**Rangkuman Kelas Memulai Pemrograman dengan Kotlin**

**Rangkuman dari Pendahuluan**

Kotlin merupakan bahasa pemrograman open-source yang mudah dipelajari oleh siapapun. Ini bisa dilakukan berkat kejeniusan para developer yang dipimpin langsung oleh Andrey Breslav. Selain mudah dipelajari, Kotlin memungkinkan kita untuk membuat program antar platform yang tentunya ini dapat mengurangi biaya dalam pembuatan program itu sendiri.

Selain itu dalam sub-modul pendahuluan ini Anda juga telah mengetahui beberapa hal sebagai berikut:

1. Selain mudah dipelajari, Anda juga dapat ikut berkontribusi di dalamnya karena Kotlin merupakan project open-source.
2. Kotlin mendukung 2 (dua) paradigma umum yang akan Anda sering jumpai yaitu object-oriented programming (OOP) dan functional programming (FP). Kedua paradigma tersebut akan sangat membantu proses pengembangan dengan masing-masing fitur yang dimilikinya.
3. Selain multiparadigm, Kotlin juga mendukung multiplatform yang berbeda dengan bahasa pemrograman mainstream lainnya di mana kita dapat membuat aplikasi mobile (iOS/Android), Web, Desktop, atau Server. Bahkan Kotlin digadang-gadang bisa digunakan untuk Deep Learning dengan [KotlinDL](https://github.com/jetbrains/kotlindl) yang saat ini sudah berstatus alpha!
4. Kotlin pernah berada di posisi pertama dalam Fastest growing languages versi [GitHub Octoverse 2018](https://octoverse.github.com/2018/projects#repositories) berkat banyaknya dukungan komunitas dalam pengembangannya.
5. Tersedia berbagai macam framework yang bisa Anda gunakan untuk mempermudah pengembangan aplikasi server-side menggunakan Kotlin seperti:
   * [Spring](https://spring.io/)Spring merupakan sebuah framework yang sangat terkenal di Java. Spring bisa digunakan pada Kotlin untuk komunikasi ke API dengan lebih ringkas. Tersedia juga [Spring Initializr](https://start.spring.io/#!language=kotlin) yang memungkinkan kita untuk membuat proyek Spring baru dengan Kotlin.
   * [Vert.x](http://vertx.io/)Merupakan sebuah framework untuk membuat reactive Web app di JVM. Anda bisa melihat repository-nya di <https://github.com/vert-x3/vertx-lang-kotlin>.
   * [Ktor](https://ktor.io/)Ktor adalah sebuah framework yang dikembangkan oleh JetBrains untuk membuat aplikasi Web di Kotlin. Ktor memanfaatkan coroutine untuk skalabilitas yang tinggi dan menawarkan API yang mudah digunakan.
   * [Kotlinx.html](https://github.com/kotlin/kotlinx.html)Merupakan sebuah DSL yang dapat digunakan untuk membuat HTML di aplikasi Web. Kotlinx.html dapat digunakan sebagai alternatif untuk sistem templating tradisional seperti JSP dan FreeMarker.
   * [Exposed](https://github.com/jetbrains/exposed)Sebuah framework SQL yang menyediakan kumpulan DSL yang mudah dibaca untuk menggambarkan struktur database SQL dan melakukan kueri sepenuhnya dengan Kotlin.
6. Kotlin mendukung dengan baik dan memiliki beberapa kelebihan dalam  mengembangkan aplikasi Android seperti di bawah ini:
   * **Compatibility**  
     Kotlin sepenuhnya kompatibel dengan JDK 6. Ini memastikan bahwa aplikasi yang dibangun dengan Kotlin dapat berjalan pada perangkat Android yang lebih lama tanpa ada masalah. Android Studio pun mendukung penuh pengembangan dengan bahasa Kotlin.
   * **Performance**  
     Dengan struktur bytecode yang sama dengan Java, aplikasi yang dibangun dengan Kotlin dapat berjalan setara dengan aplikasi yang dibangun dengan Java. Terdapat juga fitur seperti inline function pada Kotlin yang membuat kode yang dituliskan dengan lambda bisa berjalan lebih cepat dibandingkan kode yang sama dan dituliskan dengan Java.
   * **Interoperability**  
     Semua library Android yang tersedia, dapat digunakan pada Kotlin.
   * **Compilation Time**  
     Kotlin mendukung kompilasi inkremental yang efisien. Oleh karena itu, proses build biasanya sama atau lebih cepat dibandingkan dengan Java.
7. “Safe” merupakan salah satu karakteristik Kotlin yang sangat berguna di mana Anda bisa meminimalisir kesalahan NullPointerException.

