Kotlin 기초 및 Property

변 석우

경성대학교
http://www.ks.ac.kr

October 10, 2019

기초 : 함수 정의 (2.1)

- 함수 정의에 `fun` 키워드 사용
- 변수 `args`, 타입 `Array<String>`
- 함수 정의가 top level에서 이루어짐
- 문장의 끝을 알려주는 semicolon (;) 생략

```
fun main(args: Array<String>) {
    println("Hello, world!")
}
```

- 함수의 return 및 타입 (아래 `if` 는 expression)\

```
fun max(a: Int, b: Int): Int {
    return if (a > b) a else b
}

>>> println(max(1, 2))
2
```

기초 : 식 (expression)

- `if` 는 문장이 아니라 식(expression)
- loops (`for`, `do`, `do/while`) 외의 대부분은 식으로 표현
- 식으로 표현되는 함수

```
fun max(a: Int, b: Int): Int = if (a > b) a else b
```

- 식으로 정의된 함수에 대해서는 타입 추론을 적용하고, return 타입 생략

```
fun max(a: Int, b: Int) = if (a > b) a else b
```

기초 : 변수 정의 (2.1.3)

Mutable/Immutable 변수 선언

- `val` : immutable (Java의 `final`)
- `var` : mutable (Java의 non-final)
- 변수 선언 시 타입을 정의할 수도, 생략할 수도 있음 (타입 추론, Java의 `var`)

```
val str = "abc"
val num = 42
val num2 : Int = 42

var num3 : Int
num3 = 42
num3 = 44
num3 = "abc"  // error
```

기초 : String의 Formatting (2.1.4)

- $ 및 ${ } 기호를 사용하여 Strint literal을 표현함.

```
>>> var str = "abc"
>>> println ("Hello, $str!")
Hello, abc!
>>> println ("Hello, ${str}!")
Hello, abc!

>>> "Hello, " + "abc" + "!"      // Java Style
Hello, abc!
```

Property와 Field의 차이 (Encapsulation)

- OOP의 encapsulation의 원리에 따라, 클래스를 구성하는 각 구성원의 접근을 직접적인 (directly) 방법으로 하지 않고, 이를 감싸는 access 메소드 (getter와 setter)를 정의하고, 이 메소드를 통하여 접근하는 방법을 적용할 수 있다. OOP 에서는 이 방법이 선호되고 있다.
- 한 클래스에 속한 member 변수는 field라고 하며, field에 접근자(getter/setter)가 정의된 경우 이 field는 property라고 한다. 즉, property는 field 보다 더 추상화된 개념이다.
- Getter/setter는 경우에 따라 validation, exception 등의 기능을 갖도록 정의될 수도 있다. 이러한 기능을 갖는 getter/setter는 단순한 wrapper와는 다르다.
- Property를 이용함으로써, 클래스 내부의 구현 (implementation) 및 구체적인 부분에 변화가 있더라도, 이 클래스를 사용하는 클라이언트는 이런 변화에 독립적일 수 있다.

Encapsulation의 전형적인 코드

Java의 property : private field와 public getter/setter

```java
public class Person {
    private final String name;      // private field
    public Person(String name) {
        this.name = name;
    }
    public String getName() {       // public getter
        return name;
    }
}
```

Kotlin의 Default Property

```kotlin
class Person(val name: String)
```

- `final` 대신 `val`, default getter의 경우 생략됨.
- `public` 가 default 이므로 생략
- Constructor 정의 생략 (boilerplate)

Default Getter/Setter의 예 (아무런 정의를 하지 않을 때)

Kotlin에서 getter/setter 호출

```
class Person(val name: String, var isMarried: Boolean)

>>> val person = Person("Bob",true)  // no new
>>> println(person.name)             // call getter
Bob
>>> println(person.isMarried)        // call getter
true
>>> person.isMarried = false         // call setter
```

Java로 Kotlin의 Person 클래스 접근

```
>>> Person person = new Person("Bob", true);
>>> System.out.println(person.getName());   // getter: name
Bob
>>> System.out.println(person.isMarried()); // getter: isMarried
>>> person.setMarried(false)                // setter: isMarried
```

Kotlin의 Custom Property 정의

Property 정의의 문법적 구조

```
var <propertyName>[: <PropertyType>] [= <initializer>]
    [<getter>]
    [<setter>]
```

– Property 변수를 기준으로, 그에 대한 getter/setter가 정의됨.
– getter는 반드시 정의되어야 하며 (그렇지 않으면 custom propery를 정의할 이유가 없음), setter는 필요에 따라 정의될 수 있다.
– Property 변수는 대부분 mutable하며 (`var` 로 정의됨), immutable할 수도 있다. 또한, 이 값은 필요에 따라 초기화될 수도 있다.
– getter/setter 정의에서 property 변수에 대한 표현은 `field` 로서 표현되며, 이를 backing field라고 한다. 따라서 `field` 라는 이름은 특별한 의미를 가지므로, member 변수를 이 이름으로 정의할 수 없다.

Custom 접근자로 정의된 Property 예-1 (name)

```
class Address {var name = "Kotlin: " + "Addr1"}
class Address2 {
    var name: String = "Kotlin2: "
    get() { return field + "Addr2 }
    set(value) {
        if (value == "a") field = "b" else field = value
    }
}

fun main (args: Array<String>) {
    val addr1 = Address()
    val addr2 = Address2()

    println("addr1.name: ${addr1.name}")    // Kotlin: Addr2
    println("addr2.name: ${addr2.name}")    // Kotlin2: Addr2
    addr2.name = "a"
    println("addr2.name: ${addr2.name}")    // bAddr2
}
```

Custom 접근자로 정의된 Property 예-2 (isSquare)

– isSquare getter 정의에서, isSquare 값이 backing field를 사용없이 다른 변수들 만으로 결정됨.
– getter가 변수에 대한 단순한 wrapper와는 다르게 이용될 수 있음을 보여주는 예

```
class Rectangle(val height: Int, val width: Int) {
    val isSquare: Boolean
        get() {
            return height == width
        }
}

fun main(args: Array<String>) {
    val rectangle = Rectangle(41, 43)
    println(rectangle.isSquare)              // false
}
```