Belajar Kotlin

Type Data dan Variable

Membutuhkan kata kunci **var** atau **val** Penulisannya :

```
var judul: String = "Type data & Variable"
```

> **judul => identifier**
>
> **String => type data**

penulisannya bisa juga :

```
val judul = "Type Data & Variable"
```

> **var** untuk nilai yang dapat berubah ketika sudah diinisialisasikan
>
> **val** untuk nilai yang tidak dapat berubah

Function

Fungsi dapat digunakan untuk mengembalikan nilai. Pemanggilan fungsi bisa diberi argumen atau tidak. Penulisannya :

```
1.  fun functionName(param1:  Type1, param2:  Type2,  ...):  ReturnType  {
2.      return result
3.  }
```

> **functionName => Nama sebuah function param1=>Parameter 1 Type1=>Type data dari parameter 1 param2=>Parameter 2 Type2=>Type data dari parameter 2 ReturnType=>Nilai kembalian (return)**

contoh :

```
fun dataUser(nama: String, umur: Int): String {
    return "My Name is $name, I'm $age years old"
}
```

nilai yang akan dikembalikan diikuti oleh kata kunci **return** Pemanggilan fungsi, bisa dilakukan dengan pendekatan tradisional seperti berikut:

```
fun main() {
    val user = dataUser("Randy Wiratama", 27)
    println(user)
}

fun dataUser(nama: String, umur: Int): String {
    return "My Name is $name, I'm $age years old"
}
```

If Expression

If expression direpresentasikan dengan kata kunci if. If akan mengeksekusi sebuah *statement* atau *expression* jika hasil evaluasi dari *expressions* yang diberikan pada blok if bernilai **true**. Sebaliknya, jika bernilai **false** maka proses yang ditentukan akan dilewatkan. Penulisan :

```
val openHours = 7
val now = 7
val office: String

if(now > openHours) {
    office = "Office already open"
} else if (now == openHours) {
    "Wait a minute, office will be open"
} else {
    "Office is closed"
}
```

Arrays

Array adalah tipe data yang memungkinkan kita untuk menyimpan beberapa objek didalam sebuah variable. Array di kotlin direpresentasikan oleh kelas **Array** yang memiliki fungsi **set** dan **get** serta properti **size**. Untuk membuat sebuah array, kita dapat memanfaatkan library function ***arrayOf()*** Penulisan :

```
val dataArray = arrayOf(1, 4, 5, "kotlin", true)
```

Nullable Types

Ketika mengembangkan sebuah program, ada satu hal yang tak boleh kita abaikan. Ia adalah **NullPointerException (NPE)**, sebuah kesalahan yang terjadi saat ingin mengakses atau mengelola nilai dari sebuah variabel yang belum diinisialisasi atau variabel yang bernilai **null**. Karena sangat umum terjadi dan bisa berakibat fatal, **NPE** terkenal dengan istilah ***"The Billion Dollar Mistake"***. Pada Kotlin kita dimudahkan untuk mengelola variable ***nullable*** Penulisan :

```
val text: String? = null
```

Jika ingin sebuah object bernilai null, kita bisa menambahkan tanda **?** seteleh menentukan tipe data object tersebut.

Safe calls Operator `?.`

Safe call akan menjamin kode yang kita tulis aman dari **NullPointerException**. Dalam menggunakan safe call, kita akan menggantikan tanda titik ==(**.**)== dengan tanda ==(**?.**)== saat mengakses atau mengelola nilai dari object *nullable* Penulisan :

```
val data: String? = null //deklarasi variable atau object
data?.length //safe calls operator
```

dengan safe call, kompiler akan melewatkan proses jika object / variable tersebut bernilai **null**

Elvis Operator `?:`

Elvis operator memungkinkan kita untuk menetapkan *default value* atau nilai dasar jika objek bernilai **null**.

```
val data: String? = null //deklarasi variable atau object null
val textLength = data?.length ?: 7
```

kode diatas sebenarnya sama dengan **if/else** berikut :

```
val.textLength = if (data != null) data.length else 7
```