**Rangkuman dari Persiapan Membangun dan Menjalankan Program Kotlin**

* Kotlin menggunakan *Java Development Kit* (JDK) sebagai SDK-nya.
* Terdapat banyak distribusi JDK yang bisa Anda gunakan seperti *OracleJDK*, *OpenJDK* atau *Azul Zulu JDK*.
* Kotlin berjalan di atas dan mengunakan JVM untuk menjalankan program yang Anda sudah kembangkan.
* Untuk melakukan otomatisasi seperti proses kompilasi, Anda akan menggunakan Gradle sebagai *build script* dalam pengembangan program.
* Anda bisa menggunakan Groovy atau Kotlin DSL sebagai gradle *build script*.
* Dalam proses instalasi JDK dan Gradle di Linux dan macOS, Anda akan memanfaatkan SDK Manager yang bernama SDKMAN.
* Sedangkan pada OS Windows, menggunakan installer yang sudah disediakan Azul Zulu JDK.
* Dalam menuliskan kode program, Anda bisa memanfaatkan IDE yang dikembangkan oleh JetBrains, yaitu Intellij IDEA yang terbagi atas 2 (dua) versi. Yang pertama versi **Ultimate** dan **Community** yang dapat digunakan secara gratis.
* Untuk membuat projek Kotlin, bisa dilakukan dengan 2 (dua) cara yaitu melalui terminal atau langsung menggunakan IntelliJ IDEA dengan project *wizard*-nya.

**Rangkuman dari Kotlin Fundamental**

* Sama seperti bahasa pemrograman lain, Kotlin memiliki fungsi untuk mencetak nilai pada console yaitu fungsi **print()** dan **println()**.
* Untuk mendeklarasi variable, Anda akan menggunakan kata kunci var atau val. var atau val digunakan untuk mengontrol nilai dari sebuah variabel. Dengan kata kunci var kita bisa mengubah nilai yang sudah kita inisialisasikan.
* Untuk membuat variable yang menampung data berupa text, Anda dapat menggunakan tipe data **Char** untuk menyimpan satu karakter dan tipe data **String** untuk menyimpan beberapa karakter.
* Sedangkan untuk menampung data berupa number, Anda akan menggunakan beberapa tipe data di bawah ini:
  + **Int (32 Bit)**  
    Int adalah tipe data yang umumnya digunakan untuk menyimpan nilai numerik. Int dapat menyimpan data dari range -2^31 sampai +2^31-1. Dengan ukuran 32 Bit kita bisa menggunakannya untuk menyimpan nilai yang besar. Catatannya, tetap lihatlah batasan nilai maksimal yang dapat dimasukkan.
  + **Long (64 Bit)**  
    Long adalah tipe data yang digunakan untuk menyimpan nilai numerik yang lebih besar yaitu dari range -2^63 sampai +2^63-1.
  + **Short (16 Bit)**  
    Short merupakan sebuah bilangan bulat yang hanya dapat menyimpan nilai yang kecil karena hanya berukuran 16 Bit.
  + **Byte (8 Bit)**  
    Dengan ukuran yang kecil, Byte hanya mampu menyimpan nilai yang kecil sama halnya seperti Short. Byte biasa digunakan untuk keperluan proses membaca dan menulis data dari sebuah stream file atau jaringan.
  + **Double (64 Bit)**  
    Sama halnya dengan Long yang memiliki ukuran yang besar, Double mampu menyimpan nilai numerik yang besar pula. Pada umumnya Double digunakan untuk menyimpan nilai numerik pecahan.
* Selanjutnya adalah Array, yakni tipe data yang memungkinkan Anda untuk menyimpan beberapa objek di dalam satu variabel.
* Kotlin juga memungkinkan Anda untuk membuat Array dengan tipe data primitif dengan memanfaatkan beberapa fungsi spesifik seperti berikut:
  + **intArrayOf() : IntArray**
  + **booleanArrayOf() : BooleanArray**
  + **charArrayOf() : CharArray**
  + **longArrayOf() : LongArray**
  + **shortArrayOf() : ShortArray**
  + **byteArrayOf() : ByteArray**
* Kotlin mendukung juga tipe data Boolean di mana tipe data yang hanya memiliki dua nilai, yaitu true dan false. Selain itu, Terdapat 3 (tiga) operator yang dapat digunakan pada Boolean:
  + **Conjunction atau AND (&&)**  
    Operator AND (&&) akan mengembalikan nilai true jika semua hasil evaluasi expression yang diberikan bernilai true.
  + **Disjunction atau OR (||)**  
    Berbeda dengan operator AND (&&), operator OR (||) akan mengembalikan nilai true jika hasil evaluasi dari salah satu expressions yang diberikan bernilai true.
  + **Negation atau NOT (!)**  
    Berbeda dengan operator AND (&&) dan operator OR(||), operator NOT(!) digunakan untuk melakukan negasi pada hasil evaluasi expression yang diberikan.
* Jika ingin menginisialisasi nilai dari sebuah variabel berdasarkan suatu kondisi. Untuk menyelesaikannya, gunakan **If Expression**.
* Dengan Kotlin kita mudah dalam mengelola variable nullable sehingga dapat meminimalisir terjadinya **NullPointerException** dengan menggunakan *Safe Call* dan *Elvis Operator*.
* Function atau fungsi merupakan sebuah prosedur yang memiliki keterkaitan dengan pesan dan objek. Ketika kita memanggil sebuah fungsi maka sebuah *mini-program* akan dijalankan. Fungsi sendiri bisa diartikan sebagai cara sederhana untuk mengatur program buatan kita.

