

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	xi
Daftar Tabel	xiii
Kata Pengantar	xv
1 Memulai	1
1.1 Katakan Hai	2
1.2 Sintak Dasar	3
1.2.1 Deklarasi Paket	3
1.2.2 Deklarasi Fungsi	3
1.2.3 Deklarasi Variabel	4
1.2.4 Deklarasi Komentar	4
1.3 Logat	4
1.3.1 Membuat Kelas Data	4
1.3.2 Nilai <i>Default</i> Untuk Parameter Fungsi	5
1.3.3 <i>Interpolasi</i> Teks	6
1.3.4 Pemeriksaan Instan	6
1.3.5 Penggunaan <i>Range</i>	7
1.3.6 <i>Read-only List</i>	8
	vii

1.3.7	<i>Read-only Map</i> dan Cara Mengaksesnya	8
1.3.8	Jalan Pintas Perintah <code>if not null</code>	9
1.3.9	Jalan Pintas Perintah <code>if not null and else</code>	10
1.3.10	Eksekusi Perintah <code>if null</code>	10
1.3.11	Eksekusi Perintah <code>if not null</code>	10
1.3.12	Kembalikan Pada Perintah <code>when</code>	11
1.3.13	Ekspresi <code>try catch</code>	11
1.3.14	Ekspresi <code>if</code>	12
1.4	Adat	12
2	Dasar-Dasar	15
2.1	Tipe Data	15
2.1.1	Angka	15
2.1.2	Karakter	17
2.1.3	<i>Boolean</i>	19
2.1.4	Larik	19
2.1.5	<i>String</i>	20
2.2	Paket	22
2.3	Mengatur Alur	22
3	Kelas dan Objek	23
3.1	Kelas	24
3.2	Properti	24
3.3	<i>Interface</i>	24
3.4	<i>Visibility Modifiers</i>	24
3.5	Ekstensi	24
3.6	Kelas Data	24
3.7	Kelas Tertutup	24
3.8	Generik	24
3.9	Kelas Bersarang	24
3.10	Kelas <i>Enum</i>	24
3.11	Ekspresi Objek dan Deklarasi	24
3.12	Delegasi	24
3.13	Mendelegasikan Properti	24
4	Fungsi dan Lamda	25
4.1	Fungsi	25
4.2	Fungsi Lanjutan dan Lamda	25

4.3	Fungsi Sebaris	25
4.4	<i>Coroutines</i>	25
5	Java Interoperabilitas	27
6	Perkakas	29
6.1	Menggunakan Gradle	29
6.2	Menggunakan Maven	29
7	Contoh Kasus	
	Aplikasi Chat	31

DAFTAR GAMBAR

DAFTAR TABEL

KATA PENGANTAR

Saat melihat keunggulan dari bahasa pemrograman Java yang mudah untuk *dimaintenance*, 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 memasang atau menginstall *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

2 MEMULAI

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 menunjukkan bahwa Kotlin masih menggunakan JRE untuk menjalankan programnya, karena memang sebelum melakukan instalasi Kotlin, Java harus diinstall 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. Mengetikkan 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 }
```

4 MEMULAI

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 baris 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 :

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

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

```
1 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 :

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

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

```
1 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 :

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

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

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

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

```

1 data ke-10
2 data ke-9
3 data ke-8
4 data ke-7
5 data ke-6
6 data ke-5
7 data ke-4
8 data ke-3
9 data ke-2
10 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 menunjukkan 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
4     tidak ada")
5 }
```

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 kebiasaan 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 :

- menggunakan `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. Penulisan-nya 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 sederhana 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 `it >= 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, konstruktor 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 *dikompilasi* 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 diketahui, karena perbedaan tipe data dapat menyebabkan perbedaan operasi yang dapat dilakukan kepadanya.

Tipe data secara garis besar dapat dikelompokkan 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 :

*Kotlin Siapa Suka,
Dasar-Dasar Pemrograman.*
By P. Tamami

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 sub-tipe 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 tipe 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 = 'l'
3     print(a)
4 }

```

Pada kode di atas, variabel `a` bertipe data `Char` dengan diberikan nilai berupa karakter `l`. 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 :

```

1 fun main(args: Array<String>) {
2     val a: Char = '\t'
3     val b: Char = '\b'
4     val c: Char = '\n'
5     val d: Char = '\r'
6     val e: Char = '\''
7     val f: Char = '\"'
8     val g: Char = '\\
9     val h: Char = '$'
10
11     println("isi a : $a text")
12     println("isi b : $b text")
13     println("isi c : $c text")
14     println("isi d : $d text")
15     println("isi e : $e text")
16     println("isi f : $f text")
17     println("isi g : $g text")
18     println("isi h : $h text")
19 }

```

Hasil keluaran dari kode di atas adalah sebagai berikut :

```

1 isi a :   text
2 isi b : \b text
3 isi c : 
4 text
5 text
6 isi e : ' text
7 isi f : " text
8 isi g : \ text
9 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 :

```

1 fun main(args: Array<String>) {
2     val a: Int
3     a = ubahKeDesimal('b')
4     println("nilai a : $a")
5 }
6
7 fun ubahKeDesimal(c: Char): Int {
8     return c.toInt()
9 }

```

Hasil dari kode tersebut adalah sebagai berikut :

```
1 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, sebenarnya 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 diisi dengan tanggal dan jam saat ini dengan cara pemanggilan konstruktor *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

2.3.2 Ekspresi when

2.3.3 Ekspresi for

2.3.4 Ekspresi while

2.3.5 Ekspresi break dan continue

BAB 3

KELAS DAN OBJEK

3.1 Kelas

3.2 Properti

3.3 *Interface*

3.4 *Visibility Modifiers*

3.5 Ekstensi

3.6 Kelas Data

3.7 Kelas Tertutup

3.8 Generik

3.9 Kelas Bersarang

3.10 Kelas *Enum*

3.11 Ekspresi Objek dan Deklarasi

3.12 Delegasi

3.13 Mendelegasikan Properti

BAB 4

FUNGSI DAN LAMDA

4.1 Fungsi

4.2 Fungsi Lanjutan dan Lamda

4.3 Fungsi Sebaris

4.4 *Coroutines*

BAB 5

JAVA INTEROPERABILITAS

BAB 6

PERKAKAS

6.1 Menggunakan Gradle

6.2 Menggunakan Maven

BAB 7

CONTOH KASUS APLIKASI CHAT