Elvis Operator akan mengembalikan nilai `data.length` jika `data` tidak bernilai **null**. Sebaliknya, jika `data` bernilai **null** maka default value yang akan dikembalikan. Perhatikan penggunaan operator **non-null assertion** `(!!)`, misalnya seperti berikut:

```
val data: String? = null
val textLength = data!!.length
```

dengan menggunakan **non-null assertion** kompiler akan mengizinkan kita untuk mengakses atau mengelola nilai dari sebuah objek *nullable*.Namun penggunaan operator tersebut sangat tidak disarankan karena akan memaksa sebuah objek menjadi **non-null.** Sehingga ketika objek tersebut bernilai **null**, Anda tetap akan berjumpa dengan **NullPointerException**.

Enumeration

Enumeration merupakan salah satu fitur yang kita gunakan untuk menyimpan kumpulan objek yang telah didefinisikan menjadi tipe data konstanta. Enumeration dapat ditetapkan sebagai nilai kedalam sebuah variable dengan cara yang lebih efesien. Contohnya :

```
fun main () {
    val colorRed = Color.RED
    val colorGreen = Color.GREEN
    val colorBlue = Color.BLUE
}

enum class Color(val value: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}
```

Untuk mendapatkan daftar objek Enum kita bisa menggunakan fungsi `values()`. Sedangkan untuk mendapatkan nama dari objek Enum kita bisa menggunakan fungsi `valueOf()` seperti berikut:

```
fun main() {
    val color: Color = Color.valueOf("RED")
    println("Color is $color")
    println("Color value is ${color.value.toString(16)}")
}

enum class Color(val value: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}

/*
    output :
    Color is RED
    Color value is ff0000
*/
```

Di atas telah disebutkan bahwa setiap objek merupakan instance dari *enum* class yang kita definisikan. Lantas bagaimana cara kita mengecek instance dari Enum itu sendiri? Nah, untuk mengeceknya, gunakan **When Expression** seperti berikut:

```
fun main () {
    val color: Color = Color.GREEN

    when(color) {
        Color.RED -> print("Color is Red")
        Color.GREEN -> print("Color is Green")
        Color.BLUE -> print("Color is Blue")
    }
}

enum class Color(val value: Int) {
```

```
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}
```

Expression dan Statement

Contoh :

```
fun main() {
    val value1 = 10
    val value2 = 20

    sum(value1, value2)
}

fun sum(value1: Int, value2: Int) = value1 + value2
```

pada kode diatas deklarasi `value1` dan `value2` adalah sebuah *Statement*. Sedangkan pemanggilan fungsi `sum` merupakan sebuah *Expression*.

When Expression

Untuk menentukan statement dan expression kita menggunakan **If Expression**. Selain itu kita bisa juga menggunakan **When Expression**, yakni mekanisme yang memungkinkan nilai dari sebuah *variable/expression*, mampu mengubah alur program. Contoh :

```
fun main() {
    val value = 7

    when(value) {
        6 -> println(value is 6)
        7 -> println(value is 7)
        8 -> println(value is 8)
    }
}
```

`when` akan mencocokan semua argumen yang berada di setiap branch secara berurutan sampai salah satu kondisi terpenuhi. Di dalam `when` kita juga bisa menambahkan *branch* `else` seperti berikut:

```
fun main() {
    val value = 7

    when(value) {
        6 -> println(value is 6)
        7 -> println(value is 7)
```

```
        8 -> println(value is 8)
        else -> println("Failed)
    }
}
```

`when` juga memungkinkan kita untuk memeriksa *instance* dengan tipe data tertentu dari sebuah objek menggunakan `is` atau `!is`. Contoh :

```
fun main() {
    val anyType : Any = 100L
    when(anyType) {
        is Long -> println("The value is long")
        is String -> println("The value is string")
        else -> println("Undefined")
    }
}
```

Selain itu, when expression juga bisa kita gunakan untuk memeriksa nilai yang terdapat pada sebuah **Range** atau **Collection**. Range sendiri merupakan salah satu tipe data yang unik di mana kita dapat menentukan nilai awal dan nilai akhir.

```
fun main() {
    val value = 15
    val ranges = 10..50
    when(value) {
        is ranges -> println("The value is in the range")
        !is ranges -> println("The value is outside the range")
        else -> println("Undefined")
    }
}
```