**Rangkuman dari Control Flow**

* Untuk menghindari penggunaan konstan yang keliru, kita bisa memanfaatkan fitur Enumeration untuk menyimpan kumpulan objek yang telah didefinisikan menjadi tipe data konstanta.
* Jika memiliki beberapa ekspresi untuk menentukan hasil evaluasi, Anda bisa menggunakan when expression. Karena sebuah expression, when dapat mengembalikan nilai yang dapat ditampung pada sebuah variabel.
* Sama seperti when, if expression dapat mengembalikkan nilai yang dapat ditampung pada sebuah variabel. Namun sedikit berbeda dengan when, if lebih cocok digunakan jika ekspresi yang akan digunakan untuk dievaluasi hanya 1 (satu).
* Dalam mempelajari bahasa pemgrograman, Anda akan sering menjumpai istilah expressions dan statement.
* Jika ingin melakukan perulangan, ada beberapa cara yang dapat diterapkan yaitu:
  + **While**  
    While bersifat Entry Controlled Loop. Artinya, kondisi yang diberikan akan dievaluasi terlebih dahulu. Jika kondisi tersebut terpenuhi maka proses perulangan akan dijalankan.
  + **Do While**  
    Do While bersifat Exit Controlled Loop di mana proses perulangan akan langsung dijalankan di awal. Jika telah selesai, barulah kondisi yang diberikan akan dievaluasi.
  + **For Loop**For merupakan konsep perulangan pada blok yang sama selama hasil evaluasi kondisi yang diberikan terpenuhi atau bernilai true. For memanfaatkan tipe data Range untuk menentukan kondisi yang akan dievaluasi.
* Saat menggunakan **While** dan **Do While** perhatikan *infinite loop*, yaitu kondisi di mana proses perulangan berlangsung terus menerus sampai aplikasi menjadi *crash*.
* Saat menerapkan perulangan, kita bisa memanfaatkan kata kunci **break** dan **continue**. Kedua kata kunci tersebut digunakan untuk menentukan proses perulangan akan seperti apa di mana break digunakan untuk menghentikan proses perulangan walaupun hasil evaluasi masih menghasil true dan continue digunakan untuk melanjutkan proses perulangan selanjutnya.

**Data Classes**

Pada sub-modul ini, kita mempelajari sebuah fitur menarik pada Kotlin, yaitu **Data Classes**.

Kotlin mengenalkan konsep data class yang merupakan sebuah kelas sederhana yang bisa berperan sebagai data *container*. Data class adalah sebuah kelas yang tidak memiliki logika apapun dan juga tidak memiliki fungsionalitas lain selain menangani data.

Kenapa disebut dengan kelas sederhana? Seperti yang sudah kita ketahui, Kotlin memungkinkan kita untuk menulis kode dengan ringkas dan lebih efisien. Dalam membuat sebuah data class, kita tidak perlu menuliskan banyak kode yang seharusnya dibutuhkan untuk mengelola sebuah data. Data class mampu menyediakan beberapa fungsionalitas yang biasanya kita butuhkan untuk mengelola data hanya dengan sebuah *keyword* data.

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

Hanya dengan satu baris kode di atas, kompiler akan secara otomatis menghasilkan constructor, toString(), equals(), hashCode(), copy() dan juga fungsi componentN(). Tentunya ini jauh lebih mudah dan bersih dibandingkan kita harus menuliskan banyak kode secara manual.

