KOTLIN, SIAPA SUKA

KOTLIN, SIAPA SUKA

Dasar

P. Tamami
BPPKAD Kab. Brebes

Untuk Istriku yang selalu
memberi semangat, dan
anak-anak yang selalu ceria

Daftar Isi

DAFTAR GAMBAR

DAFTAR TABEL

KATA PENGANTAR

Saat melihat keunggulan dari bahasa pemrograman Java yang mudah untuk di*maintenance*, dapat berjalan di berbagai *platform*, berorientasi objek, dan beberapa keunggulan lain, ada beberapa penyempurnaan yang dilakukan oleh bahasa pemrograman Kotlin, yang sama-sama berjalan di atas JVM.

Dalam buku ini akan dijelaskan dasar dari pemrograman Kotlin yang menawarkan penulisan kode yang lebih ringkas, menjamin kesalahan seluruh kelas dari *exception* **null**, dan yang tidak kalah penting adalah integrasinya dengan sistem yang dibangun dengan menggunakan bahasa Java.

Silahkan menikmati buku yang kurang dari sempurna ini, dan berharap penulis mendapatkan kritik yang membangun guna perubahan isi buku ini ke arah yang lebih sempurna.

4 Mei 2017

Penulis

BAB 1

MEMULAI

Perlu diketahui bahwa Kotlin ini adalah bahasa pemrograman yang berjalan di atas JVM, sehingga diperlukan Java Runtime untuk menjalankannya.

Cara termudah untuk memasangkan atau meng*install compiler* Kotlin adalah dengan mengunduh di halaman https://github.com/JetBrains/kotlin/releases/, kemudian melakukan *unzip* dan menambahkan direktori `bin` ke dalam *path* sistem.

Untuk memastikan bahwa Kotlin sudah terpasang dan dapat digunakan, kita seharusnya dapat menjalankan perintah berikut di konsol pada Linux atau *command prompt* milik Windows, berikut perintahnya :

```
1 kotlinc -version
```

Perintah tersebut sebetulnya untuk mencetak informasi tentang versi *compiler* Kotlin yang aktif. Dan seharusnya akan muncul informasi yang kurang lebih sebagai berikut :

```
1 info: Kotlin Compiler version 1.1.2-2
```

Tentunya versi yang keluar akan berbeda tergantung apa yang kita *install*.

Percobaan berikutnya adalah menampilkan versi *runtime environment* dari Kotlin, jika perintah `kotlinc` digunakan untuk melakukan *compile* (kompilasi) terhadap kode yang kita ketik / tulis menjadi bahasa biner, fungsi dari *runtime environment* adalah menerjemahkan bahasa biner hasil *compile* oleh `kotlinc` menjadi bahasa *native* sesuai sistem operasi yang

digunakan, inilah prinsip yang digunakan bahasa pemrograman Java yang tetap digunakan oleh Kotlin, karena memang Kotlin masih menggunakan JRE (*Java Runtime Environment*).

Perintah untuk melihat versi *runtime environment* dari Kotlin adalah sebagai berikut :

```
1 kotlin -version
```

Dengan hasil keluaran di layar monitor seperti ini :

```
1 Kotlin version 1.1.2-2 (JRE 1.8.0_121-b13)
```

Versi Kotlin seharusnya sama dengan versi *compiler*-nya. Sedangkan muncul tambahan informasi `JRE 1.8.0_121-b13`, inilah yang menunjukan bahwa Kotlin masih menggunakan JRE untuk menjalankan programnya, karena memang sebelum melakukan instalasi Kotlin, Java harus di*install* terlebih dahulu.

1.1 Katakan Hai

Setelah melakukan percobaan dasar seperti di atas, kita akan mencoba menjalankan kode pertama yang kita buat dengan Kotlin. Berikut adalah langkahnya :

1. Membuka editor teks seperti notepad, atom, notepad++, atau aplikasi sejenis.
2. Mengetikan kode berikut :

```
1 fun main(args: Array<String>) {
2     println("Hai, selamat datang")
3 }
```

3. Simpanlah dengan nama apapun, berikan ekstensi `kt`, misal kita beri nama *file* tersebut dengan `Test.kt`.
4. Buka konsol atau *command prompt* dan aktifkan ke direktori tempat kita simpan *file* `Test.kt` tadi.
5. *Compile file* `Test.kt` tersebut dengan perintah berikut :

```
1 kotlinc Test.kt
```

6. Hasil dari *compile* tersebut adalah berupa *file* `TestKt.class`
7. Untuk menjalankan hasil program yang telah kita *compile*, gunakan perintah berikut :

```
1 kotlin TestKt
```

8. Kemudian akan program / aplikasi akan menghasilkan keluaran sebagai berikut :

```
1 Hai, selamat datang
```

9. Sampai titik ini, kita berhasil menjalankan kode yang telah kita buat.

Jadi sebetulnya, untuk memulai koding dengan bahasa Kotlin cukup sederhana, tinggal siapkan *berekstensi* `kt`, kemudian sertakan blok kode program berikut :

```
1 fun main(args: Array<String>) {
2   ...
3 }
```

Seluruh program yang dibangun dengan Kotlin akan berawal dari fungsi `main` ini.

1.2 Sintak Dasar

1.2.1 Deklarasi Paket

Sama seperti bahasa pemrograman Java, deklarasi paket berada di awal kode seperti contoh berikut :

```
1 package nama.paket
2
3 import java.net.*
4 ...
```

Perbedaannya adalah bahwa nama paket tidak perlu disesuaikan atau disamakan dengan nama direktorinya seperti pada pemrograman Java. *File* kode sumber dapat ditempatkan dimanapun pada *drive*.

1.2.2 Deklarasi Fungsi

Deklarasi fungsi tanpa parameter dan tanpa nilai balikkan (*return*) akan terlihat seperti contoh kode berikut :

```
1 fun cetak(): Unit {
2   println("Hai, apa kabar")
3 }
```

Atau deklarasi `Unit` dapat dihilangkan dengan kode akan terlihat seperti ini :

```
1 fun cetak() {
2   println("Hai, apa kabar")
3 }
```

Untuk deklarasi fungsi dengan parameter akan terlihat seperti contoh kode berikut :

```
1 fun tambah(a: Int, b: Int): Int {
2   return a + b
3 }
```

Fungsi yang sama seperti diatas dapat dibuat lebih ringkas dengan nilai balikan *return* yang sudah diprediksi oleh Kotlin, kodenya menjadi seperti berikut ini :

```
1 fun tambah(a: Int, b: Int) = a + b
```

Untuk pembahasan lebih lanjut mengenai fungsi, akan dijabarkan dalam bagian tersendiri dalam buku ini.

1.2.3 Deklarasi Variabel

Deklarasi variabel dapat dilakukan untuk 2 (dua) cara. Yang pertama adalah variabel yang hanya dapat diisi satu kali, dan ada yang dapat diisi berkali-kali.

Kode untuk deklarasi variabel yang hanya dapat diisi 1 (satu) kali adalah sebagai berikut :

```
1 val a: Int = 2
2 // atau
3 val c = 2
4 // atau
5 val d: Int
6 d = 5
```

Untuk deklarasi variabel yang dapat diubah, kodenya adalah sebagai berikut :

```
1 var e = 2
2 e *= 2
```

1.2.4 Deklarasi Komentar

Seperti bahasa pemrograman Java dan Javascript, Kotlin juga menyediakan komentar dalam bentuk komentar baris dan komentar multi-baris. Kode untuk komentar satu baris adalah sebagai berikut :

```
1 // ini komentar 1 baris
```

Untuk kode komentar multi-bari adalah sebagai berikut :

```
1 /*  ini komentar
2     multi baris */
```

Namun tidak seperti bahasa pemrograman Java, komentar di Kotlin dapat bersarang bertingkat.

1.3 Logat

Beberapa logat yang biasa digunakan di Kotlin adalah seperti di bawah ini.

1.3.1 Membuat Kelas Data

Kelas data ini biasa digunakan untuk pembuatan kelas *entity*. Contoh kodenya adalah sebagai berikut :

```
1 data class Pegawai(val nim: String, val nama: String)
```

Dengan menambahkan deklarasi `data` di depan kelas, maka untuk kelas Pegawai ini akan disediakan fungsi-fungsi berikut secara otomatis :

- *Getters* dan *Setter* untuk seluruh properti
- *Method* `equals`.

- *Method* `hashCode`
- *Method* `toString`
- *Method* `copy`

1.3.2 Nilai *Default* Untuk Parameter Fungsi

Pada saat deklarasi fungsi, sebetulnya parameter dapat kita isikan dengan nilai *default* seperti berikut :

```
1 fun isiData(nama: String, kelamin: Int = 0) {
2   ...
3 }
```

Nantinya parameter `kelamin` akan terisi otomatis dengan `0`
Contoh kodenya adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   val nama = "tamami"
3   println("Halo, $nama")
4
5   isiData(nama)
6 }
7
8 fun isiData(nama: String, kelamin: Int = 0) {
9   println(kelamin)
10 }
```

Hasil keluarannya adalah sebagai berikut :

```
1 Halo, tamami
2 0
```

Penjelasannya adalah sebagai berikut, pada baris pertama menghasilkan keluaran teks `Halo, tamami`, yang sebetulnya hasil dari eksekusi perintah kode pada baris ke-3, yaitu :

```
1 println("Halo, $nama")
```

Dimana pemanggilan variabel `$nama` pada baris ke-2 dari *source code* terjadi, dan yang ditampilkan di layar monitor adalah isi dari variabel `$nama`, yaitu `tamami`.

Sedangkan pada baris kedua dari hasil keluaran, yaitu `0`, adalah hasil dari eksekusi kode pada bari ke-9, di dalam fungsi `isiData`, tepatnya pada perintah berikut :

```
1 println(kelamin)
```

Kenapa hasil keluarannya adalah `0`, alurnya adalah seperti ini, pada saat pemanggilan fungsi `isiData(nama)` pada baris ke-5, parameter `nama` pada fungsi `isiData` ini terisi dengan nilai `tamami`, karena parameter kedua, yaitu `kelamin` tidak disertakan pada pemanggilannya pada baris ke-5, sehingga parameter `kelamin` akan terisi otomatis dengan nilai `0` sebagaimana deklarasinya pada baris ke-8.

1.3.3 *Interpolasi* Teks

Interpolasi atau penyisipan teks akan terlihat seperti baris perintah berikut ini :

```
1 val nama: String = "tamami"
2 println("name $nama")
```

Nantinya sisipan teks dengan kode `$nama` akan terisi oleh variabel `nama` yang telah dideklarasikan sebelumnya.

1.3.4 Pemeriksaan Instan

Pada bahasa Kotlin, kita dapat melakukan pemeriksaan tipe data secara instan, formatnya adalah seperti kode berikut :

```
1 when(x) {
2   is String -> ...
3   is Int -> ...
4   is KelasSaya -> ...
5   else -> ...
6 }
```

Artinya nanti isi dari variabel `x` akan dipilah, apakah merupakan tipe data `String`, `Int`, merupakan instan dari kelas `KelasSaya`, atau berupa tipe data atau kelas lain.

Contoh nyata dari penggunaan kode di atas adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   val x: Any = 2
3
4   when (x) {
5     is String -> println("Jawaban String")
6     is Int -> println("Jawaban Int")
7     else -> println("lainnya")
8   }
9 }
```

Pada kode di atas, tipe data dari variabel `x` adalah `Any`, yang artinya bisa berupa tipe data apapun, atau instan dari kelas apapun. Lalu diisikan nilai awal berupa angka 2 (dua).

Selanjutnya kode akan melakukan seleksi tipe data `x` pada baris ke-4 dengan perintah `when (x)`, kemudian melakukan pemeriksaan, apabila tipe data dari variabel `x` adalah `String` maka akan dicetak seperti pada baris ke-5 dari kode di atas, tapi ternyata memang tipe data yang tepat adalah pada baris ke-6 sehingga program yang kita bangun akan mencetak `Jawaban Int` di layar, karena variabel `x` berisi angka 2 (dua).

1.3.5 Penggunaan *Range*

Penggunaan *range* biasanya untuk melakukan iterasi atau perulangan, beberapa contohnya akan dikerjakan dengan kode berikut :

```
1 for(i in 1..10) { println("data ke-$i = $i") }
```

Kode tersebut akan menghasilkan keluaran di monitor dimana nilai `i` dari 1 sampai dengan 10 sebagai berikut :

```
data ke-1 = 1
data ke-2 = 2
data ke-3 = 3
data ke-4 = 4
data ke-5 = 5
data ke-6 = 6
data ke-7 = 7
data ke-8 = 8
data ke-9 = 9
data ke-10 = 10
```

Contoh penggunaan *range* yang lain adalah seperti kode berikut :

```
for(i in 1 until 10) { println("data ke-$i = $i") }
```

Sama seperti kode sebelumnya, hanya saja kali ini nilai `i` adalah antara 1 sampai dengan 9, angka 10 tidak masuk dalam kualifikasi proses cetak ke monitor. Berikut adalah hasil keluarannya di monitor :

```
data ke-1 = 1
data ke-2 = 2
data ke-3 = 3
data ke-4 = 4
data ke-5 = 5
data ke-6 = 6
data ke-7 = 7
data ke-8 = 8
data ke-9 = 9
```

Contoh penggunaan *range* dengan beberapa pola lompatan atau kelipatan angka adalah sebagai berikut :

```
for(i in 1..10 step 3) { println("data ke-$i") }
```

Maksudnya adalah mencetak deretan angka yang dimulai dari 1 dengan kelipatan 3 sampai nilai `i` sama dengan 10. Berikut adalah hasil keluaran dari kode tersebut :

```
data ke-1
data ke-4
data ke-7
data ke-10
```

Contoh penggunaan *range* untuk perulangan yang mundur ke nilai yang lebih kecil adalah sebagai berikut :

```
for(i in 10 downTo 1) { println("data ke-$i") }
```

Penjelasan dari kode tersebut dijelaskan dengan hasil keluaran di layar monitor sebagai berikut :

```
data ke-10
data ke-9
data ke-8
data ke-7
data ke-6
data ke-5
data ke-4
data ke-3
data ke-2
data ke-1
```

1.3.6 *Read-only List*

Artinya adalah membuat *list* yang tidak dapat diubah isinya, contoh deklarasinya adalah sebagai berikut :

```
1 val list = listOf("data A", "data B", "data C")
```

Untuk lebih jelasnya, kita akan melihat kode contoh sebagai berikut :

```
1 fun main(args: Array<String>) {
2   val list = listOf("A", "B", "C");
3
4   for(i in list) {
5     println("data $i")
6   ]
7 }
```

Kode tersebut, pada baris ke-2 adalah menyiapkan objek *list* dan diisikan langsung dengan data menggunakan perintah `listOf`, selanjutnya seluruh data dicetak ke layar monitor sebagaimana tampilan berikut :

```
1 data A
2 data B
3 data C
```

1.3.7 *Read-only Map* dan Cara Mengaksesnya

Map sebetulnya adalah kelas koleksi yang berisi pasangan kunci (*key*) dan isi (*value*), contohnya adalah sebagai berikut :

```
1 val map = mapOf("key1" to "nilai1", "key2" to "nilai2", "key3" to "nilai3")
```

Untuk melakukan akses data pada *map* ini adalah sebagai berikut :

```
1 println(map["key"]) // untuk mengambil nilainya
2 map["key"] = nilai // untuk mengganti atau mengisi nilai
```

Kode lengkap untuk percobaan *Map* ini adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   val map = mapOf("key1" to "nilai1", "key2" to "nilai2", "key3" to "nilai3")
3
4   println(map["key2"])
5
6   for(i in map) {
7     println(i)
8   }
9 }
```

Hasil keluarannya adalah sebagai berikut :

```
1 nilai2
2 key1=nilai1
3 key2=nilai2
4 key3=nilai3
```

Hasil dari kode pada baris ke-4 adalah hasil keluaran pada baris ke-1, sedangkan hasil iterasi atau perulangan dari kode pada baris ke-6 sampai dengan baris ke-8 adalah pada baris ke-2 sampai dengan baris ke-4 pada hasil keluaran.

1.3.8 Jalan Pintas Perintah `if not null`

Ini digunakan sebetulnya untuk melakukan pemeriksaan terhadap isi dari suatu variabel apakah terisi atau tidak (*null*), kodenya adalah sebagai berikut :

```
1 import java.io.File
2
3 fun main(args: Array<String>) {
4   val file = File("Test.kt").listFiles()
5
6   println(file ?. getTotalSpace())
7 }
```

Pada kode tersebut kita melihat bahwa aplikasi yang kita bangun menggunakan pustaka Java yaitu `java.io.File`, hal ini memungkinkan karena memang Kotlin adalah turunan dari Java yang dapat menggunakan kelas-kelas Java untuk membangun aplikasi. Ini adalah salah fasilitas yang Kotlin sediakan.

Yang perlu diperhatikan adalah pada baris ke-6, yaitu pada bagian `file?.getTotalSpace()`, dimana tanda tanya (`?`) yang ada pada bagian ini menunjukan seleksi atau pemeriksaan terhadap kondisi *null*, atau kosongnya isi dari variabel yang diminta.

Bila isi variabel `file` ada isinya, maka akan dicetak ukuran dari *file* tersebut, namun bila nihil, maka akan mencetak `null` ke layar monitor.

Sebagai contoh apabila isi dari variabel `file` adalah `null` adalah sebagai berikut :

```
1 import java.io.File
2
3 fun main(args: Array<String>) {
4   val file: File?
5   file = null
6   println(file?.getTotalSpace())
7 }
```

Hasil dari kode di atas adalah informasi `null` yang dicetak ke layar monitor.

1.3.9 Jalan Pintas Perintah `if not null and else`

Cara ini sebetulnya pengembangan dari kasus sebelumnya, dengan penambahan penanganan apabila memang hasilnya adalah `null`, berikut kodenya :

```
1 import java.io.File
2
3 fun main(args: Array<String>) {
4   val file: File?
5   file = null
6   println(file ?. getTotalSpace() ?: "NIHIL")
7 }
```

Yang perlu diperhatikan adalah pada baris ke-6, dimana ada tambahan tanda (`?:`), apabila variabel `file` tidak bernilai `null` maka akan dipanggil *method* `getTotalSpace()`, namun bila bernilai `null` maka akan dicetak kata `NIHIL`.

1.3.10 Eksekusi Perintah `if null`

Kondisi ini digunakan untuk melakukan eksekusi bila nilai dari suatu variabel bernilai `null`. Contoh kode untuk kasus ini adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   val data = mapOf("nama" to "tamami", "umur" to 36)
3   val email = data["email"] ?: throw IllegalStateException("Email tidak ada")
4 }
```

Inti dari kode ini sebetulnya ada pada baris ke-3, pada saat variabel `email` akan diisikan sebuah nilai yang diambilkan dari variabel *map* `data` berupa *email*, namun bila nilai untuk *email* ini `null` maka akan dilemparkan keluar sebagai `IllegalStateException`.

Hasil keluaran dari kode tersebut tentu saja sebuah *exception* dengan pesan berupa `Email tidak ada`.

1.3.11 Eksekusi Perintah `if not null`

Kondisi ini adalah kebalikan dari eksekusi sebelumnya, yaitu bila nilainya tidak `null` maka akan melakukan eksekusi yang tertuang dalam blok kode. Berikut ada contoh kodenya :

```
1 fun main(args: Array<String>) {
2   val data = mapOf("nama" to "tamami", "umur" to 33)
3   data?.let {
4     println(data["nama"])
5   }
6 }
```

Blok kode untuk kondisi ini sebetulnya ada pada baris ke-3 sampai dengan baris ke-5, pada baris ke-3, akan dilakukan pemeriksaan, apakah variabel `data` berupa `null` atau bukan, bila bukan `null` maka akan dikerjakan perintah pada baris ke-4. Bila berupa `null` maka blok kode ini akan dilewati begitu saja.

1.3.12 Kembalikan Pada Perintah `when`

Pada bagian ini, sebuah *method* akan mengembalikan nilai dari hasil seleksi pada perintah `when`. Berikut kode lengkapnya :

```
1 fun main(args: Array<String>) {
2   println("Kode warna " + kodeWarna("hijau"))
3 }
4
5 fun kodeWarna(warna: String): Int {
6   return when(warna) {
7     "merah" -> 1
8     "hijau" -> 2
9     "biru" -> 3
10    else -> throw Exception("Salah Warna")
11  }
12 }
```

Pada bagian ini intinya ada pada *method* atau fungsi `kodeWarna`, yaitu bila parameter `warna` terisi oleh teks `merah` maka akan mengembalikan nilai `1`, bila terisi oleh teks `hijau` maka akan mengembalikan nilai `2` dan seterusnya.