While dan Do While

- While

Contoh penulisan :

```
fun main () {
    var counter = 1
    while(counter <= 7) {
        println("Hello World!")
        counter++
    }
}
```

- Do While

Contoh penulisan :

```
fun main() {
    var counter = 1
    do {
        prinln("Hello World)
        counter++
    } while (counter <= 7)
}
```

Saat menggunakan While dan Do While perhatikan *infinite loop*, yaitu kondisi dimana proses perulangan berlangsung terus menerus sampai aplikasi menjadi *crash*.

Range

Dengan menggunakan range kita dapat menentukan nilai awal dan nilai akhir pada range. Range direpresentasikan dengan operator `..` atau dengan fungsi `rangeTo()` dan `downTo()`. Contoh :

```
val rangeInt = 1.rangeTo(10) //output 1 2 3 4 5 6 7 8 9 10
val downInt = 10.downTo(1) //output 10 9 8 7 6 5 4 3 2 1

fun main() {
    val dataRange = 10.downTo(1)
    if(7 in dataRange) {
        println("Value 7 is available)
    }
}
```

For Loop

For dapat digunakan pada **Ranges, Collections, Arrays** dan apapun yang menjadi *interator*. Contoh :

```
fun main() {
    //penggunaan for pada Ranges
    val ranges = 1..5
    for (i in ranges) {
        println("values is $i)
    }
}
```

Kita juga dapat mengakses `index` setiap elemen yang ada pada ranges dengan memanfaatkan fungsi `withIndex()` seperti berikut :

```
fun main() {
    val data = 1.rangeTo(10) step 3
    for ((index, value) in ranges.withIndex()) {
        println("value = $value, index = $index")
    }

    /*
    output :
    value = 1, index = 0
    value = 4, index = 1
    value = 7, index = 2
    value = 10, index = 3
    */
}
```

Kita menggunakan kata kunci `for` untuk memulai proses perulangan. Untuk tujuan yang sama kita juga bisa memanfaatkan salah satu ekstensi pada kotlin yaitu `forEach`. Contohnya seperti berikut :

```
fun main() {
    val data = 1.rangeTo(10) step 3
    data.forEach { value ->
        println("value is $value")
    }
}
```

`forEach` pada kode diatas merupakan sebuah **lambda expression** yang hanya memiliki satu argumen yaitu nilai tunggal yang dicakup pada `data/ranges`. Jika kita mendapatkan index dari tiap nilai yang dicakup kita bisa menggunakan ekstensi `forEachIndexed` seperti berikut :

```
fun main() {
    val data = 1.rangeTo(10) step 3
    data.forEachIndexed { index, value ->
        println("value = $value, index = $index")
    }
}
```

Jika kita hanya ingin menggunakan argumen index, maka kita bisa mengubah argumen value menjadi `_` seperti berikut:

```
fun main() {
    val data = 1.rangeTo(10) step 3
    data.forEachIndexed { index, _ ->
        println("index = $index")
    }
}
```

Break dan Continue

Continue digunakan untuk melewatkan proses iterasi dan lanjut dengan proses iterasi berikutnya. Sementara itu, Break digunakan untuk menghentikan proses iterasi.

Berikut adalah contoh proses iterasi pada kode di atas. Kita akan coba melewatkannya jika nilai yang dihasilkan adalah null.

```
fun main() {
    val listOfInt = listOf(1,2,3, null, 5, null, 7)
    for (i in listOfInt) {
        if(i == null) continue
        print(i)
    }
}
```

Pada kode diatas kita menggunakan kata kunci `continue`. Jika hasil evaluasi *expression* yang diberikan bernilai **true**, maka proses iterasi akan dilewatkan dan lanjut pada proses iterasi berikutnya.

