Chapter 03.코틀린 고급 - 02.데이터 클래스

데이터 클래스

1. 데이터 클래스란

- 데이터를 보관하거나 전달하는 목적을 가진 **객체**를 만들때 사용한다 사용 `e.g. DTO`

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

- 데이터 클래스를 사용하면 컴파일러가 자동으로 만들어주는 함수들이 있다
  - `equals()`
  - `hashCode()`
  - `toString()`
  - `componentN()` , `copy()`

- 기존 자바에선 주로 Lombok을 사용

```
@Data
public class Person {
```

```
  private final String name;
  private final int age;
}
```

- JDK 15에서 record 라는 이름으로 추가됨

```
public record Person(String name, int age) {
}
```

- 코틀린 디컴파일 기능을 이용해 내부가 어떤지 확인해보자

---

2. 데이터클래스가 필요한 이유

- 데이터 저장을 목적으로 하는 클래스는 일반적으로 3가지 함수 toString, equals, hashCode를 재정의하는데 데이터 클래스를 사용하면 자동으로 생성해준다
- 일반 클래스에서 toString, equals, hashCode를 쓸 경우 직접 구현하거나 IDE를 통해 생성해야한다
- 그럼 이 세가지 함수가 각각 무슨 일을 하는지 간단히 살펴보자

2.1 객체 동등성 비교 (equals)

- 일반적으로 두개의 인스턴스의 동등성 비교를 위해 `equals`를 재정의한다
- 객체의 동등성 비교시 결과에 대한 차이 `일반 클래스`

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

fun main() {
    val person1 = Person(name = "tony", age = 12)
    val person2 = Person(name = "tony", age = 12)
    println(person1 == person2)
```

```
}
// false
```

- 객체의 동등성 비교시 결과에 대한 차이 `데이터 클래스`

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

fun main() {
    val person1 = Person(name = "tony", age = 12)
    val tony2 = Person(name = "tony", age = 12)
    println(person1 == person2)
}
// true
```

2.2 해시 코드 (hashCode)

- equals를 재정의할때 반드시 hashCode도 재정의한다
- JVM 언어 기준으로 객체 비교시 equals로 true를 반환하는 객체는 hashCode도 같아야한다
- 즉 equals가 true인데 hashCode가 다르다면 `Hash 계열 자료구조`에서 정상적으로 동작하지 않는다

- equals만 구현하고 hashCode는 구현하지 않은 사례

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

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

         other as Person

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

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

        return true
    }
}

fun main() {
    val person1 = Person(name = "tony", age = 12)
    val person2 = Person(name = "tony", age = 12)
    println(person1 == person2)
    // true

    val set = hashSetOf(person1)
    println(set.contains(person2))
    // false
}
```

- equals, hashCode가 모두 구현된 사례

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

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

        other as Person

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

        return true
    }

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

fun main() {
    val person1 = Person(name = "tony", age = 12)
    val person2 = Person(name = "tony", age = 12)
    println(person1 == person2)
    // true

    val set = hashSetOf(person1)
    println(set.contains(person2))
```

```
    // true
}
```

- 데이터 클래스를 사용

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

fun main() {
    val person1 = Person(name = "tony", age = 12)
    val person2 = Person(name = "tony", age = 12)
    println(person1 == person2)
    // true

    val set = hashSetOf(person1)
    println(set.contains(person2))
    // true
}
```

2.3 객체를 문자열로 표현 (toString)

- toString을 재정의하지 않았을때

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

fun main() {
    val person1 = Person(name = "tony", age = 12)
    println(person1.toString())
}
// Person@1d251891
```

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

fun main() {
    val person1 = Person(name = "tony", age = 12)
    println(person1.toString())
```

```
}
// Person(name=tony, age=12)
```

2.4 불변성을 유지하며 복사 (copy)

- 데이터 클래스의 `copy()`를 사용하면 객체의 불변성을 쉽게 유지할 수 있다
- `var`를 사용해 프로퍼티를 변경가능하도록 하면 불변이 아니다
- 불변성이 깨졌을때의 문제 점은
- 우선 불변성이 깨진다면 `Hash 계열 자료구조`에서 의도치 않은 버그가 발생할 수 있다

```
data class Person(var name: String, var age: Int)

fun main() {
    val person1 = Person(name = "tony", age = 12)

    val set = hashSetOf(person1) // setOf면 정상 동작

    println(set.contains(person1))
    // true

    person1.name = "strange"

    println(set.contains(person1))
    // false
}
```

- 두번째로 멀티-스레드 환경에서 객체의 불변성을 유지하는 것은 동기화 처리를 줄여주고 안정성을 유지하기 위해 중요하다
- 또한 유지보수 관점에서도 여러 소스에서 객체의 프로퍼티를 제 각각 변경하고 있으면 코드를 파악하는데 어려움이 있다
- 이런 이유로 기존 객체를 수정하는 것보다 새로운 객체로 복사해서 사용하는 것이 좋다

- copy를 사용하면 프로퍼티를 `val` 로 유지해 불변성을 유지하는데 도움이 된다
- 원하는 프로퍼티만 변경하면서 새로운 불변 객체를 생성할 수 있다

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

fun main() {
    val person1 = Person(name = "tony", age = 12)

    val person2 = person1.copy(name= "strange")

    println(person2.toString())
    // Person(name=strange, age=12)
}
```

- 디컴파일해보면 아래와 같이 인자를 전달받아 새로운 객체를 생성한다

```
@NotNull
 public final Person copy(@NotNull String name, int age) {
    Intrinsics.checkNotNullParameter(name, "name");
    return new Person(name, age);
 }
```

2.5 프로퍼티를 순서대로 가져온다(componentN)

- `componentN`은 데이터 클래스에 정의된 프로퍼티를 순서대로 가져올 수 있다

```
fun main() {
    val person1 = Person(name = "tony", age = 12)

    println("이름=${person1.component1()}, 나이=${person1.component2()}")
    // 이름=tony, 나이=12
}
```

- `구조분해할당` 을 사용해 좀 더 쉽고 안전하게 변수를 선언할 수 있다

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

fun main() {
    val person1 = Person(name = "tony", age = 12)

    val (name, age) = person1
    println("이름=${name}, 나이=${age}")
    // 이름=tony, 나이=12
}
```

- 디컴파일해보면 이와 같은 코드로 변경된다

```
public static final void main() {
      Person person1 = new Person("tony", 12);
      String name = person1.component1();
      int age = person1.component2();
      String var3 = "이름=" + name + ", 나이=" + age;
      System.out.println(var3);
   }
```

- 구조분해할당은 이후 `페어와 구조분해할당` 챕터에서 자세히 알아보기로 합니다