Beberapa hal yang perlu diperhatikan dalam membuat sebuah data class adalah:

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

**Rangkuman dari Kotlin Functional Programming**

* Anatomi dari sebuah function terdiri dari 2 (adua) bagian utama, yaitu function header dan function body:
  + **Function Header**  
    Function header adalah bagian yang merupakan konstruksi dari sebuah fungsi untuk menentukan perilakunya akan seperti apa. Di dalam function header terdapat visibility modifier, kata kunci fun, nama, daftar parameter dan nilai kembalian dari fungsi tersebut.
  + **Function Body**  
    Function Body adalah bagian yang dalamnya kita akan menempatkan sebuah logika kode baik itu sebuah expression atau statement.
* Kotlin memiliki fitur seperti named dan default argument yang dapat menghindari kita dari kesalahan saat menyematkan sebuah argumen saat menggunakan sebuah fungsi.
* Selain named dan default argument, Kotlin juga memiliki fitur varargs yang dapat digunakan untuk menyederhanakan deklarasi beberapa parameter yang memiliki tipe yang sama.
* Kotlin mendukung 2 (dua) extension yang dapat digunakan, yaitu *Extension Functions* dan *Extension Properties*. Jika extension functions digunakan untuk menambahkan fungsi baru, extension properties tentunya digunakan untuk menambahkan sebuah properti baru.
* Lambda expression, atau biasa disebut dengan anonymous function atau function literal adalah fitur yang cukup populer sampai sekarang dalam dunia functional programming. Bisa disebut sebagai anonymous karena lambda tidak memiliki sebuah nama seperti halnya sebuah fungsi pada umumnya. Karena merupakan sebuah fungsi, lambda juga dapat memiliki daftar parameter, body dan return type. Berikut adalah beberapa karakteristik dari Lambda pada kotlin.
  + Dalam menggunakan lambda, kita tidak perlu mendeklarasi tipe spesifik untuk nilai kembaliannya. Tipe tersebut akan ditentukan oleh kompiler secara otomatis.
  + Walaupun merupakan sebuah fungsi, lambda tidak membutuhkan kata kunci fun dan *visibility modifier* saat dideklarasikan, karena lambda bersifat anonymous.
  + Parameter yang akan ditetapkan berada di dalam kurung kurawal {}.
  + Ketika ingin mengembalikan nilai, kata kunci return tidak diperlukan lagi karena kompiler akan secara otomatis mengembalikan nilai dari dalam body.
  + Lambda expression dapat digunakan sebagai argumen untuk sebuah parameter dan dapat disimpan ke dalam sebuah variabel.
* *Higher-Order Function*, yaitu sebuah fungsi yang menggunakan fungsi lainnya sebagai parameter, menjadikan tipe kembalian, ataupun keduanya. High-order function adalah salah satu fitur yang memanfaatkan Lambda dan .
* Kotlin standard library memiliki beberapa fungsi yang tujuan utamanya adalah bagaimana menuliskan logika kode di dalam konteks dari sebuah objek. Dalam menggunakan fungsi tersebut kita akan memanfaatkan lambda expression yang memiliki ruang lingkupnya sendiri, di mana dalam ruang lingkup tersebut kita dapat mengakses konteks dari sebuah objek. Fungsi inilah yang dinamakan sebagai scope function. Beberapa fungsi yang berada di dalamnya antara lain, let, run, with, apply, dan also.
* Sama seperti Java, Kotlin  mendukung juga mekanisme reflection yang berarti seperangkat fitur bahasa dan library yang memungkinkan kita untuk mengamati struktur kode dari proyek yang sedang kita kerjakan secara langsung.
* Saat menggunakan Kotlin, kita bisa mendeklarasi fungsi di dalam sebuah fungsi dan menggunakannya hanya di  dalam fungsi tersebut.
* Kotlin Collection adalah salah satu struktur data mumpuni yang banyak menyediakan fungsi untuk memudahkan kita dalam mengelola dan memanipulasi data. Pada sub-modul sebelumnya, kita sudah mempelajari beberapa fungsi yang disediakan seperti **map()**, **sum()**, dan **sorted()**. Selain beberapa fungsi yang sudah disebutkan, berikut fungsi lain yang dapat kita manfaatkan:
  + **Fold**  
    Fold, Anda bisa dengan mudah melakukan perhitungan setiap nilai yang berada di dalam sebuah collection tanpa harus melakukan iterasi item tersebut satu-persatu menggunakan fungsi **fold()**.
  + **Drop**  
    **drop()**, fungsi yang bisa kita manfaatkan untuk memangkas item yang berada di dalam sebuah objek collection berdasarkan jumlah yang kita tentukan.
  + **Take**  
    Fungsi **take()** bisa kita manfaatkan untuk menyaring item yang berada di dalam sebuah objek collection.
  + **Slice**  
    Bagaimana jika kita ingin menyaring item dari posisi tertentu? Untuk itu Anda bisa memanfaatkan fungsi **slice()**.
  + **Distinct**  
    Saat berurusan dengan item yang sama di dalam sebuah collection, untuk menyaring item yang sama tersebut kita akan melakukan iterasi dan membandingkan setiap itemnya. Namun dengan Kotlin kita tidak perlu melakukannya secara manual, karena Kotlin Collection menyediakan fungsi untuk melakukannya dengan mudah yaitu fungsi **distinct()**.
  + **Chunked**  
    Sama seperti fungsi **split()**, fungsi **chunked()** bisa kita gunakan untuk memecah nilai String menjadi beberapa bagian kecil dalam bentuk Array.
* Recursion merupakan sebuah teknik dasar dalam pemrograman yang bisa kita gunakan untuk menyederhanakan pemecahan masalah yang umumnya diselesaikan dengan cara yang kompleks. Di Kotlin, recursion disebut juga dengan *recursive function*.