Contoh penggunaan **break**

```
fun main() {
    val listOfInt = listOf(1,2,3, null, 5, null, 7)
    for (i in listOfInt) {
        if(i == null) break
        print(i)
    }
}
```

Penggunaan `break` pada kode diatas, akan menghentikan proses *looping* jika bertemu dengan nilai **null**

Break dan Continue Labels

Pada kotlin, sebuah *expression* dapat ditandai dengan sebuah label. Label pada kotlin memiliki sebuah *identifier* yang diikut dengan tanda `@`. Contoh dari sebuah label adalah **foo@** dan **bar@**.

Untuk melabeli sebuah *expression*, kita cukup menempatkan label didepannya, seperti berikut :

```
fun main() {
    loop@ for (i in 1..10) {
        println("Outside Loop")

        for (j in 1..10) {
            println("Inside loop")
            if(j > 5) break@loop
        }
```

```
    }
}
```

Pada kode diatas, `break` yang sudah ditandai dengan label akan dilompati ke titik awal proses perulangan yang sudah ditandai dengan label. break akan menghentikan proses perulangan terluar dari dalam proses perulangan, di mana break tersebut dipanggil.

Data *Class*

Kotlin mengenalkan konsep data class yang merupakan sebuah kelas sederhana yang bisa berperan sebagai data *container*. Data class adalah sebuah kelas yang tidak memiliki logika apapun dan juga tidak memiliki fungsionalitas lain selain menangani data. Data class mampu menyediakan beberapa fungsionalitas yang biasanya kita butuhkan untuk mengelola data hanya dengan sebuah *keyword* `data`.

Penulisannya :

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

Hanya dengan satu baris kode diatas, kompiler akan secara otomatis menghasilkan `constructor`, `toString()`, `equals()`, `hashCode()`, `copy()` dan juga fungsi `componentN()`. Beberapa hal yang perlu diperhatikan dalam membuat sebuah data class adalah :

1. Konstruktor utama pada kelas tersebut harus memiliki setidaknya satu parameter
2. Semua konstruktor utama perlu dideklarasikan sebagai **val** atau **var**
3. Modifier dari sebuah data class tidak bisa **abstract**, **open**, **sealed**, atau **inner**

Penggunaan Data *Class*

Perhatikan kode dibawah ini :

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

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

Apakah perbedaan kedua kode yang diatas ? Untuk mengetahuinya buka IDE atau Android Studio dan buat sebuah file dengan nama `DataClasses.kt`. Ketikkan kedua kelas tadi didalamnya dan buat fungsi `main()` sebagai tempat dimana kita akan mencoba mengelola atau mengoperasikan kedua kelas tersebut.

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

data class DataUser(val name: String, val age: Int)

fun main() {
```

```
}
```

untuk menampilkan 2 buah objek yang dibuat dari kelas `User` dan `DataUser`. Tambahkan kode berikut kedalam `main()`

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

data class DataUser(val name: String, val age: Int)

fun main() {
    val user = User("Randy", 27)
    val dataUser = DataUser("Randy", 27)

    println(User)
    println(DataUser)
}
```

Dari hasil dari kode diatas, kita langsung mengetahui semua informasi dari `DataUser` hanya melihat *value* dari properti yang ada. Data Class akan secara otomatis menghasilkan fungsi `toString()` didalamnya. tanpa data class, kita perlu membuat fungsi `toString()` secara manual untuk mendapatkan informasi seperti yang diberikan oleh objek `DataUser`. Sebagai contoh, untuk menampilkan informasi yang jelas dari objek `User`, maka kita perlu menambahkan fungsi `toString()` seperti berikut:

```
class User(val name: String, val age: Int) {
    override fun toString(): String {
        return "User(name=$name, age=$age)"
    }
}
```

Dengan menambahkan fungsi `toString()` seperti di atas, maka objek `User` akan bisa menghasilkan teks yang sama dengan objek `DataUser`.