Pada baris ke-2, karena dipanggil fungsi `kodeWarna` dengan parameter `warna` berisi teks `hijau`, maka hasil keluaran yang terjadi di layar monitor adalah sebagai berikut :

```
1 Kode warna 2
```

1.3.13 Ekspresi `try catch`

Pada bagian ini ditujukan untuk penanganan *exception* dengan `try catch` agar aplikasi yang dibangun lebih bersih dari informasi kesalahan yang diproduksi oleh bahasa pemrograman. Contoh kode untuk penjelasannya adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   try {
3     println("kode warna : " + kodeWarna("ungu"))
4   } catch(e: Exception) {
5     println("error : " + e.printStackTrace())
6   }
7 }
8
9 fun kodeWarna(warna: String): Int {
10  return when(warna) {
11    "merah" -> 1
12    "hijau" -> 2
13    "biru" -> 3
14    else -> throw Exception("Salah Warna")
15  }
16 }
```

Pada baris ke-3, sengaja diberikan isi teks `ungu` pada parameter fungsi `kodeWarna` agar *exception* yang ada dapat dipicu untuk dieksekusi.

Hasilnya adalah karena parameter teks `ungu` yang dikirim, maka yang diterima adalah *exception* pada baris ke-14, sehingga kode akan langsung dilempar ke baris ke-5 karena pemanggilan fungsi `kodeWarna` menghasilkan *exception* tersebut.

1.3.14 Ekspresi `if`

Pada bagian ini, sebetulnya berfungsi untuk kondisional biasa, untuk memilah sebagaimana bentuk *when* sebelumnya. Contoh kodenya adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   val i = 37
3
4   if(i % 2 == 0) {
5     println("itu bilangan genap")
6   } else {
7     println("itu bilangan ganjil")
8   }
9 }
```

Tentu saja hasil dari kode tersebut akan mencetak baris teks berikut :

```
1 itu bilangan ganjil
```

Karena memang nilai dari variabel `i` diperiksa, apabila setelah dibagi dengan angka 2 tidak memiliki sisa bagi, maka itu pertanda bahwa nilai `i` adalah bilangan genap, namun bila selain dari itu, akan dieksekusi perintah dalam blok `else`.

1.4 Adat

Pada tiap bahasa pemrograman, ada sebuah adat atau kebiasaan atau *style* dari kode agar tetap mudah terbaca bagi orang lain yang membuka kode sumbernya, begitu pula di Kotlin, ada beberapa kebiasan atau adat yang perlu kita latih agar kode kita mudah untuk dibaca oleh orang lain. Berikut adalah diantaranya :

- Gaya Penamaan

  Gaya penamaan akan mirip seperti bahasa Java, yaitu :

  - menggunalan `camelCase` untuk semua nama variabel atau fungsi (diusahakan menghilangkan garis bawah (_))
  - tipe atau kelas diawali dengan huruf besar
  - fungsi dan properti diawali dengan huruf kecil
  - menggunakan 4 spasi untuk indentasi
  - fungsi yang bersifat public harus memiliki dokumentasi seperti yang muncul pada dokumentasi Kotlin

- Titik Dua

  Penggunaan titik dua menggunakan spasi sebelum tanda titik dua ketika memisahkan tipe dan super-tipe, dan tidak menggunakan spasi sebelum tanda titik dua ketika memisahkan instan dan tipe. Berikut contohnya :

```
1 interface MyInterface<out T : Any> : Bar {
2   fun myFungsi(a: Int): T
3 }
```

- Lambda

  Ekspresi Lambda atau disebut labmda saja sebetulnya adalah fasilitas yang memudahkan penulis kode program untuk mempersingkat kode. Penulisannya seharusnya tiap penggunaan kurung kurawal buka dan tutup, disertakan spasi sebelum dan sesudahnya. Begitu lupa perlakukan untuk tanda panah yang memisahkan antara parameter dan isi lambda, harus diberikan spasi sebelum dan sesudahnya.

  Contoh sederhananya adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   val list = listOf(1, 2, 3, 4, 5)
3   println(list.filter { it >= 4 }
4 }
```

Inti dari kode diatas sebetulnya ada pada baris ke-3, pada bagian textttit ¿= 4, yang menghasilkan hanya isi yang nilainya lebih dari atau sama dengan 4. Hasil dari kode di atas adalah sebagai berikut :

```
1 [4, 5]
```

Untuk lambda sederhana tidak perlu melakukan definisi parameter, cukup menggunakan kata kunci `it`. Namun pada lambda yang memiliki parameter, harus selalu menyertakan parameter dan isinya. Seperti contoh kode berikut untuk lambda yang menyertakan parameternya :

```
1 fun main(args: Array<String>) {
2   val list = listOf(1, 2, 3, 4, 5)
3   println(list.map { data -> data * 2 } )
4 }
```

Inti dari kode di atas sebetulnya ada pada baris ke-3, pada bagian `data -> data * 2`, inilah bentuk lambda di Kotlin. Seluruh isi pada variabel `list` akan dikalikan dengan 2 dengan perintah tersebut, sehingga hasil keluarannya adalah seperti berikut :

```
1 [2, 4, 6, 8, 10]
```

- Format *Header* Kelas

  Deklarasi kelas dengan sedikit parameter dapat dituliskan dalam satu baris kode seperti berikut :

```
1 class Mahasiswa(nim: String, nama: String)
```

  Namun deklarasi kelas yang lebih banyak harus tersusun lebih rapih, sehingga setiap parameter atau argumen pembentuk kelas berada pada baris terpisah. Kemudian tutup kurung untuk parameter / argumen harus berada pada baris baru. Jika menggunakan pewarisan, kontruktor super-kelas harus ditempatkan pada baris yang sama dengan tutup kurung. Berikut adalah contoh kodenya :

```
1 class Pejabat (
2     nip: String,
3     nama: String,
4     alamat: String,
5     hp: String,
6     gaji: BigInteger
7 ) : Pegawai(nip, nama) {
8     ...
9 }
```

- Unit

  Bila sebuah fungsi mengembalikan nilai bertipe `Unit`, maka tidak perlu dituliskan, seperti contoh berikut :

```
1 fun fungsiKu() {
2     ...
3 }
```

Unit sendiri di Kotlin seperti tipe `void` di Java, yaitu tipe dengan hanya satu nilai saja.

- Fungsi vs. Properti

  Dalam beberapa hal, fungsi tanpa sebuah parameter mungkin terlihat sama dengan properti. Walau secara tertulis mirip, namun ada beberapa pertimbangan gaya untuk memilih salah satunya.

  Jadikanlah itu sebuah properti dan bukan fungsi bila :

  – tidak melamparkan eksepsi
  – kompleksitas rendah
  – perhitungan yang terjadi tidak berat
  – mengembalikan nilai yang sama setelah beberapa kali dipanggil

Demikianlah gaya penulisan kode pada Kotlin, walaupun bila tidak mengikuti aturan ini aplikasi tetap bisa di*compile* dan berjalan sebagaimana mestinya, tetapi ini adalah aturan baku untuk memudahkan sesama pemrogram bahasa Kotlin lebih cepat dan mudah dalam memahami kode yang ditulis oleh orang lain.

BAB 2

DASAR-DASAR

2.1 Tipe Data

Sebagaimana kebanyakan bahasa pemrograman, mengenali tipe data adalah hal yang penting untuk diketaui, karena perbedaan tipe data dapat menyebabkan perbedaan operasi yang dapat dilakukan kepadanya.

Tipe data secara garis besar dapat dikelompokan menjadi : angka, karakter, *boolean*, dan larik.

2.1.1 Angka

Tipe data untuk angka di Kotlin mirip dengan Java. Dan untuk karakter bukan dianggap sebagai angka di Kotlin. Berikut adalah tipe data angka yang dapat digunakan beserta ukurannya :

Tipe	Panjang Bit
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

- **Format Angka**

  Format angka pada Kotlin dapat mengakomodir beberapa format berikut :

  - Desimal, contohnya adalah `432`, untuk tipe data Long dituliskan sebagai `432L`.
  - Hexadesimal, contohnya adalah `0xa4`
  - Biner, contohnya adalah `0b00111100`

  Sebagai catatan bahwa tipe format oktal tidak didukung di Kotlin. Kotlin juga mendukung bilangan pecahan sebagai berikut :

  - Double, contohnya adalah 12.34
  - Float, contohnya adalah 154.3f

- **Garis Bawah Pada Format Angka**

  Kita dapat menggunakan garis bawah sebagai tanda pada angka yang kita isikan sebagai pengganti digit atau format lain. Ini didukung oleh Kotlin versi 1.1 ke atas, artinya bila masih menggunakan Kotlin versi sebelumnya, kode angka yang dibangun dengan garis bawah akan berantakan. Berikut contoh penulisan angka dengan garis bawah :

```
1 val satuJuta = 1_000_000
2 val telp = 0821_3828_3607
3
```

- **Konversi Eksplisit**

  Konversi eksplisit diperlukan karena tipe yang lebih kecil bukan berarti merupakan subtipe yang lebih besar, contohnya adalah seperti kode berikut :

```
1 val a: Int? = 1
2 val b: Long? = a
3 print(a == b)
4
```

  Hasil dari kode tersebut, pada saat kompilasi akan menghasilkan 2 (dua) kesalahan yaitu, yang pertama, variabel dengan tipe data `Int` tidak dapat dimasukkan langsung isinya ke dalam variabel dengan tipe data `Long`.

  Untuk tujuan ini, kita perlu melakukan konversi secara eksplisit dengan beberapa fungsi berikut, dan tiap variabel berjenis angka memilikinya :

  - `toByte()`: Byte
  - `toShort()`: Short
  - `toInt()`: Int
  - `toLong()`: Long
  - `toFloat()`: Float
  - `toDouble()`: Double
  - `toChar()`: Char

Contoh implementasi dari kode sebelumnya akan menjadi seperti ini :

```
1 val a: Int = 1
2 val b: Long? = a.toLong()
3 print(a.toLong() == b)
4
```

- **Operasi**

  Ada beberapa operasi, atau lebih dikenal sebagai fungsi, yang biasa dilakukan pada angka yang disediakan Kotlin, berikut adalah diantaranya :

  - `shl(bits)`, geser bit ke kiri
  - `shr(bits)`, geser bit ke kanan
  - `ushr(bits)`, geser bit tanpa tanda ke kanan
  - `and(bits)`, operator bit dan
  - `or(bits)`, operator bit atau
  - `xor(bits)`, operator bit *xor*
  - `inv()`, operator bit inversi

2.1.2 Karakter

Sama bahwa tipa data angka yang lebih kecil bukan berarti sub-tipe data angka yang lebih besar, berbeda dengan Java, dimana tipe karakter (`Char`) tidak dapat diperlakukan seperti tipe data angka. Berikut contoh kode yang tidak bisa dikompilasi :

```
1 fun main(args: Array<String>) {
2   val a: Char = 1
3   print(a)
4 }
```

Hasil kode di atas tidak dapat dikompilasi karena terdapat kesalahan / *error* pada baris ke-2.

Tipe data karakter harus memiliki tanda kutip tunggal sepasang, dan karakter yang berada di tengahnya. Contoh kodenya adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   val a: Char = '1'
3   print(a)
4 }
```

Pada kode di atas, variabel `a` bertipe data `Char` dengan diberikan nilai berupa karakter `1`. Kemudian mencetak isi variabel `a` ke layar monitor.

Karakter-karakter tertentu dapat dimunculkan dengan menggunakan *backslash*. Contoh kode untuk karakter-karakter tertentu atau spesial adalah sebagai berikut :

```
fun main(args: Array<String>) {
  val a: Char = '\t'
  val b: Char = '\b'
  val c: Char = '\n'
  val d: Char = '\r'
  val e: Char = '\''
  val f: Char = '\"'
  val g: Char = '\\'
  val h: Char = '\$'

  println("isi a : $a text")
  println("isi b : $b text")
  println("isi c : $c text")
  println("isi d : $d text")
  println("isi e : $e text")
  println("isi f : $f text")
  println("isi g : $g text")
  println("isi h : $h text")
}
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
isi a :    text
isi b : text
isi c :
 text
 text
isi e : ' text
isi f : " text
isi g : \ text
isi h : $ text
```

Seperti pada pembahasan di awal bahwa tipe data `Char` bukan merupakan sub-tipe dari tipe data `Int`, tetapi sebetulnya bisa saja di konversi secara eksplisit, berikut adalah contoh kodenya :

```
fun main(args: Array<String>) {
  val a: Int
  a = ubahKeDesimal('b')
  println("nilai a : $a")
}

fun ubahKeDesimal(c: Char): Int {
  return c.toInt()
}
```

Hasil dari kode tersebut adalah sebagai berikut :

```
nilai a : 98
```

2.1.3 *Boolean*

Tipe *boolean* adalah tipe dengan nilai hanya ada 2 (dua) pilihan, `true` dan `false`. Contoh kodenya adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   val a: Boolean = true
3   println("nilai a = $a")
4 }
```

Nilai yang diisikan ke variabel `a` yang bertipe data `Boolean` adalah `true`, yang kemudian apabila di *compile* dan di eksekusi akan menghasilkan keluaran sebagai berikut :

```
1 nilai a = true
```

2.1.4 Larik

Larik dalam bahasa pemrograman Kotlin berbentuk sebuah kelas `Array`. Untuk membuat larik, kita dapat memanggil fungsi `arrayOf()` atau `arrayOfNulls()`, contoh kodenya adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   val a: Array<Any>
3   a = arrayOf('a', 'c', 'i')
4
5   println("isi larik a : ")
6   for(i in a) {
7     print(" $i ")
8   }
9
10  println()
11  val b: Array<Any?>
12  b = arrayOfNulls(5)
13
14  println("isi larik b : ")
15  for(i in b) {
16    print(" $i ")
17  }
18 }
```

Kode tersebut pada baris ke-2 akan mendeklarasikan atau menyiapkan variabel `a` dengan bentuk `Array` yang mampu menampung tipe data apa pun, yang kebetulan pada baris ke-3 diisikan dengan data bertipe karakter. Baris ke-5 sampai dengan baris ke-8 digunakan untuk menampilkan isi dari larik `a`.

Pada baris ke-11, kita mendeklarasikan atau menyiapkan variabel `b` yang berbentuk larik dengan tipe data `Any` yang mampu menampung tipe data apapun, ada tanda berbentuk `?` setelahnya yang artinya data yang diisikan dapat berupa data `null`, karena memang pada baris ke-12 kita akan isikan larik `b` ini dengan 5 (lima) data `null`.

Kemudian baris ke-14 sampai dengan baris ke-17 digunakan untuk mencetak isi dari larik `b`.

Kita juga dapat membuat larik dengan menggunakan konstruktor-nya `Array`, yang menjadi parameter dari konstruktor ini adalah jumlah data dan nilai *default* yang akan diisikan. Contoh kodenya adalah seperti berikut ini :

```
1 fun main(args: Array<String>) {
2   val a: Array<Any>
3   a = Array(3, { _ -> '-' })
4
5   println("isi larik a :")
6   for(i in a) {
7     println(" $i ")
8   }
9 }
```

Pada baris ke-3, larik akan disiapkan untuk menampung 3 (tiga) data sesuai dengan parameter pertama dari konstruktor ini, dengan nilai *default* berupa tanda strip - sesuai parameter ke-2 dari konstruktor ini.

Kode dari parameter ke-2 yaitu `{ _ -> '-' }` nampak membingungkan, sebetulnya formatnya adalah `{ Int -> T }` dengan `Int` dapat digantikan variabel apapun yang nantinya akan menampung nilai *integer*, pada kode yang kita buat menggunakan tanda garis bawah (_) karena nilainya tidak dibutuhkan.

Kemudian ada huruf `T` disana yang dapat digantikan dengan data apapun, termasuk karakter yang kita masukan berupa tanda minus (-).

Kotlin juga menyediakan kelas larik untuk tipe data primitif tanpa harus mendeklarasikan dalam kurung siku, contohnya seperti kelas `ByteArray`, `ShortArray`, `IntArray`, dan sebagainya. Berikut adalah contoh kode untuk penerapan kelas larik tipe data primitif ini :

```
1 fun main(args: Array<String>) {
2   val a: IntArray
3   a = intArrayOf(3, 2, 7)
4
5   println("isi larik a:")
6   for(i in a) {
7     println(" $i ")
8   }
9 }
```

Pada baris ke-2 adalah persiapan variabel `a` dengan tipe data `IntArray`. Pada baris ke-3, variabel atau larik `a` diisikan dengan 3 (tiga) angka yaitu `3`, `2`, dan `7`.

Pada baris ke-5 sampai baris ke-8 adalah kode untuk menampilkan isi dari larik `a`.

2.1.5 *String*

Tipe data *string* digunakan untuk menampung 1 (satu) karakter atau lebih, yang sebenarnya seperti sebuah larik untuk menampung banyak karakter.

Penggunaan tipe data `String` ini sama seperti bahasa pemrograman lain, terutama Java, dapat langsung diisikan kata atau kalimat yang menjadi isinya seperti contoh berikut :

```
1 val kata = "Ini isi dari variabel kata"
```

Kode tersebut artinya memasukan kalimat `Ini isi dari variabel kata` ke variabel `kata`. Dalam *string* kita juga dapat menambahkan *escape character* seperti kode berikut :

```
1 val kata = "Ini isi dari variabel kata\n"
```

Kode tersebut nantinya akan menjadikan kursor pindah ke baris berikutnya.

Pada Kotlin, ada yang namanya *string* mentah, dimana pengetikan nilai *string* ini tidak perlu menggunakan *escape character*, dan setiap spasi atau baris baru akan dicetak sebagaimana adanya. berikut contoh kodenya :

```
1  fun main(args: Array<String>) {
2    val code = """
3      ini contoh raw string
4      spasi dan baris baru akan terbaca
5      sebagaimana mestinya
6
7        ini saat pindah baris baru
8    """
9
10   println(code)
11 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1          ini contoh raw string
2          spasi dan baris baru akan terbaca
3          sebagaimana mestinya
4
5            ini saat pindah baris baru
```

Terlihat bahwa hasil keluaran dari kode tersebut, mirip dengan kondisi di kodenya selain spasi kiri yang mungkin terlalu lebar. Kita juga dapat menggunakan *prefix* apapun sebagai tanda bahwa ini adalah awal dari pencetakan dengan deklarasi `trimMargin()`, contoh kodenya adalah sebagai berikut :

```
1  fun main(args: Array<String>) {
2      var kode = """
3          |   ini contoh raw string
4          |   spasi dan baris baru akan terbaca
5          |   sebagaimana mestinya
6
7          |     ini saat pindah baris baru
8      """.trimMargin("|")
9
10     println(kode)
11 }
```

Kita akan melihat perbedaan di baris ke-8, yaitu adanya `trimMargin("|")`, yang sebetulnya ini adalah fungsi milik *raw string* untuk melakukan pemotongan baris awal, sedangkan tanda garis lurus (`|`) adalah tanda dimulainya baris paling kiri, tanda awal baris ini pun dapat diganti sesuai keinginan kita.

Pada *string* di Kotlin, dapat menggunakan *template* dengan tanda `$` untuk menambahkan hasil dari ekspresi sebuah perintah ke dalam teks. Contoh kodenya adalah sebagai berikut :

```
1  fun main(args: Array<String>) {
2    val a = "tamami"
3
4    println("isi dari a adalah $a")
5    println("panjang dari a adalah ${a.length}")
6  }
```

Penggunaan *template* ada pada baris ke-4 dan baris ke-5. Dimana pada baris ke-4 menampilkan isi dari variabel `a` dengan *template* `$a`, dan pada baris ke-5 memanggil fungsi `length` milik variabel `a` yang bertipe *string*, pemanggilan fungsi `length` ini dilakukan dengan *template* `${a.length}`.