**Rangkuman dari Kotlin Object-Oriented Programming**

* **Object-Oriented Programming** masih menjadi salah satu paradigma atau teknik pemrograman yang banyak digunakan dalam pengembangan aplikasi.
* Gambaran umum dari Object-Oriented Programming adalah sebuah **blueprint**, di mana *blueprint* ini adalah sebuah **Class** (kelas).
* Pada Object-Oriented Programming terdapat beberapa komponen penting yaitu **Object**, **Class**, **Attribute** dan **Behaviour**.
* **Object** atau Objek merupakan hasil realisasi dari sebuah *blueprint* atau *class* yang tentunya memiliki fungsi dan juga properti sama seperti *blueprint*-nya. Salah satu contoh Object yang sering kita pakai adalah nilai primitif pada Kotlin, yaitu **String**, **Integer**, **Char** dan **Boolean**. Kegunaan dari Objek sendiri adalah untuk mengakses berbagai properti dan fungsi pada kelas.
* **Class** merupakan sebuah *blueprint*. Setiap kelas memiliki atribut dan *behaviour*. Dalam Kotlin, attributes lebih sering disebut dengan *properties*, sedangkan *behaviour* sering disebut *functions*. Berikut penjelasan singkat dari setiap komponen:
  + **Class**: Merupakan sebuah blueprint yang terdapat properti dan fungsi di dalamnya.
  + **Properties**: Karakteristik dari sebuah kelas, memiliki tipe data.
  + **Functions**: Kemampuan atau aksi dari sebuah kelas.
* Properti dapat dideklarasikan sebagai nilai *mutable* dengan menggunakan **var** atau sebagai nilai *read-only* dengan menggunakan **val**.
* Konstruktor merupakan fungsi spesial yang digunakan untuk menginisialisasi properti yang terdapat pada sebuah kelas.
* Terdapat 3 (tiga) tipe konstruktor pada Kotlin, yaitu **primary constructor**, **secondary constructor** dan **default constructor**.
  + **Primary Constructor**
    - Seperti namanya, jika kita akan membuat suatu objek dari sebuah kelas dan kelas tersebut memiliki *constructor* di dalamnya, maka konstruktor tersebut adalah *primary constructor* dan diharuskan untuk mengirim nilai sesuai properti yang dibutuhkan.
    - Primary Constructor juga dapat memiliki nilai *default*, dengan begitu jika kita tidak menetapkan nilai untuk parameter tersebut maka properti tersebut akan memiliki nilai *default*.
    - Kotlin menyediakan blok *init* yang memungkinkan kita untuk menuliskan properti di dalam *body class* ketika kita menggunakan *primary* *constructor*.
  + **Secondary Constructor**
    - Secondary constructor digunakan ketika kita ingin menginisialisasi sebuah kelas dengan cara yang lain.
    - Dapat memiliki lebih dari satu *secondary constructor*.
  + **Default Constructor**
    - Kotlin secara otomatis membuat sebuah *default constructor* pada kelas jika kita tidak membuat sebuah konstruktor secara manual.
* Pada Object-Oriented Programming terdapat **Inheritance**/Pewarisan. Inheritance dapat mencegah kita melakukan perulangan kode dengan cara mengelompokkan properti dan fungsi yang sama. Pada Inheritance, terdapat **parent class** dan juga **child class.**
* **Abstract Class** merupakan gambaran umum dari sebuah kelas. Abstract Class tidak dapat direalisasikan dalam sebuah objek.
* **Visibility Modifier** adalah **hak akses** pada Kotlin. Dengan menentukan hak akses tersebut, kita dapat membatasi akses data pada sebuah kelas. Berikut macam-macam hak akses dan penjelasan singkatnya yang dapat digunakan pada Kotlin:
  + **Public**: Hak akses yang cakupannya paling luas. Anggota yang diberi *modifier* ini dapat diakses dari manapun.
