(a)
```

# Interface

- **Format:** `interface NameOfInterface { interfaceBody }`
- Comme en Java
  - Définit le contrat que devront respecter toutes les classes qui implémentent l'interface
  - Contient la signature des méthodes et les noms des propriétés
  - Peut hériter d'autres interfaces

# Classes abstraites

- Doivent avoir le mot-clé **abstract**
- Alors on ne pourra pas créer d'instances de cette classe
- Le mot clé **open** est inutile car on devra forcément subclasser cette classe
- Les méthodes et propriétés marquées **abstract** devront obligatoirement être redéfinies avec **override**

# Extensions

- Les extensions permettent d'ajouter des méthodes et des propriétés à une classe dont on n'est pas le propriétaire
  - Alternative simple à l'héritage qui n'est pas toujours possible



```
1 fun Int.estPair() = (this % 2 == 0)
2 print(4.estPair()) // => true
```



```
1 val Int.km : Double
2     get() { return this * 1000.0 }
3 val Int.m : Double
4     get() { return this.toDouble() }
5 val Int.cm : Double
6     get() { return this * 0.001 }
7 val distance = 42.km + 110.m + 55.cm
8 print(distance) // => 42110.055
```

- Une Data Class est une classe dont le seul but est de stocker des données
- Kotlin automatiquement
  - Génère getters et setters (pour var)
  - Implémente `toString()`, `equals()`, `hashCode()`, `copy()`
- Format:

```
data class <NameOfClass>(  
    parameterList )
```

## Data Class



```
1 data class Person(val name: String, val age: Int)  
2 // c'est tout !  
3 // toString(), equals(), hashCode() et copy() sont générés automatiquement  
4  
5 val p1 = Person("Alice", 29)  
6 println(p1) // => Person(name=Alice, age=29)  
7 val p2 = p1.copy(name = "Bob")  
8 println("$p2") // => Person(name=Bob, age=29)  
9 println(p1 == p2) // => false
```

- Comme Java pour définir un type qui ne prend qu'un nombre fini de valeurs spécifiées qui sont constantes
- **Format:**

```
enum class EnumName {  
    NAME1,  
    NAME2, ...  
    NAMEn }
```



```
1 enum class Couleur(val r: Int, val g: Int, val b: Int) {  
2     ROUGE(255, 0, 0),  
3     VERT(0, 255, 0),  
4     BLEU(0, 0, 255);  
5  
6     override fun toString()  
7         = String.format("%02X%02X%02X",  
8             r, g, b)  
9     }  
10    val col = Couleur.ROUGE  
11    print(col) // => FF0000
```

- Kotlin offre la mise en place du design pattern Singleton sans rien à faire
  - Une unique instance peut exister de la classe
- On utilise la syntaxe **object** au lieu de **class**

# Singleton



```
1 object Calculator {  
2     val Pi : Double  
3         get() = Math.PI  
4     fun add(n1: Int, n2: Int): Int {  
5         return n1 + n2  
6     }  
7 }  
8 print(Calculator.Pi) // => 3.141592653589793  
9 println(Calculator.add(2,4)) // => 6
```

- Le ***companion object*** est ce qui remplace la notion de méthode ou de variable statique dans Kotlin
  - C'est un *object* marqué comme *companion* à l'intérieur d'une classe
  - Toutes les instances de la classe partage l'objet compagnon

## Objet compagnon

```
● ● ●  
1 class PhysicsSystem {  
2     companion object WorldConstants {  
3         val gravity = 9.8  
4         val unit = "metric"  
5         fun computeForce(mass: Double, accel: Double): Double {  
6             return mass * accel  
7         }  
8     }  
9 }  
10 println(PhysicsSystem.gravity) // => 9.8
```

# Visibilité

- Les classes, les méthodes, les attributs et les propriétés ont une propriété de visibilité comme dans Java
  - public
  - private
  - protected
- Il y a en plus `internal` qui spécifie la visibilité au module
- Dans Kotlin la visibilité par défaut est `public` pour tout
- Pour les fonctions en dehors d'une classe :
  - Public : visible partout
  - Private : visible dans le même fichier
  - Internal : visible dans le même module

# Modules, Fichiers et Packages

- Pas de lien entre nom de fichier et nom de classe
  - Plusieurs classes reliées peuvent être dans le même fichier ainsi que des fonctions et des variables top-level
- Les packages sont les mêmes que dans Java
  - On définit un package avec la commande `package`
  - On importe le package avec `import`
- Qu'est ce que c'est un module dans Kotlin ?
  - Un module au sens IntelliJ IDEA
  - Un module au sens maven
  - Une source set de Gradle