2.2 Paket

Pada Kotlin mendukung penamaan paket seperti di Java, hanya saja nama paket tidak terikat dengan nama direktori tempat *file* kode sumber berada, tetapi hasil dari kompilasi akan ditempatkan dalam direktori / *folder* sesuai dengan nama paket, bila direktori / *folder* belum ada, maka akan dibuatkan direktori / *folder* baru dengan nama sama dengan nama paket. Berikut contoh deklarasi paket pada kode sumber :

```
1 package tester
2
3 fun main(args: Array<String>) {
4   println("ini dicetak dalam paket tester")
5 }
```

Deklarasinya ada pada baris ke-1. Walaupun penempatan kode sumber tidak terikat dengan nama paket, namun lebih baik bila kode sumber tetap disimpan dalam direktori / *folder* sesuai dengan nama paket, agar memudahkan pencarian kelas.

Apabila suatu kelas akan menggunakan kelas lain dari paket yang lain, kita perlu mendeklarasikan dengan perintah `import`, contoh kodenya adalah sebagai berikut :

```
1 import java.util.Date
2
3 fun main(args: Array<String>) {
4   val a = Date()
5
6   println("isi data a adalah $a")
7 }
```

Pada kode tersebut kita menggunakan kelas milik Java, yaitu `java.util.Date` yang dideklarasikan pada baris ke-1. Kemudian di baris ke-4, variabel `a` akan diisikan dengan tanggal dan jam saat ini dengan cara pemanggilan kontruktor `Date`, lalu pada baris ke-6 mencetak hasil atau isi dari variabel `a` ke layar monitor.

2.3 Mengatur Alur

2.3.1 Ekspresi `if`

Ekspresi `if` di Kotlin mirip dengan Java, berikut adalah contoh kodenya :

```
1 fun main(args: Array<String>) {
2   val a = 2
3   val b = 5
4
5   if(a >= b)
6     println("a lebih besar dari b")
```

```
7   else
8     println("a lebih kecil dari b")
9 }
```

Hasil keluaran untuk kode di atas adalah sebagai berikut :

```
1 a lebih kecil dari b
```

Ini karena pada seleksi di baris ke-5 akan memeriksa, apakah `a` lebih besar atau sama dengan `b`, bila benar, maka akan dikerjakan dan dicetak perintah pada baris ke-6, bila salah, maka akan dikerjakan dan mencetak perintah pada baris ke-8.

Perintah `if` ini dapat pula dijadikan ekspresi seperti kode berikut :

```
1 fun main(args: Array<String>) {
2   val a = 2
3   val b = 5
4
5   val c = if(a > b) a else b
6   println("nilai c = $c")
7 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 nilai c = 5
```

Hasil seperti itu karena pada baris ke-5, nilai `a` akan diperiksa terlebih dahulu apakah lebih dari nilai `b`, bila benar, maka nilai `c` sama dengan `a`, bila tidak benar, maka nilai `c` sama dengan nilai `b`.

2.3.2 Ekspresi `when`

Jika bentuk `if` bertingkat terlalu rumit dan panjang untuk dideklarasikan, perintah atau bentuk `when` ini akan meringkasnya, formatnya adalah seperti contoh kode program berikut :

```
1 fun main(args: Array<String>) {
2   val a = 2
3
4   when(a) {
5     1 -> println("a = 1")
6     2 -> println("a = 2")
7     3 -> println("a = 3")
8     else -> {
9       println("lainnya")
10    }
11  }
12 }
```

Pada baris ke-4 akan memeriksa nilai dari variabel `a`, bila sesuai dengan opsi bahwa nilainya adalah `1`, maka akan dijalankan perintah pada baris ke-5, namun bila tidak cocok, maka akan diperiksa apakah nilainya adalah `2`, yang apabila cocok akan melakukan perintah pada baris ke-6, dan seterusnya sampai bila tidak ada pilihan yang cocok, maka akan menjalankan perintah pada blok `else`.

Itulah mengapa hasil keluaran dari kode program diatas apabila dijalankan akan menghasilkan teks berikut :

```
a = 2
```

Hebatnya, kita dapat melakukan kombinasi data dengan menggunakan ekspresi `when` ini, misalnya seperti kode program berikut :

```
1  fun main(args: Array<String>) {
2    val a = 2
3
4    when (a) {
5      1,3,5 -> println("a angka ganjil")
6      2,4,6 -> println("a angka genap")
7      else -> {
8        println("lainnya")
9      }
10   }
11 }
```

Pada baris ke-5 dan ke-6 adalah bentuk dari kombinasi deret angka yang akan dicocokan dengan variabel `a`. Hasil keluaran untuk kode program di atas adalah sebagai berikut :

```
a angka genap
```

Bukan hanya data yang statis seperti di atas untuk menyeleksi hasil variabel `a`, kita juga dapat menggunakan fungsi untuk seleksi seperti kode berikut :

```
1  fun main(args: Array<String>) {
2    val a = 2
3    val b = 5
4
5    when(a) {
6      varifikasi(b) -> println("hasil fungsi")
7      10 -> println("hasil statis")
8      else -> {
9        println("lainnya")
10     }
11   }
12 }
13
14 fun verifikasi(b: Int): Int {
15   if(b == 5) return 2 else return 0
16 }
```

Pada baris ke-6 terlihat bahwa ada fungsi dengan nama `verifikasi` dipanggil, isi dari variabel `b` yang bernilai `5` akan diteruskan ke fungsi ini dan melakukan pemeriksaan, bila nilai dari `b` sama dengan `5`, maka fungsi akan mengembalikan nilai `2`, bila tidak akan mengembalikan nilai `0` (nol).

Kenyataannya memang nilai dari variabel `b` adalah `5`, kemudian fungsi akan mengembalikan nilai `2` yang menjadi bahan periksa dari perintah `when` yang nilai `a`-nya adalah `2`, sehingga kode program akan mencetak keluaran sebagai berikut :

```
hasil fungsi
```

Kita juga dapat menggunakan perintah `in` untuk memeriksa nilai dari variabel yang diseleksi, berikut adalah contoh kode programnya :

```
1  fun main(args: Array<String>) {
2    val a = 'a'
```

```
3
4   when (a) {
5     in 'a'..'e' -> println("Kelas teladan")
6     in 'f'..'i' -> println("Kelas unggulan")
7     else -> {
8       println("lainnya")
9     }
10  }
11 }
```

Dari kode di atas sudah terlihat bahwa, karena variabel `a` berisi karakter `a`, maka baris yang akan dikerjakan ada pada baris ke-5. Sangat intuitif sekali kodenya bahwa pada baris ke-5 akan menyeleksi apabila isi dari variabel `a` ada di antara huruf `a, b, c, d` atau `e`, maka baris ke-5 inilah yang akan dikerjakan.

2.3.3 Ekspresi `for`

Ekspresi `for` digunakan untuk melakukan iterasi dari barisan data. Contoh kode programnya adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   val a = intArrayOf(1, 2, 3, 4)
3
4   for(i in a) {
5     println(i)
6   }
7 }
```

Pernyataan `for` ada pada baris ke-4 sampai ke-6. Yang maksudnya adalah mengisikan satu per satu nilai yang ada di dalam variabel `a` ke dalam variabel `i`, kemudian mencetak isi dari variabel `i`

2.3.4 Ekspresi `while`

Perintah `while` ini mirip seperti perintah `for`, hanya saja cara pengulangan yang dilakukan dengan `while` menggunakan kondisi. Contoh kodenya adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   var i = 1
3   while(i <= 5) {
4     println(i++)
5   }
6 }
```

Pada baris ke-2, nilai `i` mulai diinisialisasi dengan `1`, kemudian pada baris ke-3 sampai ke-5 sebetulnya adalah iterasi dimana nilai `i` akan selalu diperiksa apakah lebih kecil atau sama dengan `5`, bila benar makan akan dicetak nilai dari `i`, kemudian `i` akan ditambahkan dengan `1`. Begitu seterusnya sampai nilai `i` lebih dari `5` dan program selesai dieksekusi.

Hasil dari kode di atas adalah sebagai berikut :

```
1 1
2 2
3 3
```

```
4
5
```

Bentuk lain dari ekspresi `while` ini adalah `do..while`, contoh kodenya adalah sebagai berikut :

```
fun main(args: Array<String>) {
  var i = 1

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

Hasil keluarannya akan sama dengan kode sebelumnya, hanya saja seleksi dilakukan setelah iterasi pertama terjadi.

2.3.5 Ekspresi Loncat

Seperti Java, Kotlin pun memiliki 3 (tiga) ekspresi untuk melompat dari satu blok kode. Berikut adalah macamnya :

- `return`, yang bertugas mengembalikan nilai ke fungsi di atasnya.
- `break`, yang bertugas menghentikan proses iterasi.
- `continue`, yang bertugas untuk melanjutkan proses iterasi ke langkah berikutnya.

Berikut adalah contoh kode dari ketiga ekspresi tersebut :

```
fun main(args: Array<String>) {
  for(i in 1..5) {
    if(i == 2) continue
    if(i == 4) break
    println(i)
  }

  println(getAngka())
}

fun getAngka(): Int {
  return 10
}
```

Pada kode di atas, baris ke-2 bermaksud membuat iterasi dari angka `1` sampai dengan angka `5` yang nilainya disimpan dalam variabel `i`.

Pada baris ke-3, bila nilai `i` sama dengan `2`, maka iterasi akan diloncatkan ke nilai berikutnya tanpa melalui perintah `println(i)`.

Pada baris ke-4, bila nilai `i` sama dengan `4`, maka iterasi dihentikan dan diakhiri.

Kemudian pada baris ke-8, sebetulnya adalah pemanggilan terhadap fungsi `getAngka`, yang di dalam fungsi ini mengembalikan sebuah nilai yaitu angka `10` dengan perintah `return`.

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1
3
10
```

BAB 3

KELAS DAN OBJEK

3.1 Kelas

Kelas di Kotlin dideklarasikan dengan kata kunci `class`. Contoh dalam kodenya adalah sebagai berikut :

```
1 class Mahasiswa {
2 }
```

Deklarasi kelas memang sesederhana itu. Isi dari kelas itu sendiri terdiri dari konstruktor, properti atau yang biasa dikenal dengan istilah variabel, dan fungsi.

3.1.1 Konstruktor

Uniknya bentuk konstruktor dari Kotlin ini dibedakan menjadi 2 (dua), ada konstruktor utama, dan ada konstruktor tambahan. Kita bahas terlebih dahulu bagaimana bentuk dari konstruktor utama, contoh kode dasarnya adalah sebagai berikut :

```
1 class Mahasiswa constructor(nama: String) {
2 }
```

Jika konstruktor tidak memiliki anotasi atau *visibility modifiers*, maka kode di atas dapat disederhanakan menjadi seperti berikut :

```
1 class Mahasiswa(nama: String) {
2 }
```

Tentang apa itu anotasi dan *visibility modifiers* akan kita jelaskan di bab berikutnya.

Lalu bagaimana cara memanfaatkan parameter yang ada pada konstruktor bila deklarasi konstruktor implisit seperti itu? Ada 2 (dua) cara, yang pertama melalui blok `init`, yang kedua dengan langsung mengisikan ke variabel yang bersangkutan.

Berikut adalah contoh dari penggunaan blok `init` :

```
1  fun main(args: Array<String>) {
2    val mhs = Mahasiswa("tamami")
3
4    println(mhs.nama)
5  }
6
7  class Mahasiswa(nama: String) {
8    var nama: String
9
10   init {
11     this.nama = nama
12   }
13 }
```

Pada baris ke-2 dari kode di atas, variabel `mhs` bertipe kelas `Mahasiswa` yang langsung dipanggil konstruktornya dengan parameter berupa teks (*string*).

Kotlin akan memanggil konstruktor utamanya kemudian menjalankan blok `init` untuk pertama kalinya. Isi dari blok `init` ini hanya mengisikan variabel `nama` dari parameter konstruktor `nama`.

Kemudian aplikasi menlanjutkan tugasnya untuk mencetak variabel `nama` milik instan kelas `Mahasiswa`.

Atau kita juga bisa persingkat kode di atas menjadi seperti berikut :

```
1  fun main(args: Array<String>) {
2    val mhs = Mahasiswa("tamami")
3
4    println(mhs.nama)
5  }
6
7  class Mahasiswa(nama: String) {
8    var nama = nama
9  }
```

Hasil keluaran dari kode di atas sama persis dengan sebelumnya, seperti ini :

```
1 tamami
```

Kita telah menghapus blok `init` dan melewatkan nilai dari parameter `nama` langsung ke variabel `nama`.

Lalu bagaimana dengan konstruktor tambahan, deklarasi nya ada di dalam tubuh kelas itu sendiri, contoh kodenya adalah seperti berikut :

```
1 fun main(args: Array<String>) {
```

```
2   val mhs = Mahasiswa("tamami", "DIV–TI")
3
4   println(mhs.nama)
5   println(mhs.jurusan)
6 }
7
8 class Mahasiswa(nama: String) {
9   var nama = nama
10  var jurusan: String = ""
11
12  constructor(nama: String, jurusan: String): this(nama) {
13    this.jurusan = jurusan
14  }
15 }
```

Deklarasi konstruktor utama ada di baris ke-8 dengan satu parameter yaitu `nama`, sedangkan deklarasi konstruktor tambahan ada pada baris ke-12 sampai dengan baris ke-14. Dimana konstruktor tambahan memiliki 2 (dua) parameter, yaitu `nama` dan `jurusan`.

Ada satu tambahan lagi pada konstruktor tambahan, yaitu perintah `this` di akhir baris, ini karena Kotlin mengharuskan seluruh konstruktor tambahan memanggil konstruktor utama terlebih dahulu dengan perintah `this`.

Dalam sebuah kelas dapat memuat beberapa hal berikut :

- Konstruktor dan blok `init`
- Fungsi
- Properti (atau lebih dikenal dengan variabel)
- Kelas bersarang
- Deklarasi Objek.

3.1.2 Pewarisan

Sebetulnya seluruh kelas di Kotlin akan bermuara pada kelas `Any` sebagai super-kelas-nya. Bahkan kelas-kelas yang deklarasinya tanpa super-kelas akan menjadikan kelas `Any` ini sebagai *default*.

Untuk mendeklarasikan super kelas secara eksplisit, contoh kode berikut akan menjelaskannya :

```
1 fun main(args: Array<String>) {
2   val pegawai = Pejabat("tamami", "fungsional")
3
4   println(pegawai.nama)
5   println(pegawai.jabatan)
6 }
7
8 open class Pegawai(nama: String) {
9   var nama = nama
10 }
11
12 class Pejabat(nama: String, jabatan: String): Pegawai(nama) {
```

```
13   var jabatan = jabatan
14 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 tamami
2 fungsional
```

Pada kode di atas, yang menerangkan deklarasi super kelas secara eksplisit tepat pada baris ke-12, yang menunjukkan bahwa kelas `Pejabat` yang dibentuk adalah turunan dari kelas `Pegawai`.

Pada deklarasi kelas `Pegawai`, ada pernyataan `open` disana, ini sebetulnya menandakan bahwa kelas tersebut bukan bersifat final, karena secara *default*, semua kelas yang dibentuk di Kotlin akan bersifat final, maka agar kita dapat membuat turunan dari kelas yang telah kita buat, maka kelas tersebut harus kita berikan tanda `open` di awal deklarasi kelas.

Alur dari kode program di atas dapat diceritakan demikian, pertama pada baris ke-2 kita membuat sebuah variabel bernama `pegawai`, kemudian diisikan dengan data dari instan kelas `Pejabat`.

Kita coba melompat ke baris 12, dimana ini adalah tempat deklarasi pembentukan kelas `Pejabat` yang memang memiliki 2 (dua) parameter. Namun pada baris ke-12 inilah secara eksplisit menyebutkan bahwa kelas `Pejabat` adalah turunan dari kelas `Pegawai`. Namun parameter yang dimasukan ke kelas `Pegawai` adalah parameter yang juga masuk melalui konstruktor `Pejabat`, sehingga parameter yang dilewatkan ke konstruktor `Pegawai` adalah parameter yang juga dibawah oleh konstruktor `Pejabat`.

Pemanggilan variabel atau properti dari kelas `Pejabat` di baris ke-4 dan ke-5 sebetulnya tidak aneh karena sebetulnya, setelah kelas `Pejabat` menjadi turunan dari kelas `Pegawai`, maka semua variabel dan fungsi yang ada pada kelas `Pegawai` akan dimiliki oleh kelas `Pejabat`, sehingga variabel dari kelas `Pegawai` dapat pula diakses dari kelas `Pejabat`.

Contoh lain untuk pewarisan dengan konstruktor tambahan bisa dilihat pada kode berikut :

```
1  fun main(args: Array<String>) {
2    println("-= contoh pemanggilan konstruktor utama =-")
3    val pegawai = Pejabat("tamami", "fungsional")
4    println(pegawai.nama)
5    println(pegawai.jabatan)
6
7    println("\n-= contoh pemanggilan konstruktor tambahan =-")
8    val pegawaiLain = Pejabat("19840409001", "tamami", "fungsional")
9    println(pegawaiLain.nip)
10   println(pegawaiLain.nama)
11   println(pegawaiLain.jabatan)
12 }
13
14 open class Pegawai(nama: String) {
15   var nama = nama
16   var nip = ""
17
18   constructor(nip: String, nama: String): this(nama) {
19     this.nip = nip
20   }
21 }
```

```
22
23 class Pejabat: Pegawai {
24   var jabatan: String
25
26   constructor(nama: String, jabatan: String): super(nama) {
27     this.jabatan = jabatan
28   }
29
30   constructor(nip: String, nama: String, jabatan: String): super(nip, nama) {
31     this.jabatan = jabatan
32   }
33 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 -= contoh pemanggilan konstruktor utama =-
2 tamami
3 fungsional
4
5 -= contoh pemanggilan konstruktor tambahan =-
6 19840409001
7 tamami
8 fungsional
```

Pada kode di atas, kelas `Pegawai` memiliki 2 (dua) konstruktor, yang pertama adalah konstruktor utama dengan 1 (satu) buah parameter dengan nama `nama`, yang kedua adalah konstruktor dengan 2 (dua) parameter dengan nama `nip` dan `nama`.

Kita lihat bahwa pada konstruktor tambahan milik kelas `Pegawai` ada tambahan perintah `this` di belakang deklarasinya seperti yang telah dijelaskan pada saat cara mendeklarasikan konstruktor tambahan sebelumnya.

Kelas `Pejabat` merupakan turunan dari kelas `Pegawai` yang menggunakan 2 (dua) konstruktor dari kelas tersebut. Cara deklarasinya sama seperti konstruktor tambahan sebelumnya, hanya saja kali ini tidak menggunakan perintah `this` tetapi menggunakan kelas `super` karena yang dipanggil adalah super-kelas dari kelas `Pejabat` yaitu kelas `Pegawai`.

3.1.3 *Override* Fungsi

Fungsi dalam bahasa objek biasa dikenal dengan istilah *method*, mungkin akan digunakan secara bergantian istilah tersebut dalam buku ini yang artinya adalah sebetulnya sama.

Sama seperti kelas, untuk fungsi pun, agar dapat di *override*, suatu fungsi harus dideklarasikan secara eksplisit dengan perintah `open`. Contoh kode untuk *override* fungsi adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   println("-= contoh kelas Pegawai =-")
3   val pegawai = Pegawai("tamami")
4   pegawai.cetak()
5
6   println("\n-= contoh kelas Pejabat =-")
7   val pegawaiLain = Pejabat("tamami", "fungsional")
8   pegawaiLain.cetak()
9 }
10
```

```
11 open class Pegawai(nama: String) {
12   var nama = nama
13
14   open fun cetak() {
15     println("nama : $nama")
16   }
17 }
18
19 class Pejabat: Pegawai {
20   var jabatan: String
21
22   constructor(nama: String, jabatan: String): super(nama) {
23     this.jabatan = jabatan
24   }
25
26   override fun cetak() {
27     println("nama : $nama\njabatan : $jabatan");
28   }
29 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 -= contoh kelas Pegawai =-
2 nama : tamami
3
4 -= contoh kelas Pejabat =-
5 nama : tamami
6 jabatan : fungsional
```

Yang menjadi titik fokus, fungsi yang akan di-*override* harus diberikan perintah `open` seperti pada baris ke-14, dan pada saat akan meng-*override* pada kelas turunannya, harus diberikan perintah `override`.

Ini artinya fungsi `cetak` pada kelas `Pejabat` adalah fungsi turunan yang telah diadaptasikan sesuai kebutuhan kelas `Pejabat`, dan isi dari fungsi *cetak* pada kelas `Pegawai` telah digantikan dengan yang baru, yaitu khusus untuk kelas `Pejabat`.

Sebagai catatan lain adalah bahwa apabila ada fungsi yang dapat di-*override*maka deklarasi kelas juga harus memiliki perintah `open`, tidak boleh dalam kondisi final.

Fungsi-fungsi yang dideklarasikan secara `override`, akan otomatis menjadi `open` dan dapat di-*override* pada kelas turunannya, untuk mencegah hal ini, dapat diberikan perintah `final` diawalnya, agar tidak dapat di-*override* ulang.

Sebagai contoh, misalnya pada fungsi `cetak` milik kelas `Pejabat` tidak ingin agar kelas turunannya nanti melakukan *override* terhadap fungsi ini, maka deklarasi yang mungkin adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   println("-= contoh kelas Pegawai =-")
3   val pegawai = Pegawai("tamami")
4   pegawai.cetak()
5
6   println("-= contoh kelas Pejabat =-")
7   val pegawaiLain = Pejabat("tamami", "fungsional")
8   pegawaiLain.cetak()
9 }
10
```

```
11 open class Pegawai(nama: String) {
12   val nama = nama
13
14   open fun cetak() {
15     println("nama : $nama")
16   }
17 }
18
19 class Pejabat: Pegawai {
20   var jabatan: String
21
22   constructor(nama: String, jabatan: String): super(nama) {
23     this.jabatan = jabatan
24   }
25
26   final override fun cetak() {
27     println("nama : $nama\njabatan : $jabatan")
28   }
29 }
```

Hasil keluaran dari kode di atas sama saja dengan kode sebelumnya, hanya saja apabila kelas `Pejabat` memiliki turunan, maka turunannya tidak dapat melakukan *override* terhadap fungsi `cetak`, karena telah dilakukan kuncian dengan perintah `final` pada baris ke-26.