  + **Private**: Hak akses yang cakupannya paling terbatas. Anggota yang menerapkannya hanya dapat diakses pada *scope* yang sama.
  + **Protected**: Hak akses yang cakupannya terbatas pada hirarki kelas. Anggota hanya dapat diakses pada kelas turunannya atau kelas itu sendiri.
  + **Internal**: Hak akses yang cakupannya terbatas pada satu modul. Anggota yang menggunakannya tidak dapat diakses diluar dari modulnya tersebut.
* Seluruh konten pada Kotlin, seperti kelas dan fungsi, dibungkus dalam sebuah *package*. Package tersebut digunakan untuk mengelompokkan kelas, fungsi dan variabel yang mempunyai kemiripan fungsionalitas.
* **Exception** adalah event (kejadian) yang dapat mengacaukan jalannya suatu program. Pada Kotlin semua *exception* bersifat **Unchecked**, yang artinya *exception* terjadi karena kesalahan pada kode kita. Kode yang baik yaitu kode yang terhindar dari segala bentuk kejadian dengan efek buruk pada aplikasi kita. Kejadian tersebut pada programming disebut **Exception**. Hal seperti ini seharusnya dihindari. Oleh karena itu kita harus mengetahui cara menangani suatu *exception* (**Exception Handling**).
* Berikut ini beberapa contoh Unchecked Exception yang sering mengganggu jalannya program kita:
  + **ArithmeticException**
    - Merupakan *exception* yang terjadi karena kita **membagi** suatu bilangan dengan nilai **nol**.
  + **NumberFormatException**
    - Disebabkan karena terjadi kesalahan dalam **format** angka.
  + **NullPointerException**
    - Terjadi karena sebuah variabel atau objek memiliki nilai **null**, padahal seharusnya objek atau variabel tersebut **tidak boleh** null.
* **Exception** **Handling** dapat diterapkan dengan beberapa cara. Di antaranya adalah dengan menggunakan **try-catch**, **try-catch-finally**, dan **multiple catch**.
* Pada Kotlin menggunakan dua atau lebih fungsi dengan nama yang sama disebut dengan **overloading**. Overloading dapat dilakukan selama fungsi itu memiliki parameter yang berbeda.
* **Extension** properties pada Kotlin sama halnya seperti melakukannya pada Extension function. Kita dapat menambahkan sebuah properti tanpa harus membuat sebuah kelas yang mewarisi kelas tersebut. Hal ini dilakukan dengan deklarasi khusus yang disebut dengan Extension.
* **Interfaces** merupakan suatu konsep sifat umum yang nantinya digunakan oleh suatu kelas agar dapat memiliki sifat tersebut.
* Teknik membuat sebuah kelas yang memang bertugas untuk mengatur atau mengelola fungsi *getter* dan *setter* untuk sebuah properti kelas pada Kotlin dinamakan **Delegate**.

**Rangkuman dari Kotlin Generics**

Inti yang telah dipelajari pada sub-modul ini adalah *Generics*, sebuah konsep yang memungkinkan suatu kelas atau interface menjadi tipe parameter yang dapat digunakan untuk berbagai macam tipe data. Tidak hanya itu saja, kita telah mengenal juga tentang *variance*, yaitu salah satu konsep dari *generics*. Berikut adalah rangkuman dari keseluruhan materi tersebut :

* *Generics* pada kotlin dapat diterapkan dengan meletakkan tipe parameter ke dalam angle brackets (**<>**). Tipe parameter atau alias ini merupakan word yang bisa bebas Anda deklarasikan, contoh yang paling umum adalah menggunakan word  ‘**T**’.

1. interface List<T>{
2. operator fun get(index: Int) : T
3. }

* Ada 2 jenis penerapan generics, yaitu **Generics pada Class**, dan **Generics pada Function**.
* **Generics pada Class** dilakukan dengan meletakkan tipe parameter setelah penulisan nama class.

