아이템2: 데이터 집합 표현에 data 한정자를 사용하라

데이터를 한꺼번에 전달할 때, 데이터 한정자를 붙인 클래스를 사용합니다.

자동 생성 함수들

toString, equals, hashCode, copy, componentN함수가 자동으로 생성된다.

- toString: 모든 프로퍼티와 값을 출력. 로그 출력할 때나 디버그할 때 유용하게 활용할 수 있습니다.
- equals: 기본 생성자의 프로퍼티가 같은지 확인할 수 있습니다.
- hashCode: equals와 같은 결과를 낸다(라고 적혀 있지만, 다르더라도 같은 값이 나올 수 있다.) 아이템 40 (6-5) 참고
- copy: immutable 데이터 클래스를 만들 때 편리하다.
    - 객체를 얕은 복사 합니다. immutable 데이터 클래스는 깊은 복사가 필요 없어 편리하게 사용할 수 있습니다.

```
fun copy(
  id: Int = this.id, // reference copy == aliasing != shallow copy
  name: String = this.name,
  points: Int = this.points
) = Player(id, name, points)  // create new object

val player1 = Player(...)
val player2 = player1.copy() // create object and Do referential copy
```

A shallow copy of an object is a copy whose properties share the same references (point to the same underlying values) as those of the source object from which the copy was made
→ 사본의 프로퍼티와 원본의 프로퍼티가 같은 레퍼런스를 공유

A **deep copy** of an object is a copy whose properties do not share the same references (point to the same underlying values) as those of the source object from which the copy was made.
→ 사본의 프로퍼티와 원본의 프로퍼티가 같은 레퍼런스를 공유하지 않음.

출처 : MDN

- componentN: 위치를 기반으로 객체를 해제할 수 있게 해준다. (destructuring)

```
val (id, name, pts) = player
val id:Int = player.component1()
val name:String = player.component2()
```

▼ destructuring

- 장점: 변수 이름을 원하는 대로 지정 가능, componentN만 있다면 원하는 형태로 객체 해제 가능 (구조 분해)

```
//이렇게도 쓰고
//list내에 componentN이 5까지 정의되어있음.
val (one,two,three) = listOf(1,2,3)


// 이렇게도 씁니다.
val numEng = mapOf(1 to "one" , 2 to "two")
for ((num,eng) in numEng){
  // something to do
}
```

- 단점: 위치를 잘못 지정하면 위험하다. 순서 혼동되는 문제는 굉장히 자주 발생한다

```
data class FullName(
  val firstName: String,
  val secondName:String,
```

```
  val lastName: String
)

val elon = FullName("elon", "Reeve", "musk")
val (name, surName) = elon
// name = elon, surName = Reeve, not musk
```

    - 할당할 변수의 이름과 프로퍼티 이름을 같게 사용하면, 순서가 바뀔 경우 IDE가 경고해줍니다.

- 값이 하나만 있는 경우 destructuring을 하지 않는 것이 좋다.
    - 특히 람다의 경우 언어마다 argument에 괄호를 붙이는 경우도 있고 아닌 경우도 있어서 주의 필요.

```
data class User(val name: String)
val (name) = User("John") // String
// 특히 람다의 경우

val user = User("John)
user.let{ a -> print(a)}   //User(name=John)유저를 출력
user.let{ (a) -> print(a)} //john을 출력
user.let{ ((a)) -> print(a)} // error
```

튜플 대신 데이터 클래스 사용하기

과거에는 (UInt, String, String, Long) 처럼 원하는 형태의 튜플을 정의할 수 있었고, 데이터 클래스와 같은 역할이었지만 가독성이 너무 나빴다. 결국 데이터 클래스를 사용하는 것이 더 좋았기 때문에 없이지게 되었다고 한다. Pair, Triple은 코틀린에 남아있는 마지막 튜플이다.

튜플을 사용하는 경우

몇 가지 지역적인 목적이 남아있다.

1. 값에 간단하게 이름을 붙이는 경우

```
val (description, color) = "cold" to Color.BLUE
```

2. 표준 라이브러리에 볼 수 있는 것 처럼 미리 알 수 없는 집합을 표현할 때
    - 왜 알 수 없다고 하는지 아세요..?

```
// 타입을 미리 알 수 없는 경우 일 것 같아요.
// 데이터 클래스로 만들 수도 있을 것 같긴 합니다. (근데 굳이 만들어야 하는지..?)
```

```
val (odd, even) = numbers.partition {it % 2 == 1}

val map = mapOf (1 to "San Francisco", 2 to "Seoul")
```

데이터 클래스를 사용하는 경우

```
fun String.parseName(): Pair<String, String>? {
// 복잡한 코드
 return Pair(firstName, lastName))
}

val fullName = "Goh Sejin"
val (firstName, lastName) = fullName.parseName() ?: return
```

위 코드의 문제점

1. 리턴 타입인 Pair<String,String>이 이름을 나타내는지 알기 어려움
2. 이름과 성중 어떤 것이 먼저인지 알기 어려움.

```
data class FullName(
  val firstName: String,
  val lastName: String
)

fun String.parseName(): FullName? {
  // 복잡한 코드
  return FullName(firstName, secondname)
}
val fullName = "Goh Sejin"
val (firstName, lastName) = fullName.parseName() ?: return
```

개선점

1. **함수 리턴타입이 명확**해진다.
2. **리턴 타입이 짧아지고 전달하기 쉬워진다**.
3. 데이터 클래스에 적혀 있는 것과 다른 이름을 활용해 변수를 해제하면 **경고가 출력된다**.

정리

데이터 클래스의 장점

1. 함수의 리턴타입이 명확해진다. (Pair<String,String> → FullName)
2. 함수의 리턴타입이 간결해진다.
3. 변수를 해제할 경우 경고가 출력된다.
4. 가시성 제한자를 사용할 수 있다