3.1.4 *Override* variabel

Variabel dalam bahasa objek biasa dikenal dengan istilah properti, jadi mungkin akan dibahas dalam buku ini bahwa properti adalah variabel milik kelas.

Override variabel atau properti ini sama seperti *override method*, ada penambahan perintah `open` pada properti yang dapat di-*override* oleh kelas turunan, dan memberikan tambahan perintah `override` pada properti di kelas turunannya. Kodenya akan terlihat seperti berikut ini :

```
1  fun main(args: Array<String>) {
2    println("-= contoh kelas Pegawai =-")
3    val pegawai = Pegawai()
4    pegawai.cetak()
5
6    println("\n-= contoh kelas Pejabat =-")
7    val pegawaiLain = Pejabat()
8    pegawaiLain.cetak()
9  }
10
11 open class Pegawai {
12   open var nama = "nama Pegawai"
13
14   fun cetak() {
15     println("nama : $nama")
16   }
17 }
18
19 class Pejabat: Pegawai() {
20   override var nama = "nama Pejabat"
21 }
```

Keluaran dari kode tersebut akan terlihat seperti ini :

```
-= contoh kelas Pegawai =-
nama : nama Pegawai

-= contoh kelas Pejabat =-
nama : nama Pejabat
```

Seperti dijelaskan sebelumnya, bahwa agar properti dapat di-*override*, maka perlu ditambahkan perintah `open` pada kelas utama seperti pada baris ke-12 dari kode di atas, kemudian menambahkan perintah `override` pada kelas turunannya seperti pada baris ke-20.

3.1.5 *Override rule*

Override rule ini terjadi karena kondisi dimana sebuah kelas diwajibkan melakukan *override* atas suatu *method* yang biasanya memiliki moyang 2 (dua) atau lebih kelas / *interface*. Contoh kodenya adalah sebagai berikut :

```
fun main(args: Array<String>) {
  println("-= contoh kelas Pegawai =-")
  val pegawai = Pegawai("tamami")
  pegawai.cetak()

  println("\n-= contoh kelas Pejabat =-"")
  val pejabat = Pejabat("tamami", "BPPKAD")
  pejabat.cetak()
}

open class Pegawai(nama: String) {
  var nama = nama

  open file cetak() {
    println("nama : $nama")
  }
}

interface Struktural {
  var unit: String

  fun cetak() {
    println("ini dicetak dari interface Struktural")
  }
}

class Pejabat(nama: String, unit: String): Pegawai(nama), Struktural {
  override var unit = unit

  override fun cetak() {
    println("nama : $nama\nunit : $unit")
  }
}
```

Hasil dari kode di atas adalah sebagai berikut :

```
-= contoh kelas Pegawai =-
nama : tamami
```

```
3
4 -= contoh kelas Pejabat =-
5 nama : tamami
6 unit : BPPKAD
```

Titik fokus untuk pembahasan kali ini ada pada baris ke-27, dimana kelas `Pejabat` mewarisi properti dan *method* dari 1 (satu) kelas yaitu `Pegawai` dan 1 (satu) *interface* `Struktural`.

Kelas dan *interface* tersebut memiliki 1 (satu) *method* yang sama dengan nama `cetak`. Karena hal inilah kelas `Pejabat` dengan terpaksa harus melakukan *override* untuk mendefinisikan bagaimana implementasi *method* `cetak` pada kelas ini.

3.2 Properti

Deklarasi sebuah properti di Kotlin ada 2 (dua) cara, yaitu properti yang dapat diubah, dan properti yang hanya dapat dibaca saja. Untuk properti yang dapat diubah, perlu dideklarasikan dengan perintah `var` sedangkan untuk property yang hanya dapat diberikan nilai sekali dan tidak dapat berubah menggunakan kata perintah `val`.

Contohnya sudah banyak kita lakukan di atas, kita akan coba lagi dengan menggunakan 2 (dua) perintah tersebut, `var` dan `val`, berikut kodenya :

```
1 fun main(args: Array<String>) {
2   var data = Pegawai()
3
4   data.nama = "p. tamami"
5
6   println(data.nama)
7   println(data.nip)
8 }
9
10 class Pegawai {
11   var nama = "tamami"
12   val nip = "19840409001"
13 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 p. tamami
2 19840409001
```

Hal ini karena setelah pembentukan instan kelas `Pegawai` di baris ke-2, properti `nama` diubah kembali pada baris ke-4, sehingga nama yang barulah yang keluar di layar.

Hal yang sama tidak dapat dilakukan terhadap properti `nip`, karena properti `nip` tidak dapat diubah nilainya.

Secara lengkap, sebuah deklarasi properti akan dituliskan seperti format kode berikut :

```
1 var <namaProperti>[: <tipeData>] [= <initializer>]
2     [<getter>]
3     [<setter>]
```

Penjelasannya adalah sebagai berikut :

- `<namaProperti>` ini nantinya digantikan dengan nama properti

- <tipeData> ini akan digantikan dengan tipe data / nama kelas
- <initializer> adalah data awal yang akan diisikan ke dalam properti
- <getter> adalah kode untuk mengambil nilai dari properti
- <setter> adalah blok kode untuk memberikan nilai ke properti

Terlihat agak rumit, namun sebetulnya sederhana, seperti, <getter> dan <setter> sebetulnya opsional, boleh hadir, boleh tidak. Saat <getter> dan <setter> tidak hadir, maka sebetulnya Kotlin akan membuatkan kedua fungsi tersebut secara umum.

<tipeData> pun sebetulnya bisa dijadikan implisit hanya dengan memberikan data awal pada bagian <initializer>. Mari kita coba contoh kode lebih lengkapnya untuk implementasi pembentukan properti seperti di atas, berikut adalah kodenya :

```
1  fun main(args: Array<String>) {
2    var data = Pegawai("tamami")
3
4    println(data.nama)
5  }
6
7  class Pegawai(nama: String) {
8    var nama: String = nama
9      get() = field
10     set(nama) {
11       field = nama
12     }
13 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 tamami
```

Kode yang kita maksud berada pada baris ke-8 sampai dengan baris ke-12, blok baris ini menerangkan detail bagaimana properti nama, apa tipe datanya, bagaimana cara mengisi nilainya dengan set, dan bagaimana mengambil datanya dengan get. Namun secara sederhana dengan tujuan yang sama, kode tersebut dapat diringkas menjadi seperti kode berikut :

```
1  fun main(args: Array<String>) {
2    var data = Pegawai("tamami")
3
4    println(data.nama)
5  }
6
7  class Pegawai(nama: String) {
8    var nama = nama
9  }
```

Keluaran dari kode di atas pun sama, dapat kita lihat bahwa baris yang tadi dapat digantikan hanya dengan 1 (satu) baris saja, yaitu pada baris ke-8.

Di Kotlin, ada juga yang namanya konstanta, atau lebih tepatnya *compile-time constants*, untuk menjadikan sebuah properti menjadi konstanta, diperlukan perintah const.

Lalu apa bedanya dengan val yang nilainya juga tidak dapat diubah? Perbedaannya adalah, pada penggunaan const, properti ini hanya dapat dideklarasikan langsung di bawah

kelas / objek, dan inisialisasinya hanya dapat dilakukan dengan nilai yang tipe datanya adalah `String` atau tipe data primitif.

Akan lebih jelas bila kita melihat kode program berikut sebagai contoh :

```
1  val pegawai = Pegawai()
2  val nama = pegawai.namaLengkap()
3
4  fun main(args: Array<String>) {
5    println(nama)
6  }
7
8  class Pegawai {
9    fun namaLengkap(): String {
10     return "p. tamami"
11   }
12 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 p. tamami
```

Pada penggunaan `val`, kita mungkin dapat menggunakan sebuah fungsi untuk mengisi nilainya seperti pada baris ke-2. Ini tidak mungkin dilakukan oleh properti dengan kata kunci `const`. Yang mungkin dilakukan apabila menggunakan kata kunci `const` adalah dengan kode berikut :

```
1  const val nama = "P. Tamami"
2
3  fun main(args: Array<String>) {
4    println(nama)
5  }
```

Hasil keluaran dari kode program di atas adalah sebagai berikut :

```
1 P. Tamami
```

Seperti pada penjelasan sebelumnya bahwa konstanta ini hanya dapat diberikan nilainya langsung dengan tipe data berupa String atau tipe data primitif lainnya, dan tidak dapat diubah pada saat *runtime*.

3.3 *Interface*

Interface disini bukan tampilan jendela atau tatap muka sebuah aplikasi, tetapi *interface* disini adalah bentuk implementasi dari orientasi objek.

Interface di Kotlin bukan hanya dapat menyiapkan kerangka berupa *method*, tetapi diperbolehkan melakukan implementasi di dalam *method* tersebut. Pada *interface* boleh terdapat properti / atribut hanya saja harus bersifat abstrak atau tanpa isi, karena implementasi isi akan dideklarasikan langsung dalam kelas turunannya.

Contoh penggunaan *interface* adalah seperti kode berikut ini :

```
1  fun main(args: Array<String>) {
2    var data = Pejabat("tamami")
3
```

```
4   println(data.getNamaLengkap())
5 }
6
7 interface Pegawai {
8   fun getNamaLengkap(): String
9 }
10
11 class Pejabat(nama: String): String {
12   var nama: String = nama
13
14   override fun getNamaLengkap(): String {
15     return "Pa/Bu $nama"
16   }
17 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 Pa/Bu tamami
```

Kita lihat bagaimana *interface* `Pegawai` pada baris ke-7 sampai dengan baris ke-9 membentuk sebuah kerangka implementasi tentang Pegawai dan kelas `Pejabat` melakukan pewarisan dan mengimplementasikan fungsi `getNamaLengkap()` seperti pada baris ke-14 sampai dengan baris ke-16.

3.4 *Visibility Modifiers*

Visibility Modifiers di Kotlin ada 4 (empat) macam, yaitu : `public`, `private`, `protected`, dan `internal`. Secara *default* bila tidak ada deklarasi lain, maka yang terpilih adalah `public`.

Penjelasannya adalah sebagai berikut :

- `public`, artinya deklarasi akan dapat diakses dari manapun
- `private`, artinya hanya sebatas file yang mendeklarasikan
- `internal`, artinya dapat diakses oleh objek-objek yang berada pada modul yang sama
- `protected` artinya sapa seperti `private`, tetapi dapat terlihat pada kelas turunannya.

Penjelasan untuk `public` tidak perlu kita lakukan kembali, karena dari contoh-contoh sebelumnya kita sudah mendapati lingkup `public` itu demikian, dapat diakses dari manapun, dan secara *default* deklarasi tiap properti atau fungsi akan berada pada lingkup `public`.

Untuk `private`, contoh kodenya adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   var data = Pegawai("tamami")
3   println(data.nama)
4 }
5
6 class Pegawai(nama: String) {
7   private var nama: String = nama
8 }
```

Kode di atas tidak akan pernah bisa di-*compile* karena kesalahan bahwa pada baris ke-7, deklarasi properti `nama` memiliki lingkup `private` sehingga perintah pada baris ke-3 tidak akan pernah dapat dieksekusi.

Sekarang kita coba untuk implementasi dengan kata kunci `internal`, buatlah 2 (dua) buah *file*, isi dari *file* kode sumber pertama adalah sebagai berikut :

```
1 public class Pegawai(nama: String) {
2   internal var nama = nama
3 }
```

Lalu *file* yang kedua, isi kodenya adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   var data = Pegawai("tamami")
3   println(data.nama)
4 }
```

Yang pertama dilakukan kompilasi adalah *file* dengan nama kelas `Pegawai`, misal saya beri nama `Pegawai.kt`, lalu coba *compile* dengan perintah berikut :

```
1 kotlinc Pegawai.kt
```

Namun pada saat *compile file* yang ke-2, karena saya beri nama *file* yang kedua dengan `Test.kt`, lakukan seperti kode / perintah berikut :

```
1 kotlinc -cp . Test.kt
```

Saat *compile* akan muncul peringatan kesalahan bahwa deklarasi `nama` pada kelas `Pegawai` bersifat `internal` sehingga tidak dapat dilakukan pemanggilan di *method* `main`.

Untuk percobaan `protected`, perhatikan kode berikut ini :

```
1  fun main(args: Array<String>) {
2    var data = Pejabat("tamami")
3    println(data.getNamaLengkap())
4  }
5
6  open class Pegawai(nama: String) {
7    protected var nama = nama
8  }
9
10 class Pejabat(nama: String): Pegawai(nama) {
11   fun getNamaLengkap(): String {
12     return "pa/bu $nama"
13   }
14 }
```

Pada baris ke-7 kode di atas, ada deklarasi `protected`, artinya pada baris ke-3 kita tidak dapat memanggil langsung dengan perintah `data.nama`.

Kemudian di baris ke-10 sampai dengan ke-14, ada deklarasi kelas `Pejabat` yang mewarisi atribut dan fungsi dari kelas `Pegawai`, yang diwarisi tentu saja adalah atribut `nama`, maka dari itu, pemanggilan atribut `nama` masih dapat dilakukan pada kelas `Pejabat` tanpa harus mendeklarasikan atribut ini karena atribut `nama` ini adalah hasil dari pewarisan kelas `Pegawai`.

3.5 Ekstensi Fungsi

Kotlin memeberikan kita sebuah fasilitas ekstensi fungsi. Artinya tanpa harus melakukan pewarisan, kita dapat menambahkan beberapa fungsi di luar deklarasi kelas. Mari kita lihat contoh kode berikut :

```
1  fun main(args: Array<String>) {
2    var data = Pegawai("tamami")
3    data.cetakNama()
4  }
5
6  fun Pegawai.cetakNama() {
7    println("pa/bu ${this.nama}")
8  }
9
10 open class Pegawai(nama: String) {
11   var nama = nama
12 }
```

Hasil dari kode di atas adalah seperti berikut ini :

```
1 pa/bu tamami
```

Kita lihat pada blok deklarasi kelas `Pegawai` pada baris ke-10 sampai dengan baris ke-12, tidak ada fungsi dengan nama `cetakNama()`, tambahan fungsi tersebut muncul di luar kelas, yaitu pada baris ke-6 sampai dengan baris ke-8, sehingga memungkinkan perintah pada baris ke-3 dilaksanakan.

Kondisi ekstensi fungsi ini sebetulnya tidak menambahkan anggota baru ke dalam kelas, tetapi memungkinkan sebuah variabel untuk dipanggil dengan fungsi yang belum ada pada dirinya. Untuk membuktikan bahwa ekstensi fungsi tidak sama dengan fungsi kelas yang asli, perhatikan kode berikut :

```
1  fun main(args: Array<String>) {
2    var pejabat = Pejabat()
3
4    cetakIsinya(pejabat)
5  }
6
7  fun Pegawai.cetak() = "ini kelar Pegawai"
8
9  fun Pejabat.cetak() = "ini kelas Pejabat"
10
11 fun cetakIsinya(pegawai: Pegawai) {
12   println(pegawai.cetak())
13 }
14
15 open class Pegawai
16
17 class Pejabat: Pegawai()
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 ini kelas Pegawai
```

Terlihat bahwa pada saat dilewatkan instan dari kelas `Pejabat` pada baris ke-4, seharusnya yang tercetak adalah `ini kelas Pejabat`, namun yang keluar justru seba-

liknya, ini karena yang dipanggil dalam fungsi `cetakIsinya` adalah extensi fungsi `cetak` yang dideklarasikan menempel pada kelas `Pegawai`, sehingga walaupun sebenarnya kelas `Pejabat` mewarisi fungsi dan *method* dari kelas Pegawai, tapi perlakuannya bahwa fungsi `cetak` tidak ikut diwariskan, melainkan dipanggil fungsi ekstensi aslinya, yaitu `Pegawai.cetak()`.

Jika kita ingin mengakses ekstensi fungsi dari luar paket, caranya adalah dengan melakukan `import`, contohnya dapat kita lihat pada kode berikut dengan 2 (dua) *file* yang berbeda paket.

File pertama akan mendeklarasikan kelas `Pegawai`, yang berada di paket `entity`, berikut adalah kodenya :

```
1 package entity
2
3 public class Pegawai() {
4   internal var nama: String = ""
5 }
6
7 fun Pegawai.cetakInfo() {
8   println("ini cetakInfo dari Pegawai")
9 }
```

Setelah file tersebut di*compile*, *file* hasil kompilasi akan berada pada *folder* `entity` dengan 2 (dua) *file* berekstensi `.class`.

Hasil dari 2 (dua) *file* tersebut bila nama *file*-nya adalah `Pegawai.kt`, akan menjadi `PegawaiKt.class` dan `Pegawai.class`. Sebetulnya *file* `Pegawai.class` berisi kelas `Pegawai`, sedangkan *file* `PegawaiKt.class` berisi fungsi yang dideklarasikan di luar kelas `Pegawai` namun masih dalam satu *file* `Pegawai.kt`.

Lalu *file* yang ke-2 isinya adalah sebagai berikut :

```
1 import entity.Pegawai
2 import entity.cetakInfo
3
4 fun main(args Array<String>) {
5   var pegawai = Pegawai()
6   pegawai.cetakInfo()
7 }
```

Pada saat melakukan *compile* terhadap *file* yang ke-2, anggaplah kita berikan *file* ini nama `Test.kt`. Untuk memberikan petunjuk bagi Kotlin bahwa pustaka yang akan kita gunakan (kelas `Pegawai`) berada pada paket `entity`, maka kita arahkan Kotlin agar mengerti dengan opsi perintah `classpath`. Berikut perintah lengkap yang dapat dieksekusi :

```
1 > kotlinc -cp . Test.kt
```

Pada saat menjalankan aplikasi tersebut, maka hasil keluaran yang tampak di layar adalah seperti berikut ini :

```
1 ini cetakInfo dari Pegawai
```

Yang perlu di perhatikan adalah pada *file* `Test.kt` pada baris ke-2. Untuk dapat menggunakan ekstensi fungsi yang berada di luar paket, maka diperlukan deklarasi `import` seperti pada baris ke-2.

Bagaimana deklarasi untuk ekstensi fungsi apabila berada di dalam kelas yang lain, bukan langsung di bawah paket tertentu. Untuk memahami ini, kita akan coba awali dari kode berikut :

```
1  fun main(args: Array<String>) {
2    Pelanggan.eksekusi(Barang())
3  }
4
5  class Barang {
6    fun cetakInfo() {
7      println("ini dari kelas Barang")
8    }
9  }
10
11 class Pelanggan {
12   fun cetakInfo() {
13     println("ini dari kelas Pelanggan")
14   }
15
16   fun Barang.cetakAll() {
17     cetakInfo()
18     this@Pelanggan.cetakInfo()
19   }
20
21   fun eksekusi(barang: Barang) {
22     barang.cetakAll()
23   }
24 }
```

Hasil dari kode program di atas adalah sebagai berikut :

```
1 ini dari kelas Barang
2 ini dari kelas Pelanggan
```

Alur programnya sebetulnya adalah seperti ini, pada saat pemanggilan fungsi `eksekusi` milik kelas `Pelanggan` di baris ke-2, program akan loncat ke baris 21 dan memanggil ekstensi fungsi `cetakAll` milik kelas `Barang` yang berada di dalam kelas `Pelanggan` di baris ke-16.