Selanjutnya, kelebihan lain dari data class adalah ia sudah memiliki fungsi `equals()` secara otomatis, yang berguna untuk komparasi 2 buah objek. Contohnya seperti berikut :

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

fun main() {

    val dataUser = DataUser("randy", 27)
    val dataUser2 = DataUser("randy", 27)
    val dataUser3 = DataUser("wiratama", 24)

    println(dataUser.equals(dataUser2)) // Hasilnya true
    println(dataUser.equals(dataUser3)) // Hasil false
```

```
}
```

Lain halnya jika kita melakukan komparasi pada 2 buah objek yang bukan dari data class. Kita tidak bisa mendapatkan hasil yang akurat karena konsol akan selalu menghasilkan nilai false.

Dan jika Anda menginginkan hasil yang akurat seperti pada data class, maka Anda perlu membuat fungsi `equals()` secara manual:

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

    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (javaClass != other?.javaClass) return false

        other as User

        if (name != other.name) return false
        if (age != other.age) return false

        return true
    }

    override fun hashCode(): Int {
        var result = name.hashCode()
        result = 31 * result + age
        return result
    }
}
```

Kita perlu menuliskan beberapa *boilerplate* code di atas untuk mendapatkan hasil yang sesuai. Belum lagi ketika menambahkan fungsi `equals()`, Kita juga perlu menambahkan fungsi `hashCode()`.

Menyalin dan Memodifikasi Data *Class*

Data class juga memungkinkan kita untuk menyalin sebuah objek dengan sangat mudah hanya dengan memanfaatkan fungsi `copy()` di dalamnya. Sebagai contoh :

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

fun main(){
    val dataUser = DataUser("randy", 27)
    val dataUser2 = DataUser("randy", 27)
    val dataUser3 = DataUser("wiratama", 28)
    val dataUser4 = dataUser.copy()

    println(dataUser4)

    /*
```

```
    output :
    name = randy, age = 27
    */
}
```

Menariknya, dengan fungsi `copy()` kita juga bisa memodifikasi objek tersebut dengan nilai yang baru. Contoh :

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

fun main(){
    val dataUser = DataUser("randy", 27)
    val dataUser2 = DataUser("randy", 27)
    val dataUser3 = DataUser("wiratama", 28)
    val dataUser4 = dataUser.copy(age = 26)

    println(dataUser4)

    /*
    output :
    name = randy, age = 26
    */
}
```

Destructuring Declarations

Destructuring Declarations adalah proses memetakan objek menjadi sebuah variable. Ini bisa dengan mudah kita lakukan pada data class. Dengan fungsi `componentN` yang ada pada data class, kita bisa menguraikan sebuah objek menjadi beberapa properti yang dimilikinya, sebagai contoh :

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

fun main() {
    val dataUser = DataUser("Randy", 27)

    val name = dataUser.component1()
    val age = dataUser.component2()

    println("My name is $name, I am $age years old")

    /*
    console output :
    My name is Randy, I am 27 years old
    */
}
```

fungsi `component1()` dan `component2()` dihasilkan sesuai dengan jumlah properti yang ada pada data class tersebut. Maka jika sebuah data class memiliki sejumlah **N** properti, maka secara otomatis `componentN()` akan dihasilkan.

Kita juga dapat membuat beberapa variable dari objek secara langsung dengan kode seperti berikut :

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

fun main(){
    val dataUser = DataUser("Randy", 27)
    val (name, age) = dataUser

    println("My name is $name, I am $age years old")

    /*
    console output :
    My name is Randy, I am 27 years old
    */
}
```

Kesimpulannya, seperti aspek-aspek lain dari kotlin, data class bertujuan untuk mengurangi jumlah kode *boilerplate* yang kita tuliskan. Dan perlu diketahui bahwa data class tidak hanya sekedar untuk mengelola properti yang ada didalamnya. Ketika mempunyai data yang sangat kompleks, kita juga bisa menerapkan sebuah *behavior* didalam data class. Contoh sederhananya kita bisa membuat fungsi didalam data class seperti berikut :

```
data class DataUser(val name: String, val age: Int) {
    fun intro() {
        println("My name is $name, I am $age years old")
    }
}
```

dan langsung mengaksesnya pada fungsi `main()`

```
fun main() {
   val dataUser = DataUser("Randy", 27)
   dataUser.intro()
}
```

Collection

Collection adalah sebuah objek yang dapat menyimpan kumpulan objek lain termasuk data class. Dengan collection kita bisa menyimpan banyak data sekaligus. Di dalam collections terdapat beberapa objek turunan, di antaranya adalah **List**, **Set**, dan **Map**.