1. interface List<T>{
2. operator fun get(index: Int) : T
3. }

* Class yang mewarisi sebuah **Generic Class** atau Interface harus menentukan tipe argumen sebagai tipe dasar dari parameter generic kelas tersebut. Tipe parameter yang digunakan bisa berupa tipe yang spesifik (Integer, Float, Double, atau Object lainnya) atau bisa juga class tersebut mendeklarasikan tipe parameter yang lainnya.
* **Generics pada Function** dilakukan dengan meletakkan tipe parameter sebelum penulisan nama dari fungsi tersebut.

1. fun <T> run(): T {
2. /\*...\*/
3. }

* Dalam menggunakan sebuah class maupun function generic, tipe parameter diletakkan setelah memanggil nama dari sebuah function ataupun class.

1. fun main() {
2. val longArrayList = ArrayList<Long>()
3. displayArray<Long>(longArrayList)
4. }

* *Constraint Type Parameter* adalah teknik untuk membatasi cakupan tipe data apa saja yang dapat di sediakan oleh **Generic Class** maupun **Generic Function**. Pembatasan ini dilakukan dengan membubuhkan tanda *colon* (:) setelah tipe parameter yang kemudian diikuti oleh tipe data yang akan dijadikan batasan.
* Batasan dalam **Constraint Type Parameter** adalah*child class* dari tipe data batasan tersebut. Jadi, semisal ditentukan batasannya adalah tipe **List**, maka hanya berlaku pada *child class* dari class tersebut.
* **Variance**, adalah konsep pada Generics yang menggambarkan bagaimana sebuah tipe yang memiliki subtipe yang sama dan tipe argumen yang berbeda saling berkaitan satu sama lain. Ada beberapa jenis dari *variance* yaitu:
  + **Covariant**  
    Ditandai dengan *keyword* ‘**out**’ sebelum deklarasi dari tipe parameter. Nilai dari tipe parameter tersebut hanya bisa diproduksi seperti menjadikanya sebagai return type dan tidak dapat dikonsumsi seperti menjadikannya sebagai tipe argumen untuk setiap fungsi di dalam kelas tersebut.
  + **Contravariant**  
    Ditandai dengan *keyword* ‘**in**’ sebelum deklarasi dari tipe parameter. Nilai dari tipe parameter tersebut hanya bisa dikonsumsi seperti menjadikannya sebagai tipe argumen untuk setiap fungsi dan tidak untuk dikonsumsi sebagai *return type* dari sebuah function di dalam kelas tersebut.

**Rangkuman dari Berkenalan dengan Coroutines**

Di sub-modul ini, kita telah membahas tentang apa itu *concurrency*, *pararelism*, *threading*, bahkan penerapannya pada kotlin dengan menggunakan fitur **Coroutines**. Cukup menarik dan mudah ‘kan ? Oke, mari kita ingat - ingat kembali apa saja yang telah dipelajari hingga saat ini:

* *Concurrency* adalah beberapa proses yang terjadi pada suatu sistem. Terjadi apabila terdapat 2 (dua) atau lebih proses yang tumpang tindih dalam satu waktu.
* *Parallelism*, sama seperti *concurrency*, namun 2 (dua) atau lebih proses tersebut dijalankan pada waktu yang sama persis.
* Ada beberapa konsep dasar dari penerapan **concurrency** yang harus dipelajari yaitu:
  + **Process**  
    Merupakan bagian dari proses aplikasi yang dijalankan. Setiap kali aplikasi dijalankan, maka saat itu juga proses dijalankan. Tergantung pada sistem operasi yang digunakan, suatu proses dapat terdiri dari beberapa thread yang menjalankan instruksi secara bersamaan.
  + **Thread**  
    Biasa dikenal sebagai proses yang ringan dan merupakan komponen dari suatu proses aplikasi yang menjalankan tugas tertentu secara spesifik.
  + **I/O-Bound**  
    Sebuah algoritma yang bergantung pada perangkat input atau output. Waktu untuk mengeksekusi sebuah I/O-bound tergantung pada kecepatan perangkat yang digunakan.
* Ada beberapa permasalahan yang ditimbulkan ketika menerapkan **concurrency** pada aplikasi, yaitu :
  + **Deadlocks**  
    Sebuah kondisi di mana dua proses atau lebih saling menunggu proses yang lain untuk melepaskan resource yang sedang digunakan yang mengakibatkan proses yang sedang berjalan tak kunjung selesai melakukan operasinya.
  + **Livelocks**  
    Kondisi di mana sebuah proses tidak bisa melanjutkan tugasnya, sama seperti **deadlocks**. Namun, perbedaannya adalah selama **livelocks** terjadi, keadaan aplikasi tetap bisa berubah. Walau perubahan keadaan tersebut menyebabkan proses berjalan dengan tidak semestinya.
  + **Starvation**  
    Kondisi yang biasa terjadi setelah **deadlock**. Kondisi ini sering kali menyebabkan sebuah proses kekurangan sumber daya. Pada kondisi ini, thread menjadi tidak mendapatkan akses reguler ke sumber daya bersama dan membuat proses terhenti.
  + **Race Conditions**Kondisi di mana terdapat banyak thread yang mengakses data yang dibagikan bersama dan mencoba mengubahnya secara bersamaan. Ini bisa terjadi ketika kode concurrent yang dituliskan untuk berjalan secara sekuensial.
* **Coroutines** merupakan fitur unggulan dan cara baru dalam menuliskan thread yang ringan dan efisien pada Kotlin. Disediakan **JetBrains** khusus untuk Kotlin.
* Untuk menjalankan **coroutine**, diperlukan fungsi yang biasa disebut sebagai *coroutine* *builder* yang mengambil suspending lambda dan membuat coroutine untuk menjalankannya. Ada beberapa macam *coroutine builder* yaitu:
  + **launch**  
    Fungsi ini digunakan untuk memulai sebuah coroutines yang tidak akan mengembalikan sebuah hasil dan menghasilkan **Job** yang dapat digunakan untuk membatalkan eksekusi.
  + **runBlocking**  
    Dibuat untuk menjembatani blocking code menjadi kode yang dapat ditangguhkan. **runBlocking** akan memblokir sebuah thread yang sedang berjalan hingga eksekusi coroutine selesai
  + **async**  
    Fungsi ini digunakan untuk memulai sebuah coroutine yang akan mengembalikan sebuah hasil.
* **Job**, merupakan hasil dari perintah asynchronous yang dijalankan dan merepresentasikan coroutine yang sebenarnya. Memiliki 3 properti terdiri dari **isActive, isCompleted, dan isCancelled** yang menunjukkan *state* dari **Job** tersebut.
* **Deferred**, adalah nilai kembalian dari fungsi **async** yang dapat berupa hasil atau exception. Merupakan nilai tangguhan yang dihasilkan dari proses coroutines dan bisa kita kelola sesuai dengan kebutuhan.
* **CoroutineDispatcher**, merupakan *base class* yang menentukan thread yang berfungsi untuk menjalankan coroutines. Class ini diimplementasikan pada beberapa class berikut :
  + **Dispatcher.Default**  
    Merupakan dispatcher dasar yang digunakan oleh semua standard builders seperti *launch, async*, dll jika tidak ada dispatcher lain yang ditentukan. **Dispatcher.Default** menggunakan kumpulan thread yang ada pada JVM. Jumlah maksimal thread yang digunakan adalah sama dengan jumlah core dari CPU.
  + **Dispatcher.IO**  
    Merupakan dispatcher yang digunakan untuk membongkar pemblokiran operasi I/O dan akan menggunakan kumpulan thread yang dibuat berdasarkan permintaan.
  + **Dispatcher.Unconfined**  
    Merupakan Dispatcher yang menjalankan coroutines pada thread yang sedang berjalan sampai mencapai titik penangguhan. Setelah penangguhan, coroutines akan dilanjutkan pada thread dimana komputasi penangguhan yang dipanggil.
* Selain dengan menggunakan metode diatas, ada beberapa *builder* lain yang dapat digunakan untuk menentukan *thread* yang dibutuhkan, antara lain:
  + **Single Thread Context**  
    Dispatcher ini menjamin bahwa setiap saat coroutine akan dijalankan pada thread yang ditentukan. Penerapannya dilakukan dengan memanfaatkan fungsi **newSingleThreadContext()** pada parameter dispatcher dalam fungsi **launch()**.
  + **Thread Pool**  
    Sebuah dispatcher yang memiliki kumpulan thread. Ia akan memulai dan melanjutkan coroutine di salah satu thread yang tersedia pada kumpulan tersebut. Runtime akan menentukan thread mana yang tersedia dan juga menentukan bagaimana proses distribusi bebannya. Penerapannya dilakukan dengan memanfaatkan fungsi **newFixedThreadPoolContext()** pada parameter dispatcher dalam fungsi **launch()**.
* **Channel**,  adalah adalah nilai deferred yang menyediakan cara mudah untuk mentransfer nilai tunggal antara coroutine atau bisa disebut melakukan interaksi antar coroutine serta membuat kode *concurrent* dapat berkomunikasi dengan mengirim dan menerima pesan tanpa harus peduli di thread mana coroutine berjalan..