Pada saat eksekusi baris ke-17, yang dipanggil adalah fungsi `cetakInfo` milik kelas `Barang` yang berada di baris ke-6, sehingga tercetaklah lebih dahulu teks `ini dari kelas Barang`.

Yang terakhir karena fungsi `cetakInfo` milik kelas `Pelanggan` dan kelas `Barang` memiliki bentuk yang sama, maka pada baris ke-18 diperjelas dengan `this@Pelanggan`, sehingga tercetaklah seperti pada baris ke-2 hasil keluaran program.

Bila fungsi yang dipanggil memiliki nama atau bentuk yang berbeda, tidak perlu ditegaskan dengan perintah `this@Pelanggan`.

3.6 Kelas Data

Ada saatnya dimana kita membuat sebuah kelas dengan tujuan spesifik yaitu menjadi tempat simpanan data. Di Kotlin, kelas ini ditandai dengan perintah `data`, berikut contohnya :

```
1 data class Pegawai(var nip: String, var nama: String, var gaji: Int)
```

Kelas di atas sebetulnya akan membuat beberapa fungsi secara otomatis seperti berikut :

- Fungsi `equals` yang fungsinya untuk melakukan pemeriksaan apakah kedua objek yang dibandingkan sama isinya.
- Fungsi `hashCode` yang fungsinya untuk menghasilkan kode *hash* yang memastikan bahwa tiap objek yang terbentuk bersifat unik.
- Fungsi `toString` yang akan mencetak nama kelas beserta parameternya.
- Fungsi `componentN` sesuai jumlah parameter yang hadir di konstruktor.
- Fungsi `copy` yang akan memberikan duplikasi data untuk instan kelas lain yang terbentuk.

Coba kita buktikan, apakah benar semua fungsi itu terbentuk secara otomatis, seharusnya saat fungsi tersebut terbentuk otomatis, dapat langsung dipanggil pada instan kelas yang terbentuk, berikut kode lengkap pembuktiannya :

```
1  data class Pegawai(var nip: String, var nama: String, var gaji: Int)
2
3  fun main(args: Array<String>) {
4    var pegawai = Pegawai("1984001", "tamami", 3000)
5
6    println("hashCode = ${pegawai.hashCode()}")
7    println("toString = ${pegawai.toString()}")
8    println("comp1 = ${pegawai.component1()}")
9    println("comp2 = ${pegawai.component2()}")
10   println("comp3 = ${pegawai.component3()}")
11
12   var pegawai2 = pegawai.copy(nip = "1984002", nama = "ami")
13   println("data pegawai2 = ${pegawai2.toString()}")
14
15   var pegawaiSama = pegawai.copy()
16   println("pegawai sama dengan pegawaiSama ga? ${pegawai.equals(pegawaiSama)
       }")
17   println("pegawai sama dengan pegawai2 ga? ${pegawai.equals(pegawai2)}")
18 }
```

Hasil dari kode di atas adalah sebagai berikut :

```
1 hashCode = 337043080
2 toString = Pegawai(nip=1984001, nama=tamami, gaji=3000)
3 comp1 = 1984001
4 comp2 = tamami
5 comp3 = 3000
6 data pegawai2 = Pegawai(nip=1984002, nama=ami, gaji=3000)
7 pegawai sama dengan pegawai pegawaiSama ga? true
8 pegawai sama dengan pegawai pegawai2 ga? false
```

Ternyata terbukti bahwa semua fungsi di atas terbentuk otomatis pada kelas data. Yang menarik adalah pada baris ke-7 dan ke-8 dari hasil keluaran, terlihat bahwa apabila data dari tiap parameter sama persis, maka 2 (dua) kelas yang dibandingkan akan dianggap sama karena isinya sama.

Adat yang biasanya terjadi untuk menjaga konsistensi kode program terkait kelas data ini adalah sebagai berikut :

- Konstruktor utama harus menyediakan paling sedikit 1 (satu) parameter
- Parameter yang hadir di konstruktor harus diberi tanda `var` atau `val`
- Kelas data tidak boleh memiliki kata kunci `abstract`, `open`, `sealed`, atau `inner`.

3.7 Kelas Tertutup

Penggunaan kelas tertutup ini sangat spesifik, yaitu membantu kita melakukan seleksi `when` dengan lebih sempurna. Artinya, pada saat aplikasi melakukan seleksi, jangan sampai ada 1 (satu) opsi atau lebih yang tertinggal tanpa sebuah pemeriksaan.

Akan lebih jelas apabila kita melihat contoh kodenya seperti berikut :

```
1  fun main(args: Array<String>) {
2    val angka1 = Positif(4)
3    val angka2 = Negatif(-20)
4    val angka3 = Nihil()
5
6    print("angka1 : ")
7    println(cek(angka1))
8
9    print("angka2 : ")
10   println(cek(angka2))
11
12   print("angka3 : ")
13   println(cek(angka3))
14 }
15
16 fun cek(data: Cek): String = when(data) {
17   is Positif -> "${data.angka} : bilangan positif"
18   is Negatif -> "${data.angka} : bilangan negatif"
19   is Nihil -> "0"
20 }
21
22 sealed class Cek
23
24 class Positif(val angka: Int): Cek()
25
26 class Negatif(val angka: Int): Cek()
27
28 class Nihil: Cek()
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 angka1 : 4 : bilangan positif
2 angka2 : -20 : bilangan negatif
3 angka3 : 0
```

Terlihat bahwa deklarasi *sealed class* `Cek` ada di baris ke-22, diikuti dengan deklarasi sub-kelas dari `Cek`. Ada 3 (tiga) kelas yang mewarisi kelas `Cek` ini.

Fungsi dari `sealed class` secara spesifik akan bermanfaat bila dihubungkan dengan deklarasi `when` pada baris ke-16.

Kita akan coba melihat alur dari kode program yang telah kita buat, pertama, kita siapkan beberapa variabel / properti / atribut seperti terlihat di baris ke-2 sampai dengan baris ke-4.

Kemudian melakukan pencetakan ke layar monitor dengan memanggil fungsi `cek` untuk masing-masing variabel yang sudah disiapkan, seperti terlihat di baris ke-7, baris ke-10, dan baris ke-13.

Kekuatan `sealed class` sebetulnya ada di sini, yaitu dibaris ke-16, dengan adanya perintah `when` dengan variabel `data` yang menjadi objek seleksi, maka akan diperiksa tipe data dari variabel `data` ini terhadap seluruh sub-kelas dari `sealed class Cek`, apakah `Positif`, `Negatif`, atau `Nihil`.

Apabila salah satu baris dihilangkan di antara baris ke-17 sampai dengan baris ke-19, maka saat *compile* akan muncul peringatan bahwa masih ada perintah seleksi yang tertinggal, harap deklarasikan atau ganti dengan deklarasi `else`.

Misal, pada baris ke-19, kita hapus, maka akan ada peringatan bahwa perintah `is Nihil` tidak ada dan harus dideklarasikan.

3.8 Generik

Seperti bahasa pemrograman Java, Kotlin pun memiliki fasilitas generik yang memungkinkan deklarasi atas tipe data dibuat umum. Biasanya akan kita temukan di pustaka *collection*. Coba perhatikan contoh kode berikut untuk implementasi generik :

```
1  import java.util.ArrayList
2
3  fun main(args: Array<String>) {
4    var data = Data(2)
5    data.addData(5)
6
7    data.cetak()
8  }
9
10 class Data<T>(t: T) {
11   var larik = ArrayList<T>()
12
13   init {
14     larik.add(t)
15   }
16
17   fun addData(t: T) {
18     larik.add(t)
19   }
20
21   fun cetak() {
22     for(data in larik) {
23       println(data)
24     }
25   }
26 }
```

Hasil keluaran dari kode program di atas adalah sebagai berikut :

```
1 2
2 5
```

Deklarasi generik sebetulnya ada pada baris ke-10, yaitu dengan tanda seperti ini `<T>`, yang artinya, kelas `Data` yang kita buat dapat menerima tipe data apapun untuk disimpan dalam properti `larik` yang sudah disediakan.

Pada contoh kita di atas, kita menggunakan kelas `Data` dengan tipe data `Int` sebagai pengganti `T`. Sehingga `ArrayList` yang dideklarasikan pada baris ke-11 akan digunakan hanya untuk tipe data `Int`.

3.9 Kelas Bersarang

Artinya di dalam sebuah kelas, bisa diperbolehkan kita membentuk kelas lain, dimana kelas yang berada di dalam ini dapat mengakses variabel yang berada pada kelas di atasnya. Contoh kodenya adalah sebagai berikut :

```
1  fun main(args: Array<String>) {
2    val data = Pegawai().Jabatan()
3
4    data.cetak()
5  }
6
7  class Pegawai {
8    var nama = "tamami"
9
10   inner class Jabatan {
11     fun cetak() {
12       println("$nama sebagai direktur")
13     }
14   }
15 }
```

Hasil keluaran dari kode di atas akan terlihat seperti berikut ini :

```
1 tamami sebagai Direktur
```

Kita lihat bahwa kelas `Jabatan` bersarang atau berada di dalam kelas `Pegawai` pada baris ke-10 dengan kode `inner` di depannya yang memungkinkan di dalam kelas `Jabatan` memanggil properti milik kelas induknya. Apabila kata kunci `inner` kita hilangkan, maka pemanggilan variabel `$nama` pada baris ke-12 tidak akan mengalami kegagalan kompilasi.

Ada saatnya sebuah kelas dibentuk tanpa memerlukan nama instan seperti di Java, istilahnya biasa dikenal dengan *anonymous class*, di Kotlin implementasinya seperti kode berikut :

```
1  fun main(args: Array<String>) {
2    var data = Pegawai(object: Jabatan() {
3      init {
4        jabatan = "direktur"
5      }
6    })
7
8    println(data.jabatan.jabatan)
9  }
10
11 class Pegawai(jabatan: Jabatan) {
```

```
12   var jabatan = jabatan
13 }
14
15 open class Jabatan {
16   var jabatan = ""
17 }
```

Hasil dari kode di atas akan terlihat seperti berikut ini :

```
1 direktur
```

Deklarasi dari *anonymous class* pada kode di atas ada pada baris ke-2 dengan kata kunci `object`. Kelas Jabatan sendiri harus di deklarasikan dengan kata kunci `open` seperti pada baris ke-15 agar kelas anonim yang terbentuk di baris ke-2 dapat diubah sesuai kebutuhan. Hasilnya adalah mengisikan variabel `jabatan` dengan teks `direktur`.

3.10 Kelas *Enum*

Kelas *enum* sendiri sebetulnya ditujukan untuk membentuk definisi data baru yang bisa disamakan dengan konstanta. Kita lihat contoh kode berikut untuk lebih jelasnya :

```
1 fun main(args: Array<String>) {
2   var jenisKelamin = Kelamin.PRIA
3
4   println(jenisKelamin)
5 }
6
7 enum class Kelamin {
8   PRIA, WANITA
9 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 PRIA
```

Hasil keluaran yang ditampilkan bukan berupa `String`, tetapi berupa nama *enum*nya, masing-masing konstanta *enum* dipisahkan menggunakan tanda koma.

Sebetulnya tiap konstanta dalam kelas *enum* adalah sebuah objek atau lebih tepatnya adalah instan dari kelas *enum*nya, dan tiap objek ini dapat dilakukan inisialisasi. Berikut contoh kodenya :

```
1 fun main(args: Array<String>) {
2   var angka = Angka.SEPULUH
3
4   println(angka)
5   when(Angka.SATU.nilai >= Angka.SEPULUH.nilai) {
6     true -> println("Satu lebih besar dari Sepuluh")
7     false -> println("Sepuluh lebih besar dari Satu")
8   }
9 }
10
11 enum class Angka(val nilai: Int) {
12   SATU(1),
13   SEPULUH(10),
14   SERATUS(100)
15 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 SEPULUH
2 Sepuluh lebih besar dari Satu
```

Pada baris ke-2, kita melihat bahwa variabel `angka` diisikan dengan data *enum* berupa `Angka.SEPULUH`, kemudian mencetaknya pada baris ke-4.

Pada baris ke-5, kita mencoba melakukan perbandingan, apakah data *enum* `SATU` lebih besar atau sama dengan data *enum* `SEPULUH`, namun ternyata hasilnya adalah `false` sehingga yang tercetak ke layar adalah perintah `println` pada baris ke-7.

Perhatikan pada baris ke-5 bahwa yang dilakukan perbandingan adalah parameter `nilai` yang ada pada tiap-tiap *enum* konstantanya.

Satu lagi dari *enum* di Kotlin adalah bahwa tiap *enum* konstantanya memiliki properti yang memberikan kita informasi nama (`name`) dan posisi (`ordinal`) urutan deklarasinya dalam kelas *enum*. Perhatikan kode berikut untuk memperjelasnya :

```
1  fun main(args: Array<String>) {
2    val data = Angka.SEPULUH
3
4    println("namanya ${data.name}")
5    println("urutan ${data.ordinal}")
6  }
7
8  enum class Angka(val nilai: Int) {
9    SATU(1),
10   SEPULUH(10),
11   SERATUS(100)
12 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 namanya SEPULUH
2 urutan 1
```

Kita lihat bahwa nama dari *enum* konstanta `SEPULUH` adalah `SEPULUH` juga, sedangkan urutan deklarasinya adalah `1` (satu), dimana data urutan awal yang digunakan dalam *enum* ini adalah 0 (nol).

3.11 Objek Ekspresi dan Deklarasi

Ada kalanya saat kita ingin menggunakan dan mengubah sebuah kelas, namun tidak ingin dengan cara membuat sub-kelas baru, melainkan melakukannya dengan cara membuat kelas anonim, saat inilah objek ekspresi dan objek deklarasi dapat digunakan. Coba perhatikan kode berikut ini :

```
1  fun main(args: Array<String>) {
2    var data = Pegawai(object : Bio() {
3      override fun cetakNama() {
4        println("nama : $nama")
5        println("hp : $hp")
6      }
7    })
8
```

```
9   data.bio.cetakNama()
10 }
11
12 class Pegawai(bio: Bio) {
13   var bio = bio
14 }
15
16 abstract class Bio {
17   val nama = "tamami"
18   val hp = "08123456"
19
20   abstract fun cetakNama()
21 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 nama : tamami
2 hp : 08123456
```

Deklarasi objek ekspresi ada pada baris ke-2, yaitu membuat instan dari kelas `Bio` dengan langsung merubah implementasinya sekaligus tanpa membuat sub-kelas terlebih dahulu.

Untuk objek deklarasi, penjelasannya kita awali dari kode berikut :

```
1 fun main(args: Array<String>) {
2   println(Rumus.tambah(2,5))
3 }
4
5 object Rumus {
6   fun tambah(a: Int, b: Int): Int {
7     return a + b
8   }
9 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 7
```

Object deklarasi ada pada baris ke-5 sampai dengan baris ke-9, cara memanggil objek deklarasi ini mirip dengan pemanggilan tipe `static` di Java, langsung memanggil nama objek dan nama fungsi yang dibutuhkan tanpa harus membuat instan dari kelasnya.

Untuk penggunaan objek deklarasi di dalam kelas, harus ditambahkan kata kunci `companion`. Berikut adalah contoh kodenya :

```
1 fun main(args: Array<String>) {
2   println(Pegawai.ambilSandi(2,3))
3 }
4
5 class Pegawai {
6   companion object Sandi {
7     fun ambilSandi(a: Int, b: Int): Int {
8       return a * b - a
9     }
10  }
11 }
```

Hasil keluaran dari kode tersebut adalah sebagai berikut :

```
1 4
```

Perhatikan cara pemanggilan fungsi `ambilSandi` pada baris ke-2 yang juga tanpa harus membuat instan dari kelas `Pegawai` dan tanpa menyertakan pemanggilan nama objeknya, `Sandi`.

3.12 Delegasi

Pola delegasi telah terbukti dapat menjadi alternatif implementasi pewarisan *interface*, dan Kotlin menyediakannya secara *native*. Mari kita perhatikan contoh kode berikut :

```
1  fun main(args: Array<String>) {
2    val data = Honda()
3    Proses(data).cetakInfo()
4  }
5
6  interface Mobil {
7    fun cetakInfo()
8  }
9
10 class Honda(): Mobil {
11   override fun cetakInfo() {
12     println("Ini implementasi mobil Honda")
13   }
14 }
15
16 class Proses(mobil: Mobil): Mobil by mobil
```

Hasil keluaran yang muncul adalah sebagai berikut :

```
1 Ini implementasi mobil Honda
```

Kita lihat bahwa kelas `Honda` mewarisi *interface* `Mobil`, kemudian melakukan `override` pada fungsi `cetakInfo`.

Kemudian ada kelas `Proses` yang pada konstruktornya memiliki 1 (satu) parameter, yaitu `Mobil`, deklarasi kelas `Proses` ini pun mewarisi *interface* `Mobil`, tetapi dengan kata kunci `by`.

Maksud dari penggunaan kata kunci `by` ini sebetulnya yaitu bahwa *method* implementasi yang berada di kelas `Proses` ini, yang diwariskan dari *interface* `Mobil`, akan bergantung pada nilai yang dilewatkan melalui parameter `mobil` pada konstruktor.

Contoh lebih jelas dapat dilihat pada kode program berikut :

```
1  fun main(args: Array<String>) {
2    var data = Honda()
3    Proses(data).cetakInfo()
4
5    var dataLain = Toyota()
6    Proses(dataLain).cetakInfo()
7  }
8
9  interface Mobil {
10   fun cetakInfo()
11 }
12
13 class Honda(): Mobil {
```

```
14   override fun cetakInfo() {
15     println("Ini implementasi mobil Honda")
16   }
17 }
18
19 class Toyota(): Mobil {
20   override fun cetakInfo() {
21     println("Ini implementasi mobil Toyota")
22   }
23 }
24
25 class Proses(mobil: Mobil): Mobil by mobil
```

Mirip seperti percobaan sebelumnya, kali ini kita memiliki 2 (dua) kelas yang mewarisi *interface* `Mobil`, dan masing-masing kelas memiliki implementasi fungsi `cetakInfo` masing-masing.

Kita lihat pada saat pemanggilan kelas `Proses` dengan parameter yang berbeda pada baris ke-3 dan baris ke-6, akan menghasilkan fungsi `cetakInfo` yang berbeda pula.

3.13 Mendelegasikan Properti

Tadi adalah delegasi yang terjadi pada tingkat kelas, untuk tingkat properti pun telah disiapkan oleh Kotlin. Contoh dasarnya dapat dilihat pada kode berikut :

```
1  import kotlin.reflect.KProperty
2
3  fun main(args: Array<String>) {
4    var data: String by GSData()
5
6    data = "tamami"
7    println("cetak $data")
8  }
9
10 class GSData {
11   var data: Any? = ""
12
13   operator fun getValue(thisRef: Any?, property: KProperty<*>): String {
14     return "nama properti : ${property.name} isinya $data"
15   }
16
17   operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String)
       {
18     println("nilai $value sudah disimpan di ${property.name}")
19     data = value
20   }
21 }
```

Hasil keluaran untuk kode di atas adalah sebagai berikut :

```
1 nilai tamami sudah di simpan di data
2 cetak nama properti : data isinya tamami
```

Padahal operasi yang terjadi adalah sederhana, namun karena implementasi `String` untuk variabel `data` telah didelegasikan ke kelas `GSData`, maka ada perubahan yang terjadi.

Perubahan yang pertama adalah pada saat properti `data` diisikan sebuah nilai, yaitu `tamami`, seperti pada baris ke-6, maka fungsi yang bekerja untuk menangani ini adalah fungsi `setValue` milik kelas `GSData`.

Ada 2 (dua) aktifitas yang dilakukan di fungsi `setValue`, yang pertama adalah mencetak informasi ke layar bahwa nilai dari parameter `$value` sudah disimpan. Kemudian menyimpan nilai dari `value` ke variabel atau properti `data` milik kelas `GSData`.

Saat inilah dicetak ke monitor informasi teks yang berbunyi `nilai tamami sudah di simpan di data`. Yang pada kondisi ini, data teks `tamami` telah diisikan ke properti `data`.