List

Dengan List kita dapat menyimpan banyak data menjadi satu objek. Sebagai contoh, kita bisa membuat sebuah List yang berisi sekumpulan data angka, karakter atau yang lainnya. Yang menarik, sebuah List tidak hanya bisa menyimpan data dengan tipe yang sama. Namun juga bisa berisi bermacam - macam tipe data seperti **Int**, **String**, **Boolean** atau yang lainnya. Cara penulisannya pun sangat mudah. Perhatikan saja beberapa contoh kode berikut :

```
val numberList : List<Int> = listOf(1, 2, 3, 4, 5)
```

Kode di atas adalah contoh dari satu objek List yang berisi kumpulan data dengan tipe Integer. Karena kompiler bisa mengetahui tipe data yang ada dalam sebuah objek List, maka tak perlu kita menuliskannya secara eksplisit. Ini tentunya akan menghemat kode yang kita ketikkan:

```
val numberList = listOf(1, 2, 3, 4, 5)
val charList = listOf('a', 'b', 'c')
```

Sedangkan untuk membuat List dengan tipe data yang berbeda, cukup masukkan saja data tersebut seperti kode berikut :

```
val dataList = listOf('a', "Kotlin", 3, true)
```

Karena setiap objek pada Kotlin merupakan turunan dari kelas Any, maka variabel anyList tersebut akan memiliki tipe data `List<Any>`. Jika kita tampilkan list di atas maka konsol akan menampilkan:

```
[a, Kotlin, 3, true]
```

Bahkan kita pun bisa memasukkan sebuah data class ke dalam List tersebut:

```
val dataList = listOf('a', "Kotlin", 3, true, User())
```

Ketika bermain dengan sebuah List, tentunya ada saat di mana kita ingin mengakses posisi tertentu dari List tersebut. Untuk melakukannya, kita bisa menggunakan fungsi *indexing* seperti berikut:

```
println(dataList[3])
```

Jika kita berusaha menampilkan item dari List yang berada dari luar ukuran List tersebut adalah akan muncul error dengan pesan error `Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5`

Pesan di atas memberitahu kita bahwa List telah diakses dengan indeks ilegal. Ini akan terjadi jika indeks yang kita inginkan negatif atau lebih besar dari atau sama dengan ukuran List tersebut.

Jika kita ingin mengubah atau menambah data pada suatu List, kita harus menambahkan fungsi `mutableListOf` seperti berikut :

```
val dataList = mutableListOf('a', "Kotlin", 3, true, User())
```

dengan begitu, `dataList` sekarang merupakan sebuah List yang bersifat **mutable** dan kita memanipulasi data didalamnya.

```
val dataList = mutableListOf('a', "Kotlin", 3, true, User())

dataList.add('tambahan List') // menambahkan item pada akhir List
dataList.add(1, 'tambahan List') //menambahkan List pada index ke-1
dataList[3] = false // mengubah nilai item pada index ke-3
dataList.removeAt(1) //mengahapus item User() berdasarkan index atau posisi nilai
di dalam array
```

Set

Set merupakan sebuah collection yang hanya dapat menyimpan nilai yang unik. Ini akan berguna ketika kita menginginkan tidak ada data yang sama atau duplikasi dalam sebuah collection. Kita bisa mendeklarasikan sebuah Set dengan fungsi `setOf`.

```
val integerSet = setOf(1, 2, 4, 2, 1, 5)
```

Perhatikan kode di atas. Di sana terdapat beberapa angka yang duplikat, yaitu angka 1 dan 2. Silakan tampilkan pada konsol dan lihat hasilnya.

```
println(integerSet)

// Console output: [1, 2, 4, 5]
```

Secara otomatis fungsi `setOf` akan membuang angka yang sama, sehingga hasilnya adalah `[1, 2, 4, 5]`. Selain itu urutan pada Set bukanlah sesuatu yang penting, sehingga apabila kita bandingkan dua buah Set yang memiliki nilai yang sama dan urutan yang berbeda, akan tetap dianggap sama.

```
val setA = setOf(1, 2, 4, 2, 1, 5)
val setB = setOf(1, 2, 4, 5)
println(setA == setB)
```

```
// Console output : true
```