Perubahan berikutnya adalah pada saat melakukan akses data terhadap properti `data` seperti pada baris ke-7.

Pada baris ke-7, seharusnya hanya mencetak teks `cetak $data` dimana `$data` akan digantikan oleh nilainya, namun pada kenyataannya tidak demikian, yang terjadi adalah mencetak kalimat `cetak nama properti : data isinya tamami`, kata `cetak` disana sebetulnya memang isi dari baris ke-7, sedangkan sisanya adalah nilai yang dikembalikan dari fungsi `getValue` milik kelas `GSData`.

Ada beberapa delegasi properti yang umum yang mungkin dapat digunakan nantinya, yaitu :

- `lazy`
- `observable`
- `map`

3.13.1 `lazy`

`lazy` ini sebetulnya adalah fungsi hasil implementasi dari *interface* `Lazy` yang hanya berlaku untuk deklarasi `val`, nantinya operasi yang terjadi hanya akan terjadi sekali saja, kemudian nilai akhir yang dihasilkan akan disimpan, dan dibaca berulang saat ada pemanggilan atau akses terhadap properti ini.

Coba kita perhatikan contoh kode berikut :

```
1 val lazyVar: String by lazy {
2   println("init")
3   "tamami"
4 }
5
6 fun main(args: Array<String>) {
7   println(lazyVar)
8   println(lazyVar)
9 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 init
2 tamami
3 tamami
```

Kenapa hasilnya demikian? padahal pada baris ke-7 dan baris ke-8, akses terhadap `lazyVar` dilakukan 2 (dua) kali, tetapi proses pencetakan `init` hanya sekali.

Hal tersebut karena sifat dari `lazy` ini hanya melakukan prosesnya sekali di awal akses, kemudian nilainya disimpan untuk akses selanjutnya.

Jadi pada saat melakukan akses `lazyVar` di baris ke-7, delegasi propertinya akan melakukan secara lengkap, mulai dari proses mencetak teks `init`, hingga mengembalikan nilai teks `tamami`.

Namun pada saat melakukan akses `lazyVar` di baris ke-8, delegasi properti hanya akan mengembalikan nilai teks `tamami` saja.

3.13.2 observable

Untuk menggunakan delegasi properti `observable`, kita harus melakukan *import* terlebih dahulu terhadap kelas `kotlin.properties.Delegates`.

Delegasi properti `observable` ini membutuhkan 2 (dua) parameter, yang pertama adalah nilai awal, yang kedua adalah parameter untuk melakukan tugas perubahan data. Parameter kedua inilah yang akan dipanggil berulang-ulang apabila ada perubahan data.

Sebetulnya parameter kedua dari `observable` ini akan dieksekusi setelah pengisian data pada variabel / properti terjadi.

Pada parameter ke-2 ini, memiliki 3 (tiga) parameter di dalamnya, yaitu properti yang diisikan nilainya, nilai lama, dan nilai baru. Untuk lebih jelasnya coba perhatikan kode berikut :

```
1  import kotlin.properties.Delegates
2
3  var nama: String by Delegates.observable("[kosong]") {
4    properti, lama, baru ->
5      println("properti ${properti.name} telah berubah dari $lama ke $baru")
6  }
7
8  fun main(args: Array<String>) {
9    nama = "tamami"
10   nama = "ami"
11 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1  properti nama telah berubah dari [kosong] ke tamami
2  properti nama telah berubah dari tamami ke ami
```

Pada awalnya, nilai yang diberikan ke properti `nama` adalah teks `"[kosong]"`, kemudian pada saat ada perubahan atau pengisian data di baris ke-9, maka parameter ke-2 dari `observable` yaitu bagian yang menangani perubahan data akan dijalankan.

Beberapa parameter dari bagian yang menangani perubahan ini akan terisi, dimana `properti` adalah bagian properti yang dideklarasikan dengan kata kunci `observable`, kemudian `lama` akan diisikan dengan data sebelumnya, yaitu teks `[kosong]`, dan `baru` akan diisikan dengan data yang telah diisikan, yaitu teks `tamami`.

Selanjutnya akan mencetak informasi sebagaimana baris ke-1 dari hasil keluaran di atas.

Pada saat perubahan data kedua, yaitu pada baris ke-10, isi dari parameter `properti` tentunya tetap, sedangkan isi dari parameter `lama` akan berubah menjadi teks `tamami` dan isi parameter `baru` akan menjadi teks `ami`.

Setelah itu akan dicetak keluaran dengan hasil sebagaimana baris ke-2 dari hasil keluaran di atas.

3.13.3 `map`

Penggunaan `Map` sebagai delegasi properti sebetulnya memanfaatkan `Map` sebagai parameter konstruktor atau fungsi yang biasanya diimplementasikan pada pencacah kode JSON atau hal yang berhubungan dengan data dinamis. Mari kita perhatikan contoh kode program berikut :

```
1  fun main(args: Array<String>) {
2    val pegawai = Pegawai(mapOf(
3      "nama" to "tamami",
4      "gaji" to 10000
5    ))
6
7    println(pegawai.nama)
8    println(pegawai.gaji)
9  }
10
11 class Pegawai(val map: Map<String, Any?>) {
12   val nama: String by map
13   val gaji: Long by map
14 }
```

Hasil keluaran dari kode program di atas adalah sebagai berikut :

```
1 tamami
2 10000
```

Parameter deklarasi kelas `Pegawai` hanya membutuhkan sebuah parameter saja seperti pada baris ke-11. Nantinya, isi dari `map` tersebut akan dipetakan otomatis dimana `key`-nya adalah nama properti dan `value`-nya adalah isi dari propertinya.

Sehingga pada saat properti diakses seperti pada baris ke-7 dan baris ke-8, akan ditampilkan nilai dari masing-masing propertinya.

BAB 4

FUNGSI DAN LAMDA

4.1 Fungsi

Fungsi sebetulnya adalah sekumpulan kode program yang spesifik untuk menyelesaikan satu permasalahan. Fungsi dalam Kotlin ditandai dengan kata kunci `fun`.

Contoh kode berikut akan memberikan gambaran deklarasi dan penggunaan fungsi pada Kotlin :

```
1 fun main(args: Array<String>) {
2   cetak("tamami")
3 }
4
5 fun cetak(nama: String) {
6   println("nama : $nama")
7 }
```

Hasil keluaran dari kode program di atas adalah sebagai berikut :

```
1 nama : tamami
```

Bila kita perhatikan, sebetulnya deklarasi fungsi `main` pada baris ke-1 sampai dengan baris ke-3 adalah deklarasi fungsi yang selalu kita sertakan, karena fungsi ini wajib ada apabila aplikasi ingin kita jalankan.

Fungsi `main` ini hanya menerima sebuah argumen bertipe `Array` yang sebetulnya akan terisi apabila kita menyertakan parameter pada saat menjalankan aplikasi. Kita akan coba ini nanti.

Yang kedua adalah fungsi dengan nama `cetak` yang deklarasinya ada pada baris ke-5 sampai dengan baris ke-7. Fungsi ini sederhana, hanya meminta sebuah parameter bertipe `String` dan mencetak parameter tersebut ke monitor.

Sekarang coba ubah kode di atas menjadi seperti berikut :

```
1 fun main(args: Array<String>) {
2   cetak(args[0])
3 }
4
5 fun cetak(nama: String) {
6   println("nama : $nama")
7 }
```

Setelah melakukan *compile*, jalankan aplikasi dengan memberikan parameter pada program yang dijalankan. Caranya adalah dengan menambahkan teks setelah nama *file* utamanya, misalkan nama *file* kode yang saya buat adalah `Test.kt`, setelah melalui proses *compile*, menghasilkan *file* dengan nama `TestKt.class`.

Cara eksekusi *file* `TestKt.class` tersebut adalah dengan cara berikut :

```
1 > kotlin TestKt [parameter]
```

Kali ini kita coba mengganti bagian `[parameter]` dengan teks `tamami`, perintah lengkapnya adalah sebagai berikut :

```
1 > kotlin TestKt tamami
```

Hasil yang dikeluarkan dari pemanggilan kode program di atas adalah sebagai berikut :

```
1 nama : tamami
```

Terlihat bahwa teks `tamami` yang menjadi parameter pertama, akan dilewatkan sebagai parameter di pemanggilan fungsi `cetak` seperti di baris ke-2, yang kemudian di cetak ke layar sebagaimana perintah pada baris ke-6.

4.1.1 Cara Penggunaan

4.1.1.1 Notasi Infix Apa itu notasi *infix*, notasi ini sebetulnya membuat pemanggilan sebuah fungsi menjadi seperti operator. Mungkin akan lebih mudah bila disajikan dalam bentuk kode program berikut :

```
1 fun main(args: Array<String>) {
2   var nama = "tamami"
3
4   println(nama sambung "Selamat Datang")
5 }
6
7 infix fun String.sambung(data: String): String {
8   return "$this $data"
9 }
```

Sehingga hasil yang muncul ke layar adalah sebagai berikut :

```
1 tamami Selamat Datang
```

Perhatikan pada baris ke-4, kita melihat bahwa fungsi `sambung` yang dideklarasikan pada baris ke-7 sampai dengan baris ke-9, karena memiliki kata kunci `infix` dapat berbentuk seperti operator biasa, cara memanggilnya tidak seperti kondisi fungsi pada umumnya.

Pada baris ke-8, kita melihat ada 2 (dua) variabel yang dipanggil, yaitu `$this` yang isinya adalah `String` yang akan disambung, dan variabel `$data` yang isinya adalah `String` sambungannya.

Beberapa hal agar model *infix* ini dapat diimplementasikan yaitu :

1. Fungsi yang dideklarasikan merupakan anggota dari kelas (berada dalam sebuah kelas) atau merupakan fungsi ekstensi dari suatu kelas..

2. Fungsi ini hanya memiliki sebuah parameter

3. Fungsi ini harus dideklarasikan dengan kata kunci `infix`.

4.1.1.2 Parameter

Parameter seperti halnya variabel atau properti, harus dideklarasikan dengan format `[nama] : [tipe data]`, nama pada kondisi deklarasi parameter wajib menyertakan tipe data secara eksplisit. Tiap parameter pada konstruktor atau fungsi dipisahkan dengan koma.

Coba perhatikan fungsi `sambung` pada kode sebelumnya, disana ada sebuah parameter dengan nama `data` dengan tipe data berupa `String`. Deklarasinya harus berbentuk seperti ini.

4.1.1.3 Parameter Default

Sebuah fungsi dapat memiliki parameter yang secara *default* sudah memiliki nilai. Parameter-parameter ini dapat dilewatkan saat memanggil fungsi. Contoh kodenya adalah sebagai berikut :

```
1  fun main(args: Array<String>) {
2    var pegawai = Pegawai()
3    pegawai.setBio("tamami")
4
5    println("nama : ${pegawai.nama}")
6    println("jabatan : ${pegawai.jabatan}")
7  }
8
9  class Pegawai {
10   var nama: String = ""
11   var jabatan: String = ""
12
13   fun setBio(nama: String, jabatan: String = "STAF") {
14     this.nama = nama
15     this.jabatan = jabatan
16   }
17 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 nama : tamami
2 jabatan : STAF
```

Perhatikan pada baris ke-3 dari kode di atas, bahwa pemanggilan fungsi `setBio` milik kelas `Pegawai` hanya menyertakan sebuah parameter sedangkan pada deklarasinya membutuhkan 2 (dua) buah parameter, yaitu `nama` dan `jabatan`.

Namun karena parameter `jabatan` memiliki nilai *default* berupa teks `STAF`, maka parameter ini dapat dilewatkan, karena akan diisikan otomatis dengan teks `STAF`.

Maka dari itu hasil keluaran yang kita dapatkan adalah seperti tampilan di atas.

Apabila ada kelas turunan yang melakukan *override* terhadap fungsi yang memiliki nilai *default*, maka aturannya tidak boleh didefinisikan ulang nilai *default*nya pada kelas turunan, jadi kelas turunannya akan mengikuti nilai *default* milik pendahulunya.

Kita perhatikan contoh kode berikut :

```
1  fun main(args: Array<String>) {
2    var pegawai = Pejabat()
3    pegawai.setBio("tamami")
4
5    println("nama : ${pegawai.nama}")
6    println("jabatan : ${pegawai.jabatan}")
7  }
8
9  open class Pegawai {
10   var nama: String = ""
11   var jabatan: String = ""
12
13   open fun setBio(nama: String, jabatan: String = "STAF") {
14     this.nama = nama
15     this.jabatan = jabatan
16   }
17 }
18
19 class Pejabat: Pegawai() {
20   override fun setBio(nama: String, jabatan: String) {
21     this.nama = nama
22     if(jabatan.equals("STAF")) {
23       println("Masa pejabat jabatannya staf")
24       return
25     }
26     this.jabatan = jabatan
27   }
28 }
```

Hasil keluaran untuk kode di atas adalah sebagai berikut :

```
1 Masa pejabat jabatannya staf
2 nama : tamami
3 jabatan :
```

Lihatlah pada baris ke-19 sampai dengan baris ke-28, disini adalah tempat deklarasi kelas `Pejabat` yang sebetulnya adalah turunan dari kelas `Pegawai`, perhatikan kembali bahwa kelas `Pegawai` ini harus memiliki kata kunci `open` agar bisa diwariskan ke kelas di bawahnya, kemudian fungsi `setBio` milik kelas `Pegawai` juga diberikan kata kunci `open` agar kelas dibawahnya dapat melakukan *override* terhadap fungsi ini.

Pada baris ke-20 fungsi `setBio` milik kelas `Pegawai` di *override*, dimana pada parameter `jabatan` sudah tidak dapat lagi diberikan nilai *default* karena nilai *default* sudah diberikan pada kelas pendahulunya, yaitu kelas `Pegawai`.

Dan pada baris ke-22, kita lakukan seleksi, apabila nilai parameter `jabatan` masih berisi nilai *default*, yaitu teks `STAF`, maka kita akan cetak informasi bahwa yang masuk kelas `Pejabat` ini tentunya harus sudah memberikan jabatan strukturan yang baru selain `STAF`, maka kemudian isi variabel / properti `jabatan` milik kelas `Pejabat` akan dikosongkan / tidak diisikan, sehingga hasil keluaran akan tampak seperti di atas.

4.1.1.4 Argumen Bernama

Parameter / argumen pada fungsi dapat diberikan nama parameternya untuk memudahkan pengisian. Contohnya coba perhatikan pada kode berikut ini :

```
1  fun main(args: Array<String>) {
2    var pegawai: Pegawai()
3    pegawai.setBio(nama="tamami, gaji=5000L)
4
5    println("nama : ${pegawai.nama}")
6    println("jabatan : ${pegawai.jabatan}")
7    println("gaji : ${pegawai.gaji}")
8    println("jenis kelamin : ${pegawai.jenisKelamin}")
9  }
10
11 class Pegawai {
12   var nama = ""
13   var jabatan = ""
14   var gaji = 0L
15   var jenisKelamin = true
16
17   fun setBio(nama: String, jabatan: String = "STAF", gaji: Long = 1500L,
18       jenisKelamin: Boolean = true) {
19     this.nama = nama
20     this.jabatan = jabatan
21     this.gaji = gaji
22     this.jenisKelamin = jenisKelamin
23   }
24 }
```

Hasil keluaran dari kode program di atas adalah sebagai berikut :

```
1 nama : tamami
2 jabatan : STAF
3 gaji : 5000
4 jenis kelamin : true
```

Penggunaan argumen / parameter bernama ini ada pada baris ke-3, dimana fungsi `setBio` milik kelas `Pegawai` hanya diisikan 2 (dua) parameter saja, yaitu `nama` dan `gaji`.

Dengan menggunaan penamaan parameter ini, mempermudah apabila ada banyak parameter, dan beberapa parameter sudah ada nilai *default*nya, sehingga untuk mengganti atau mengisikan salah satu parameter cukup dikenali dengan nama parameternya.

Fasilitas ini tidak dapat digunakan apabila memanggil fungsi milik kelas Java.

4.1.1.5 Pengembalian Unit

Setiap fungsi pasti memiliki nilai kembalian, apabila tidak dituliskan secara eksplisit, sebetulnya tetap ada nilai kembalian berupa objek `Unit`, mirip seperti `void` di bahasa pemrograman Java, yang ditujukan untuk *method* atau fungsi yang tidak memiliki nilai kembalian.

Perhatikan kode berikut ini :

```
1 fun main(args: Array<String>) {
2   cetak("tamami")
3 }
4
5 fun cetak(nama: String) {
6   println("nama : $nama")
7 }
```

Hasil keluaran dari kode program di atas adalah sebagai berikut :

```
1 nama : tamami
```

Pada baris ke-5 dari kode di atas yang mendeklarasikan fungsi `cetak` akan sama artinya bila kita ganti dengan deklarasi seperti kode berikut :

```
1 fun cetak(nama: String): Unit {
2   println("nama : $nama")
3 }
```

4.1.1.6 Ekspresi Tunggal

Untuk fungsi-fungsi yang hanya memiliki satu baris perintah, sebetulnya dapat menghilangkan tanda kurung kurawal ({}), dan isi dari baris perintahnya langsung dideklarasikan setelah tanda sama dengan (=).

Hal lain adalah, tipe data kembalian dapat dihilangkan, karena *compiler* akan otomatis melakukan deteksi terhadap ini. Berikut adalah contoh kodenya :

```
1 fun main(args: Array<String>) {
2   cetak("tamami")
3 }
4
5 fun cetak(nama: String) = println("nama : $nama")
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 nama : tamami
```

Pada baris ke-5 adalah contoh dari ekspresi tunggal dari fungsi `cetak`. Deklarasi fungsinya sama saja seperti fungsi pada umumnya, hanya saja tanda kurung kurawal dihilangkan, dan diganti dengan tanda sama dengan (=) dan langsung mendeklarasikan isi dari fungsi `cetak`.

4.1.1.7 Argumen Dinamis

Dalam kotlin, dimungkin kita membuat sebuah fungsi dengan jumlah argumen / parameter dinamis, dimana bisa saja jumlah parameter hanya satu, dua, atau lebih dari itu. Mari kita lihat contoh kode berikut :

```
1  fun main(args: Array<String>) {
2    println(sum(1,3,5,7))
3    println(sum(2,4,6))
4  }
5
6  fun sum(vararg data: Int): Int {
7    var result = 0
8    for(x in data) {
9      result += x
10   }
11
12   return result
13 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
16
12
```

Terlihat bahwa pada baris ke-2 dan baris ke-3 sama-sama memanggil fungsi `sum` tetapi dengan jumlah parameter yang berbeda, baris ke-2 menggunakan 4 (empat) parameter, sedangkan pada baris ke-3 menggunakan hanya 3 (tiga) parameter.

Hal tersebut mungkin terjadi karena deklarasi fungsi `sum` pada baris ke-6, parameternya menggunakan kata kunci `vararg` sehingga jumlah parameter yang ada dapat lebih fleksibel.

4.1.2 Skup Fungsi

Seperti telah kita coba sebelumnya, fungsi di Kotlin tidak hanya berada di dalam kelas, melainkan dapat ditempatkan di tingat teratas dari *file*.

Bukan hanya ditempatkan di tingkatan teratas dari *file*, fungsi juga dapat ditempatkan sebagai lokal, *member* dari suatu kelas, atau yang telah kita bahas sebelumnya, sebagai fungsi ekstensi.

Fungsi ekstensi telah kita bahas pada materi sebelumnya, kita akan bahas mengenai apa itu fungsi lokal, dan fungsi *member*.

4.1.2.1 Fungsi Lokal Fungsi yang berada di dalam fungsi lain bisa disebut fungsi lokal, fungsi lokal ini dapat melakukan akses terhadap variabel yang dideklarasikan di fungsi atasnya, coba perhatikan kode berikut ini :

```
fun main(args: Array<String>) {
  fun tambah(a: Int, b: Int) = a + b

  println(tambah(2,4))
}
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
6
```

Perhatikan pada baris ke-2, yaitu deklarasi fungsi `tambah` yang berada pada fungsi `main`, yang kemudian dipanggil pada baris ke-4 untuk dilihat hasilnya.

4.1.2.2 Fungsi Member Fungsi member ini sebetulnya sudah sering kita gunakan pada percobaan sebelumnya, fungsi ini menjadi anggota / member di dalam sebuah kelas. Perhatikan contoh berikut :

```
fun main(args: Array<String>) {
  var mhs = Mahasiswa()

  mhs.cetakNama("tamami")
}

class Mahasiswa {
  fun cetakNama(nama: String) {
    println("nama : $nama")
  }
}
```

Hasil keluaran untuk kode di atas adalah sebagai berikut :

```
1 nama : tamami
```

Deklarasi fungsi member untuk kode di atas ada pada baris ke-8 dengan nama `cetakNama`, karena fungsi ini berada di dalam kelas `Mahasiswa`.

4.1.3 Fungsi Generik

Seperti halnya Java, Kotlin pun dapat membuat fungsi generik dengan deklarasi tipe generik berupa kurung siku sebelum deklarasi nama fungsinya, perhatikan contoh kode berikut yang kita ubah dari kode sebelumnya :

```
1  fun main(args: Array<String>) {
2    var mhs = Mahasiswa()
3
4    mhs.cetakData("tamami")
5    mhs.cetakData(50L)
6  }
7
8  class Mahasiswa {
9    fun <T> cetakData(data: T) {
10     println("data : $data")
11   }
12 }
```

Hasil keluaran dari kode tersebut adalah sebagai berikut :

```
1 data : tamami
2 data : 50
```

Deklarasi fungsi generik ini ada pada baris ke-9 sampai dengan baris ke-11. Terlihat bahwa sebelum nama fungsi, ada deklarasi generik dengan tanda `<T>`, dan tipe data ini dijadikan tipe data para parameter `data`.

Pemanggilan fungsi ini ada pada baris ke-4 dan baris ke-5, dimana pada baris ke-4 dipanggil dengan mengisikan tipe data `String` pada parameternya. Kemudian pada baris ke-5, parameter diisi dengan tipe data `Long`. Keduanya dapat diterima dan di proses, karena tipe data apapun yang dilewatkan melalui parameter tersebut akan dicetak ke layar.

4.1.4 Fungsi rekursif

Fungsi rekursif adalah fungsi yang memanggil dirinya sendiri dalam implementasinya. Misalnya saja, untuk sederhananya adalah fungsi perkalian dengan operasi sederhana berupa penjumlahan dan pengurangan. Kita lihat contoh kodenya berikut :

```
1 fun main(args: Array<String>) {
2   val hasil = kali(3,5)
3
4   println(hasil)
5 }
6
7 fun kali(a: Int, b: Int): Int = if(b == 1) a else a + kali(a, b-1)
```

Hasil dari kode program di atas [1] adalah sebagai berikut :

```
15
```

Perhatikan bahwa di baris ke-7 ada deklarasi fungsi `kali` yang di dalamnya terdapat pemanggilan ke fungsi `kali` yang sama, namun dengan nilai yang berbeda, ini yang disebut fungsi rekursif.

Fungsi ini terlihat lebih sederhana daripada menggunakan perintah iterasi seperti berikut ini :

```
fun kali(a: Int, b: Int): Int {
  var result = a
  var pengali = b

  if(b == 1) return a
  else {
    while(pengali > 1) {
      pengali--
      result += a
      println("$pengali : $result")
    }
    return result
  }
}
```

Hasil dari kode program di atas adalah seperti berikut ini :

```
4 : 6
3 : 9
2 : 12
1 : 15
15
```

Dari kode iterasi yang panjang di atas, mendapatkan hasil yang sama dari sebuah perkalian.

Namun sayangnya, bentuk rekursif biasa seperti di atas sangat rentang sekali akan kesalahan yang dikenal dengan *stack overflow*, karena tiap dipanggil sebuah fungsi, komputer akan menyiapkan memori sebesar kebutuhan fungsi tersebut, bayangkan saat rekursi yang terjadi sudah lebih dari kemampuan memori komputer menyiapkan ruang untuk itu.

Optimalisasi yang dilakukan di Kotlin adalah mendukung *tail* rekursif, yaitu rekursif yang terjadi di belakang (bukan secara fisik, namun secara logika). Maksudnya rekursif yang telah dipanggil tidak akan ditunggu hasilnya untuk kemudian diproses kembali, melainkan prosesnya akan berlanjut ke fungsi berikutnya dan dapat menghilangkan fungsi sebelumnya dari memori.

Contoh kode programnya adalah sebagai berikut :

```
import java.math.BigDecimal

fun main(args: Array<String>) {
  val hasil = angkaAjaib()

  println(hasil)
}
```

[1] https://bertzzie.com/knowledge/analisis-algoritma/Rekursif.html#tail-call

```
8
9  tailrec fun angkaAjaib(a: BigDecimal = BigDecimal("0.1")): BigDecimal =
10   if((a.plus(a)).compareTo(a.times(a)) == 0) a else
11     angkaAjaib(a.plus(BigDecimal("0.1")))
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 2.0
```

Pemanggilan rekursif kali ini berbeda, ada kata kunci `tailrec` sebelum deklarasi fungsi `angkaAjaib`.

Fungsi `angkaAjaib` memiliki sebuah parameter dengan tipe data `BigDecimal` milik Java. Nilai awal yang diberikan adalah `0.1`. Fungsi ini hanya memastikan angka berapa yang nilai penjumlahan dan perkaliannya adalah sama.

Apabila angka yang diberikan di parameter fungsi (variabel `a`) tidak dapat memenuhi ketentuan itu, maka jumlahkan dengan `0.1` kemudian panggil kembali fungsi `angkaAjaib`. Fungsi inilah yang disebut *tail* rekursif.

Saya menggunakan tipe data `BigDecimal` karena pada saat penggunaan tipe data `Double` maupun `Float` menghasilkan angka yang aneh pada prosesnya.

4.2 Fungsi *Higher-Order* dan Lamda

Fungsi *higher-order* adalah fungsi yang memiliki parameter berupa pemanggilan terhadap fungsi lain atau mengembalikan sebuah fungsi. Perhatikan contoh kode berikut untuk lebih jelasnya :

```
1  fun main(args: Array<String>) {
2    val dataList = listOf(1,3,5,7)
3    val data: String = cetakList(dataList, ::cetak)
4
5    println(data)
6  }
7
8  fun <T> cetakList(list: List<Int>, body: () -> T): T {
9    println(list.first())
10   try {
11     return body()
12   } finally {
13     println(list.last())
14   }
15 }
16
17 fun cetak(): String = "Cetak dulu ini"
```

Hasil dari kode program di atas adalah sebagai berikut :

```
1 1
2 7
3 Cetak dulu ini
```

Deklarasi dari fungsi *higher-order* ini ada pada baris ke-8 sampai dengan baris ke-15, fungsi tersebut bernama `cetakList`.

Fungsi `cetakList` ini memiliki 2 (dua) parameter, yaitu yang pertama adalah `List`, dan yang kedua adalah sebuah fungsi yang memiliki nilai kembalian berupa tipe data `T`.

Dalam fungsi `cetakList` ini, hanya ada 3 (tiga) perintah penting, yang pertama adalah mencetak elemen pertama dari parameter `List`, yang kedua adalah mengembalikan nilai fungsi dari parameter kedua dari fungsi `cetakList`, yang ketiga adalah mencetak elemen terakhir dari parameter `List`.

Parameter kedua dari fungsi `cetakList` ini dikembalikan keluar fungsi untuk diproses kemudian.

Bila diperhatikan pada baris ke-3 dari kode di atas, parameter pada fungsi `cetakList` yang pertama berisi `List` dari angka `1, 3, 5`, dan `7`. Dan parameter yang kedua adalah fungsi `cetak` yang mengembalikan `String` berupa teks `Cetak dulu ini`, teks inilah yang dikembalikan keluar fungsi `cetakList` sehingga tercetak ke layar melalui perintah pada baris ke-5.

Cara lain yang lebih umum adalah dengan menggunakan ekpresi lamda. Kita ubah kode di atas dengan cara ekspresi lamda, perhatikan kode berikut ini :

```
1  fun main(args: Array<String>) {
2    val dataList = listOf(1,3,5,7)
3    val data: String = cetakList(dataList, { "Cetak ini" })
4
5    println(data)
6  }
7
8  fun <T> cetakList(list: List<Int>, body: () -> T): T {
9    println(list.first())
10   try {
11     return body()
12   } finally {
13     println(list.last())
14   }
15 }
```

Hasil keluaran untuk kode di atas adalah sebagai berikut :

```
1 1
2 7
3 Cetak ini
```

Perhatikan pada baris ke-3 dimana pemanggilan fungsi `cetak` sudah tidak lagi diperlukan disini, tetapi langsung mendeklarasikan teks `Cetak ini` dengan tanda kurung kurawal di depan dan di belakangnya.

Beberapa ciri penggunaan ekspresi lamda ini adalah sebagai berikut :

- Expresi lamda selalu di awali dan di akhiri dengan tanda kurung kurawal.
- Parameter (jika ada) akan ditempatkan di sebelah kiri tanda panah `->`, namun parameter bisa diabaikan bila memang tidak ada.
- Inti / tubuh ekspresi lamda adalah setelah tanda panah `->`.

Perhatikan kode ekspresi lamda berikut yang sudah kita ubah agar memiliki parameter :

```
1 fun main(args: Array<String>) {
2   val dataList = listOf(1,3,5,7,9,11)
3   val data: Int = cetakList(dataList, {dataList.get(it * 2)})
4
5   println(data)
6 }
7
8 fun <T> cetakList(list: List<Int>, body: (data: Int) -> T): T {
9   println("data ke-1: ${list.first()}")
10  return body(2)
11 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 data ke-1 : 1
2 9
```

Deklarasi fungsi `cetakList` pada baris ke-8 sampai dengan baris ke-11, membutuhkan ekspresi lamda pada parameter kedua dengan sebuah parameter fungsi lamda bertipe data `Int` yang bernama `body`.

Di dalamnya hanya perintah sederhana yang mencetak elemen pertama dari `List` seperti pada baris ke-9 dan mengembalikan fungsi dari parameter ke-2 dengan isi parameter berupa angka `2`.

Angka `2` tersebut akan dikembalikan ke ekspresi lamda pada baris ke-3, dimana karena isinya hanya sebuah parameter, maka variabel parameter ini akan dibuatkan secara implisit dengan kata kunci `it`.

Hasilnya dapat dilihat bahwa parameter yang diisikan angka `2` pada baris ke-10, kemudian dikalikan dengan `2` pada baris ke-3, didapat hasilnya adalah elemen ke-4 dari variabel `dataList`.

Contoh lain adalah pada penggunaan `Map` yang membutuhkan 2 (dua) parameter, yaitu *key* dan *value*, karena memang sifat tampungan data pada `Map` adalah pasangan antara *key* dan *value*.

Jika kita melihat pada dokumentasi API yang disediakan untuk fungsi ekstensi `forEach` milik `Map` akan terlihat deklarasi seperti berikut :

```
1 fun <K,V> Map<out K, V>.forEach(action: (Entry<K, V>) -> Unit)
```

Bila diperhatikan, bahwa fungsi ekstensi `forEach` ini hanya memerlukan sebuah parameter yang dapat kita buatkan fungsi lamdanya. Sekarang kita implementasikan informasi API `forEach` tersebut dalam kode yang akan kita buat berikut :

```
1 fun main(args: Array<String>) {
2   val dataMap = mapOf(1 to "data1", 2 to "data2", 3 to "data3")
3
4   dataMap.forEach { _, nilai -> println(nilai) }
5 }
```

Hasil keluaran di atas akan terlihat sebagai berikut :

```
1 data1
2 data2
3 data3
```

Pada implementasinya di baris ke-4, karena kita akan menampilkan nilainya saja (*value*), kita tidak akan menampilkan *key*-nya, maka parameter *key* dapat kita ganti dengan karakter garis bawah (_), selebihnya tinggal melakukan pencetakan `nilai` pada bagian tubuh fungsi lamda.

4.3 Fungsi Dalam Baris

Fungsi dalam baris itu secara sederhana adalah memindahkan fungsi yang seharusnya menjadi parameter fungsi lain, ke dalam baris yang memanggil fungsi tersebut. Penjelasan akan lebih terlihat apabila kita melihat kode berikut :

```
1 fun main(args: Array<String>) {
2   fungsiSaya{ println("isinya") }
3 }
4
5 fun fungsiSaya(data: () -> Unit) {
6   println("awal")
7   data()
8   println("akhir")
9 }
```

Hasil keluaran dari kode program di atas adalah sebagai berikut :

```
1 awal
2 isinya
3 akhir
```

Terlihat kode di atas [2] biasa saja, parameter yang diberikan pada saat pemanggilan fungsi `fungsiSaya` akan dicetak setelah mencetak teks `awal` dan sebelum teks `akhir`.

Namun sebenarnya, Kotlin akan mengubah kode dari fungsi `fungsiSaya` tersebut ke dalam kode dasar kelas Java yang kurang lebih akan jadi seperti ini :

```
1 public void fungsiSaya(Function data) {
2   System.out.println("awal");
3   data.panggil();
4   System.out.println("akhir");
5 }
```

Kemudian fungsi lambda pada baris ke-2 akan diubah ke dalam kode dasar kelas Java menjadi seperti ini :

```
1 fungsiSaya(new Function() {
2   @Override
3   public void panggil() {
4     System.out.println("isinya");
5   }
6 });
```

Bayangkan bila ada banyak pemanggilan fungsi lambda disana, maka akan sebanyak itulah instan dari kelas `Function` yang terbentuk yang akan menimbulkan tidak efisiennya penggunaan memori komputer.

[2] https://stackoverflow.com/questions/44471284/when-to-use-an-inline-function-in-kotlin

Maka Kotlin menyediakan fungsi *inline* untuk menjadikannya lebih efisien, kata kunci yang digunakan untuk mendeklarasikan fungsi ini adalah dengan kata kunci *inline*. Berikut adalah contoh kodenya yang kita ubah dari kode sebelumnya :

```
1 fun main(args: Array<String>) {
2   fungsiSaya{ println("isinya") }
3 }
4
5 inline fun fungsiSaya(data: () -> Unit) {
6   println("awal")
7   data()
8   println("akhir")
9 }
```

Hasil keluarannya sama persis dengan kode sebelumnya tanpa kata kunci `inline`. Namun, terjemahan untuk kode dasar kelas Java-nya menjadi demikian :

```
1 println("awal");
2 println("isinya");
3 println("akhir");
```

Tanpa membuat instan dari kelas `Function` yang artinya, tidak perlu menyiapkan ruang memori yang terlalu besar sebesar kelas `Function` untuk melakukan tindakan pencetakan seperlunya.

BAB 5

JAVA INTEROPERABILITAS

Karna Kotlin melakukan kompilasi ke dalam kelas Java, maka sebetulnya Kotlin mampu untuk menggunakan pustaka-pustaka yang ditulis dan dibangun menggunakan bahasa Java. Begitu pula sebaliknya.

5.1 Gunakan Java di Kotlin

Kotlin dibangun dengan memikirkan penggabungannya dengan pustaka Java. Kode yang dibangun di Java dapat dengan mudah digunakan di Kotlin, begitu pula sebaliknya. Coba perhatikan kode yang ditulis dalam bahasa Kotlin yang menggunakan pustaka `ArrayList` dari Java :

```
1 import java.util.ArrayList;
2
3 fun main(args: Array<String>) {
4   val list = ArrayList<Int>()
5
6   list.add(1)
7   list.add(4)
8   list.add(7)
9
```

```
10   for(i in list) {
11     println(i)
12   }
13 }
```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```
1 1
2 4
3 7
```

Terlihat bahwa kode di atas menggunakan kelas `ArrayList` yang ada pada pustaka Java.

5.1.1 Fungsi *Getter* dan *Setter*

Untuk fungsi *getter* dan *setter*, di Kotlin akan dikenal sebagai properti. Jadi misalkan ada fungsi *getter* dan *setter*-nya di Java, cukup dipanggil nama propertinya saja.

Perhatikan kode Java berikut ini :

```
1 public class Pegawai {
2   private String nama;
3
4   public void setNama(String nama) {
5     this.nama = nama;
6   }
7
8   public String getNama() {
9     return nama;
10  }
11 }
```

Kode tersebut dapat di*compile* dengan `javac` kemudian nantinya akan kita gunakan pada kode Kotlin berikut :

```
1 fun main(args: Array<String>) {
2   val pegawai = Pegawai()
3
4   pegawai.nama = "tamami"
5
6   println(pegawai.nama)
7 }
```

Yang perlu di perhatikan untuk *compile* kode di atas adalah, kelas `Pegawai` yang dibuat dengan kode Java harus dalam 1 (satu) direktori dengan *file* Kotlin yang kita buat.

Compile terlebih dahulu kelas `Pegawai` agar dapat digunakan. Untuk melakukan *compile file* Kotlin harus menggunakan opsi *classpath* seperti berikut ini (misalkan nama *file* Kotlin yang saya buat adalah `Test.kt`) :

```
1 kotlinc -cp . Test.kt
```

Begitu pula pada saat kita akan menjalankan aplikasi Kotlin yang telah kita buat, gunakan opsi *classpath* seperti berikut ini :

```
1 kotlin -cp . TestKt
```

Hasil keluarannya akan tampak seperti berikut ini :

```
1 tamami
```

Terlihat bahwa pada kode Kotlin yang telah kita buat, kita menggunakan kelas `Pegawai` dari bahasa Java. Kemudian melakukan akses ke properti `nama` milik kelas `Pegawai` dengan memberinya nilai seperti pada baris ke-4.

Pada saat melakukan pengambilan data seperti pada baris ke-6 pun, cukup melakukan akses ke nama propertinya yang Kotlin akan menerjemahkan untuk melakukan akses atau mengambil nilai dengan fungsi `getNama` yang ada di Java.

Bagaimana jika properti di Java hanya memiliki sebuah *method* `set` saja di dalamnya, maka Kotlin tidak akan bisa melakukan akses terhadap properti ini, yang dapat kita lakukan adalah dengan melakukan akses langsung terhadap nama fungsinya. Dijelaskan dengan kode (kita ubah kode Java sebelumnya) adalah sebagai berikut :

```
1 public class Pegawai {
2   private String nama;
3
4   public void setNama(String nama) {
5     this.nama = nama;
6   }
7 }
```

Maka kode di Kotlin akan melakukan akses dengan cara berikut :

```
1 fun main(args: Array<String>) {
2   val pegawai : Pegawai()
3
4   pegawai.setNama("tamami")
5 }
```

5.1.2 Nilai Kembalian `void`

Saat sebuah *method* mengembalikan nilai `void` di Java, maka dalam Kotlin akan digantikan dengan kelas `Unit` pada saat *compile*.

5.1.3 Kata Kunci di Kotlin Jadi Nama *Method* di Java

Bila di Java menggunakan nama *method* yang sama dengan kata kunci di Kotlin seperti `is`, `in`, `object`, dan lainnya, kita masih dapat menggunakannya dengan cara memberikan tanda *backtick*. Berikut contoh kode di Javanya :

```
1 public class Pegawai {
2   private String bagian;
3
4   public Pegawai(String bagian) {
5     this.bagian = bagian;
6   }
7
8   public boolean in(String bagian) {
9     if(bagian.equals(this.begin)) return true;
10    else return false;
11  }
12 }
```

Kita lihat pada baris ke-8 bahwa *method* dengan nama `in` ada di Java, untuk melakukan akses terhadap *method* ini di Kotlin adalah dengan cara berikut ini :

```
1 fun main(args: Array<String>) {
2   val pegawai = Pegawai("dattap")
3
4   if(pegawai.`in`("dattap")) println("betul") else println("lain")
5 }
```

Untuk memanggil *method* `in` seperti terlihat pada baris ke-4 di atas.

5.1.4 *Null-Safety*

Di Java ada kemungkinan variabel atau properti yang ada dalam sebuah kelas bernilai `null` sedangkan Kotlin akan menjaga agar tidak ada satu variabel atau properti yang bernilai `null` pada saat melakukan *compile*, lalu bagaimana solusinya, perhatikan 2 (dua) kode berikut.

Kode yang pertama di tulis dalam Java dan memiliki peluang untuk memberikan nilai `null` pada kelas yang melakukan akses terhadap properti `nama`. Berikut kodenya :

```
1 public class Pegawai {
2   private String nama;
3
4   public void setNama(String nama) {
5     this.nama = nama;
6   }
7
8   public String getNama() {
9     return nama;
10  }
11 }
```

Kita dapat menggunakan operator tanda tanya (`?`) untuk menampung data yang mungkin akan menghasilkan nilai `null` dari kelas `Pegawai` di atas, berikut kodenya di Kotlin :