Kita juga dapat melakukan pengecekan apakah sebuah nilai ada di dalam Set dengan menggunakan kata kunci `in`.

```
val setA = setOf(1, 2, 4, 2, 1, 5)
val setB = setOf(1, 2, 4, 5)
println(5 in setA)

// Console output : true
```

Kemudian ada juga fungsi union dan *intersect* untuk mengetahui gabungan dan irisan dari 2 (dua) buah Set. Sebagai contoh :

```
val setA = setOf(1, 2, 4, 2, 1, 5)
val setC = setOf(1, 5, 7)
val union = setA.union(setC) // Menggabungkan setA dengan setC
val intersect = setA.intersect(setC) //pengirisan setA dan setC

println(union)
println(intersect)

// union : [1,2,4,5,7]
// intersect : [1,5]
```

Pada Mutable Set, kita bisa menambah dan menghapus items Set, tetapi tidak dapat mengubah nilainya seperti pada List.

```
val dataSet = mutableSetOf(1,2,4,2,1,5)
dataSet.add(6) // Menambah item pada akhir set
dataSet.remove(1) // Menghapus item yang memiliki nilai 1
```

Map

Map adalah sebuah collection yang dapat menyimpan data dengan format **key-value**. Sebagai contoh :

```
val capital = mapOf(
    "Jakarta" to "Indonesia",
    "London" to "England",
    "Bangkok" to "Thailand"
)
```

> String yang berada pada sebelah kiri `to` adalah sebuah **key**
>
> String yang berada pada sebelah kanan `to` adalah sebuah **value**

Untuk mengakses nilai dari Map tersebut, kita bisa menggunakan key yang sudah dimasukkan. Misalnya kita bisa menggunakan key `Jakarta` untuk mendapatkan value `Indonesia`

```
println(capital["Jakarta])

// Console output : Indonesia
```

Atau bisa juga menggunakan fungsi `getValue()`:

```
println(capital.getValue("Jakarta"))

// Console output : Indonesia
```

Hasilnya sama saja. Namun sebenarnya terdapat sebuah perbedaan antara keduanya. Saat menggunakan simbol **[ ]** atau yang kita kenal dengan indexing, konsol akan menampilkan hasil **null** saat key yang dicari tidak ada di dalam Map. Sedangkan saat kita menggunakan `getValue()`, konsol akan memberikan sebuah **Exception**.

```
println(capital["Amsterdam"])

// Output: null


println(capital.getValue("Amsterdam"))

// Output: Exception in thread "main" java.util.NoSuchElementException: Key
Amsterdam is missing in the map.
```

Kita dapat menampilkan key apa saja yang ada di dalam Map dengan menggunakan fungsi `keys()`. Fungsi ini akan mengembalikan nilai berupa Set karena key pada Map haruslah unik.

```
val mapKeys = capital.keys

// mapKeys: [Jakarta, London, Bangkok]
```

Sedangkan untuk mengetahui nilai apa saja yang ada di dalam Map kita bisa menggunakan fungsi `values()`. Fungsi ini akan mengembalikan collection sebagai tipe datanya.

```

```

```
val mapValues = capital.values

// mapValues: [Indonesia, England, Thailand]
```

Untuk menambahkan **key-value** ke dalam map, kita perlu memastikan bahwa map yang digunakan adalah **mutable**. Mari kita ubah map capital yang sudah kita buat sebelumnya menjadi **mutable**.

```
val mutableCapital = capital.toMutableMap()
```

```
mutableCapital.put("Amsterdam", "Netherlands")
mutableCapital.put("Berlin", "Germany")
```

Note :

Perlu diperhatikan bahwa menggunakan mutable collection itu tidak disarankan. Apabila terdapat sebuah mutable collection yang diubah oleh lebih dari 1 proses, hasil nya akan sulit untuk diprediksi. Untuk itu, sebaiknya gunakan immutable sebisa mungkin, jika memang dibutuhkan untuk diubah, baru gunakan mutable.