```
1 import java.util.ArrayList
2
3 fun main(args: Array<String>) {
4   val list = ArrayList<String>()
5   list.add("data")
6
7   val pegawai = Pegawai()
8
9   val boleh: String? = pegawai.nama
10  val tidak: String = list[0]
11
12  println(boleh)
13  println(tidak)
14 }
```

Hasil keluaran dari kode program di atas adalah sebagai berikut :

```
1 null
2 data
```

Pada baris ke-9 dari kode Kotlin yang kita buat, ada operator tanda tanya (`?`) disana yang akan memberikan kelonggaran, atau melewatkan pemeriksaan atas properti `nama` milik kelas

`Pegawai` yang dimungkinkan bernilai `null`. Dan benar saja, pada saat dicetak seperti pada baris ke-12, hasilnya adalah `null`.

Selain itu, ada saatnya sebuah tipe data yang tidak dapat disebutkan secara eksplisit, apakah `null` atau apakah nilainya ada, yang biasa disebut tipe *platform*. Untuk kasus ini bisa menggunakan notasi tanda seru (`!`) setelah tipe datanya. Misalkan untuk tipe data `T`, dapat menggunakan `T!` yang artinya dapat berupa data yang mungkin `null` seperti notasi `T?` atau tipe data yang pasti bukan `null` seperti tanda `T`.

5.1.5 Persamaan Tipe Data

Dari percobaan sebelumnya, kita telah ketahui bahwa tipe data primitif Java tidak diterjemahkan seperti data primitif di Kotlin, namun akan dipetakan ke bentuk kelas, beberapa pemetaan yang dilakukan untuk tipe data primitif adalah sebagai berikut :

Java	**Kotlin**
byte	kotlin.Byte
short	kotlin.Short
int	kotlin.Int
long	kotlin.Long
char	kotlin.Char
float	kotlin.Float
double	kotlin.Double
boolean	kotlin.Boolean

Beberapa tipe data yang bukan termasuk ke tipe data primitif pun akan dipetakan sebagai berikut di Kotlin :

Java	**Kotlin**
java.lang.Object	kotlin.Any!
java.lang.Cloneable	kotlin.Cloneable!
java.lang.Comparable	kotlin.Comparable!
java.lang.Enum	kotlin.Enum!
java.lang.Annotation	kotlin.Annotation!
java.lang.Deprecated	kotlin.Deprecated!
java.lang.CharSequence	kotlin.CharSequence!
java.lang.String	kotlin.String!
java.lang.Number	kotlin.Number!
java.lang.Throwable	kotlin.Throwable!

Beberapa tipe data primitif yang dibentuk dalam kelas juga akan dipetakan di Kotlin seperti berikut ini :

Java	Kotlin
java.lang.Byte	kotlin.Byte?
java.lang.Short	kotlin.Short?
java.lang.Integer	kotlin.Int?
java.lang.Long	kotlin.Long?
java.lang.Char	kotlin.Char?
java.lang.Float	kotlin.Float?
java.lang.Double	kotlin.Double?
java.lang.Boolean	kotlin.Boolean?

Collection di Kotlin bisa berupa *read-only* atau dapat diubah, jadi ketentuan *collection* di Java akan berlaku seperti ini :

Java	Kotlin *read-only*	Kotlin dapat diubah	Kotlin *Platform*
Iterator¡T¿	Iterator¡T¿	MutableIterator¡T¿	(Mutable) Iterator¡T¿!
Iterable¡T¿	Iterable¡T¿	MutableIterable¡T¿	(Mutable) Iterable¡T¿!
Collection¡T¿	Collection¡T¿	MutableCollection¡T¿	(Mutable) Collection¡T¿!
Set¡T¿	Set¡T¿	MutableSet¡T¿	(Mutable) Set¡T¿!
List¡T¿	List¡T¿	MutableList¡T¿	(Mutable) List¡T¿!
ListIterator¡T¿	ListIterator¡T¿	MutableListIterator¡T¿	(Mutable) ListIterator¡T¿!
Map¡K, V¿	Map¡K, V¿	MutableMap¡K, V¿	(Mutable) Map¡K, V¿!
Map.Entry¡K, V¿	Map.Entry¡K, V¿	MutableMap.MutableEntry¡K, V¿	(Mutable) Map.(Mutable)Entry¡K, V¿

Semua kelas di atas berada dalam paket `kotlin.collections`.
Untuk larik sendiri, di Kotlin akan diterjemahkan sebagai berikut :

Java	Kotlin
int[]	kotlin.IntArray!
String[]	kotlin.Array¡(out) String¿!

5.1.6 Java Generik

Generik di Kotlin seperti pembahasan pada bagian 3.8, berbeda dengan kondisi generik di Java. Beberapa konversi yang dilakukan antara generik di Java dengan Kotlin adalah sebagai berikut :

- *Wildcard* di Java akan diterjemahkan berikut :
  - `MyMethod<? extends Kelas>` menjadi `MyMethod<out Kelas!>!.`
  - `MyMethod<? super Kelas>` menjadi `MyMethod<in Kelas!>!`
- Untuk tipe data mentah, yang biasanya tidak dideklarasikan akan menjadi seperti berikut :
  - `ArrayList` menjadi `ArrayList<*>!`

5.1.7 Larik Java

Untuk larik, ingatlah bahwa larik di Kotlin berbeda dengan larik di Java. Di Kotlin tidak diperbolehkan mengisikan data dari sub-kelas atau super-kelas, contohnya misalkan deklarasi yang disebutkan adalah `Array<Any>`, deklarasi tersebut tidak bisa diisikan dengan data seperti `Array<String>`. Dan untuk tipe data primitif di Java, ada kelas yang bertugas menangani masing-masing tipe data tersebut dalam larik seperti `IntArray`, `CharArray`, `FloatArray` dan seterusnya.

Gambaran kodenya akan terlihat seperti percobaan berikut, pertama kita buat dahulu kode dari Java dimana salah satu *method* membutuhkan larik primitif `int` untuk di cetak ke layar monitor. Kode Javanya sebagai berikut :

```
1 public class Pencetak {
2   public void go(int[] data) {
3     for(int i=0; i<data.length; i++) {
4       System.out.println(data[i]);
5     }
6   }
7 }
```

Kode Kotlin yang menggunakan kelas `Pencetak` tersebut adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   val data = intArrayOf(0,2,4,6)
3   val pencetak = Pencetak()
4
5   pencetak.go(data)
6 }
```

Hasil yang dikeluarkan oleh kode program di atas adalah sebagai berikut :

```
1 0
2 2
3 4
4 6
```

5.1.8 Varargs di Java

Varargs di Java sebetulnya mirip dengan larik, hanya saja deklarasinya memiliki perbedaan dan lebih terlihat dinamis. Contoh berikut akan memberikan gambaran bahwa *varargs* di Java dapat diakses dengan cara yang mirip dengan larik di Java apabila digunakan di dalam kode Kotlin. Berikut adalah kode dengan bahasa pemrograman Java yang memiliki parameter *varargs* :

```
1 public class Pencetak {
2   public void go(int... data) {
3     for(int i=0; i<data.length; i++) {
4       System.out.println(data[i]);
5     }
6   }
7 }
```

Sedangkan kode di Kotlin untuk melakukan akses terhadap fungsi `go` dengan parameter `data` yang berupa *varargs* adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   val data = intArrayOf(0, 2, 4, 6)
3   val pencetak = Pencetak()
4   pencetak.go(*data)
5 }
```

Hasil keluarannya sama seperti kode sebelumnya, perbedaan yang dapat kita lihat ada pada baris ke-4, yaitu saat melakukan pemanggilan fungsi `go` dimana ada tanda bintang (`*`) pada parameter `data` yang diberikan.

5.1.9 Pemeriksaan *Exception*

Di Kotlin, *exception* tidak diperiksa terlebih dahulu, artinya, saat kita menggunakan *method* di Java yang kemudian menghasilkan *exception*, maka kita dapat abaikan saja, karena pada saat kompile dilakukan pun tidak ada peringatan kesalahan.

Untuk lebih jelasnya, lihatlah kode berikut, kode pertama adalah kode Java yang menghasilkan *exception*, berikut kodenya :

```
1 public class Pencetak {
2   public void go(int... data) throws Exception {
3     if(data.length > 5) throw new Exception("Banyak Amat");
4
5     for(int i=0; i<data.length; i++) {
6       System.out.println(data[i]);
7     }
8   }
9 }
```

Dan kode Kotlin yang memanggil *method* Java `go` tersebut akan membuat kesalahan dengan mengisikan 6 (enam) parameter seperti kode berikut :

```
1 fun main(args: Array<String>) {
2   val data = intArrayOf(0, 2, 4, 6, 8, 10)
3   val pencetak = Pencetak()
4
5   pencetak.go(*data)
6 }
```

Pada saat *compile* dilakukan terhadap kode Kotlin tidak akan memunculkan masalah, tetapi begitu kita jalankan aplikasi Kotlin yang telah kita buat, maka baru akan muncul *exception* yang dikembalikan oleh *method* `go` dari kelas `Pencetak` di Java.

5.1.10 *Method* Kelas Objek di Java

Pada saat kita mendeklarasikan kelas `Object` di Java, maka pada saat digunakan di Kotlin, kelas tersebut akan dikonversi ke kelas `Any` di Kotlin.

Kelas `Any` ini berbeda dengan kelas `Object` karena hanya memiliki 3 (tiga) *method* saja, yaitu `toString()`, `hashCode()`, dan `equals()`. Untuk implementasi *method* lain dari kelas `Object` maka perlu didefinisikan dengan *fungsi ekstensi* seperti yang telah kita bahas sebelumnya.

5.1.11 Mengakses `static`

Static Member di Java apabila digunakan di Kotlin akan dikonversi menjadi *companion object* seperti penjelasan sebelumnya. Contoh kodenya adalah seperti berikut, kita akan ubah kode sebelumnya baik kode Java maupun kode di Kotlinnya, berikut adalah kode di Java yang kita ubah :

```
1 public class Pencetak {
2   public static void go(int... data) {
3     for(int i=0; i<data.length; i++) {
4       System.out.println(data[i]);
5     }
6   }
7 }
```

Kode di Kotlin yang akan menggunakan *method static* di atas adalah sebagai berikut :

```
1 fun main(args: Array<String>) {
2   val data = intArrayOf(0, 2, 4, 6)
3
4   Pencetak.go("data")
5 }
```

Hasilnya masih sama seperti di atas, yaitu akan mencetak seluruh angka yang ada dalam larik `data`. Yang berbeda adalah kita tidak perlu membuat instan baru seperti sebelumnya, namun langsung dapat dipanggil nama *method*nya seperti pada baris ke-4.

5.1.12 Java Reflection

Java Reflection ini sebetulnya sebuah paket yang digunakan untuk memeriksa kelas pada saat kondisi *runtime*. Beberapa hal yang dapat kita lihat dari *Java Reflection* ini adalah sebagai berikut :

- Nama Kelas
- Lingkup kelas
- Informasi Paket
- Super-Kelas
- Implementasi *Interface*
- Konstruktor
- *Method*
- Properti / Atribut
- Anotasi

Untuk mudahnya, perhatikan kelas berikut yang menggunakan *Java Reflection* :

```
1 public class Test {
2   public static void main(String args[]) {
3     Class cl = Test.class;
4
5     System.out.println(cl.getName());
6   }
7 }
```

Hasil dari kode di atas adalah sebagai berikut :

```
1 Test
```

Kode tersebut adalah kode Java yang menggunakan *Java Reflection* seperti pada deklarasi di baris ke-3, yaitu penggunaan kelas `Class`, yang kemudian informasi dapat diakses dengan *method* milik `Class`.

Hal tersebut dapat kita lakukan dalam Kotlin dengan kode sebagai berikut :

```
1 fun main(args: Array<String>) {
2   val data = Test()
3   val nama = data::class.java
4
5   println(nama.name)
6 }
```

Hasil dari kode di atas akan sama seperti kode Java sebelumnya, seperti berikut ini :

```
1 Test
```

Penggunaan *Java Reflection* ini seperti terlihat di baris ke-3, dimana pada baris ke-2 membentuk instan dari kelas `Test` milik Java, kemudian pada baris ke-3 dilakukan inspeksi terhadap kelas `Test` melalui instannya, yaitu `data`, kemudian dilakukan pencetakan nama kelas seperti pada baris ke-5.

5.2 Gunakan Kotlin di Java

Karena sebetulnya tiap proses *compile* yang terjadi di Kotlin adalah merubah kode Kotlin menjadi kode biner Java, jadi sangat mudah menggunakan kode yang dibangun dengan Kotlin untuk digunakan di Java.

5.2.1 Properti

Properti di Kotlin akan diubah atau diurai menjadi elemen-elemen Java seperti berikut ini :

- *Method* get, dimana *method* ini adalah gabungan dari nama properti yang diawali dengan kata kunci `get`.
- *Method* set, dimana *method* ini adalah gabungan dari nama properti yang diawali dengan kata kunci `set`. (Hanya berlaku untuk properti dengan kata kunci `var`).
- Properti dengan nama yang sama dengan tambahan kata kunci `private`.

Contohnya adalah seperti kode berikut ini :

```
class Pegawai {
  var nama: String
}
```

Dari kelas `Pegawai` di atas, memiliki sebuah properti yang apabila di-*compile* akan menjadi kode Java berikut :

```
public class Pegawai {
  private String nama;

  public void setNama(String nama) {
    this.nama = nama;
  }

  public String getNama() {
    return nama;
  }
}
```

Secara sederhana akan terbentuk seperti kode di atas. Apabila sebuah properti diberi nama dengan awalan `is` seperti misalkan `isStudent`, maka konversinya adalah *method get*-nya akan sama persis seperti nama propertinya, sedangkan *method set*-nya akan merubah awalan `is` menjadi awalan `set`, sehingga nama *method* akan menjadi `setStudent`.

Aturan tersebut berlaku untuk seluruh tipe data selama nama propertinya mengikuti pola seperti itu.

5.2.2 Fungsi Pada Paket

Fungsi yang berada di dalam paket langsung, bukan di dalam kelas, akan diperlakukan sebagai fungsi `static` di Java. Termasuk di dalamnya adalah fungsi ekstensi. Berikut adalah kode Kotlin yang dibangun yang memiliki *method* di bawah paket langsung:

```
package data

fun keterangan() {
  println("Test Keterangan")
}
```

Misalkan *file* untuk kode di atas diberi nama `Pegawai.kt`. Untuk kode Java yang melakukan / menggunakan akses fungsi `keterangan` dari kode Kotlin di atas adalah sebagai berikut :

```
public class Test {
  public static void main(String args[]) {
    data.PegawaiKt.keterangan();
  }
}
```

Hasil dari *compile* dan menjalankan kode di atas, keluarannya akan terlihat seperti ini :

```
Test Keterangan
```

Hal ini mungkin dicapai karena memang kode Kotlin yang dibuat bertugas hanya mencetak kata tersebut. Perhatikan baris ke-3 dari kode Java yang dibuat, ada beberapa bagian disana, dimana `data` adalah nama paketnya, `PegawaiKt` adalah hasil *compile* dari *file*

`Pegawai.kt` karena ada fungsi di dalam paketnya. Dan yang terakhir adalah `keterangan` yang sebetulnya nama fungsi di level atau tingkatan paket dengan nama `data`.

Untuk bagian `PegawaiKt` sebetulnya dapat kita ganti agar terlihat lebih pantas dengan kata kunci `@file:JvmName()`. Berikut contoh kode programnya dalam Kotlin :

```
1 @file:JvmName("Pegawai")
2
3 package data
4
5 fun keterangan() {
6   println("Test Keterangan")
7 }
```

Sehingga kode di Java akan berubah menjadi seperti ini :

```
1 public class Test {
2   public static void main(String args[]) {
3     data.Pegawai.keterangan();
4   }
5 }
```

Dengan hasil yang sama, namun kata `Pegawai` tidak lagi menggunakan seperti nama sebelumnya, yaitu `PegawaiKt`.

5.2.3 *Field* Instan

Jika ingin membuat sebuah properti di Kotlin persis seperti sebuah properti di Java, gunakan kata kunci `@JvmField`, batasannya adalah properti ini bukan tipe `private`, bukan pula `open`, `override` atau `const`. Jenis properti lain adalah properti `late-initialized` akan juga menjadi *field* murni di Java.

5.2.4 *Field* Statis

Properti yang dideklarasikan sejajar dengan nama objek atau dalam *companion* objek akan didefinisikan sebagai *field* statis di Java. Biasanya akan bersifat `private` juga, namun dapat dibuka dengan menggunakan kata kunci berikut :

- Anotasi `@JvmField`
- Kata Kunci `lateinit`
- Kata Kunci `const`

5.2.5 *Method* Statis

Seperti penjelasan sebelumnya bahwa *method* yang terbentuk di bawah paket langsung akan menjadi *method* statis di Java. Kotlin juga akan membentuk *method* statis apabila fungsi tersebut terbentuk didalam `object` atau *companion* objek jika diberikan kata kunci `@JvmStatic`.

5.2.6 Lingkup

Lingkup di Kotlin akan diterjemahkan dalam kelas Java sebagai berikut :

- `private` di kelas Kotlin akan tetap `private` di kelas Java.
- `private` di level paket akan menjadi milik paket lokal.
- `protected` akan tetap sebagai `protected`, namun yang perlu diingat adalah bahwa tiap properti dengan lingkup `protected` di Java masih dapat diakses selama dalam 1 (satu) paket, namun di Kotlin tidak memperbolehkan hal ini.
- `internal` akan menjadi `public` di Java.
- `public` akan tetap menjadi `public` di Java.

5.2.7 KClass

Adakalanya kita memerlukan sebuah *method* dengan parameter berupa tipe data `KClass`, tipe data ini tidak ada padanannya di Java, maka kita harus melakukan konversi secara manual seperti kelas `MainView` berikut :

```
1 kotlin.jvm.JvmClassMappingKt.getKotlinClass(MainView.class)
```

5.2.8 Tangani Kesamaan Ciri dengan `@JvmName`

Apabila ada 2 (dua) fungsi yang dibuat di Kotlin seperti berikut ini :

```
1 fun List<String>.filterValid(): List<String>
2 fun List<Int>.filterValid(): List<Int>
```

Kedua fungsi tersebut akan menjadi bentrok apabila di konversi ke Java karena Java hanya akan melihat sebuah fungsi berikut :

```
1 List<String> filterValid() {}
```

Hal ini dapat diselesaikan dengan kata kunci `@JvmName` dengan perubahan sebagai berikut di kode Kotlin :

```
1 fun List<String>.filterValid(): List<String>
2
3 @JvmName("filterValidInt")
4 fun List<Int>.filterValid(): List<Int>
```

Karena dengan tambahan kode tersebut, di Java kodenya akan menjadi seperti ini :

```
1 List<String> filterValid() {}
2
3 List<Int> filterValidInt() {}
```

5.2.9 Pembentukan *Overload*

Sebenarnya, saat kita menulis *method* di Kotlin dengan parameter yang sudah terisi secara *default*, saat diterjemahkan ke Java maka Java hanya akan mengenal satu *method* tersebut dengan jumlah parameter lengkap, misalnya kode kotlin seperti ini :

```
1 fun fungsi(nama: String, kelamin: String = "pria", jabatan: String = "Staf")
      {}
```

Maka di Java hanya akan dikenal satu fungsi dengan pola sebagai berikut :

```
1 void fungsi(String nama, String kelamin, String jabatan) {}
```

Padahal seharusnya di Kotlin fungsi `fungsi` masih dapat dipanggil dengan hanya sebuah parameter saja, yaitu `nama` seperti ini :

```
1 fungsi("tamami")
```

Agar kode di Java dapat dengan mudah mengikuti pola ini, maka fungsi tersebut harus diberikan kata kunci `@JvmOverloads` seperti berikut :

```
1 @JvmOverloads fun fungsi(nama: String, kelamin: String = "pria",
2   jabatan: String = "Staf") {}
```

Nantinya di Java akan dapat menggunakan salah satu dari *method* berikut yang terbentuk secara otomatis :

```
1 void fungsi(String nama) {}
2 void fungsi(String nama, String kelamin) {}
3 void fungsi(String nama, String kelamin, String jabatan) {}
```

BAB 6

PERKAKAS

6.1 Menggunakan Gradle

6.2 Menggunakan Maven

BAB 7

CONTOH KASUS APLIKASI CHAT