



fast campus

Chapter 02.코틀린 기초 - 07.클래스와 프로퍼티

클래스와 프로퍼티

1. 클래스

- `class` 키워드를 사용하여 클래스를 선언한다

```
class Coffee {  
    //...  
}
```

- 코틀린의 클래스는 본문을 생략할 수 있다

```
class EmptyClass
```

- 코틀린의 생성자는 기본 생성자와 하나 이상의 보조 생성자가 존재할 수 있다

```
class Coffee constructor(val name: String)
```

- 기본 생성자는 `constructor` 를 생략할 수 있다

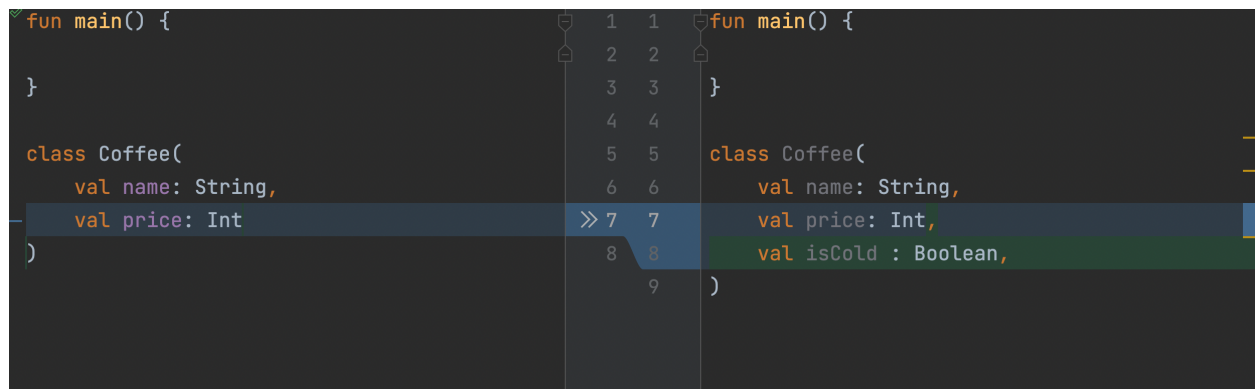
```
class Coffee(val name: String)
```

- 코틀린에선 클래스에 프로퍼티를 선언할때 `후행 심표` `trailing comma` 를 사용할 수 있다

```
class Coffee(  
    val name: String,  
    val price: Int, // trailing comma  
)
```

- 후행 심표를 쓰면 이전의 마지막 줄을 수정하지 않고 프로퍼티를 쉽게 추가할 수 있고 git에서 diff 등으로 코드를 비교했을때 변경사항을 명확히 알 수 있다.

- 후행 심표를 쓰지 않고 diff 했을때

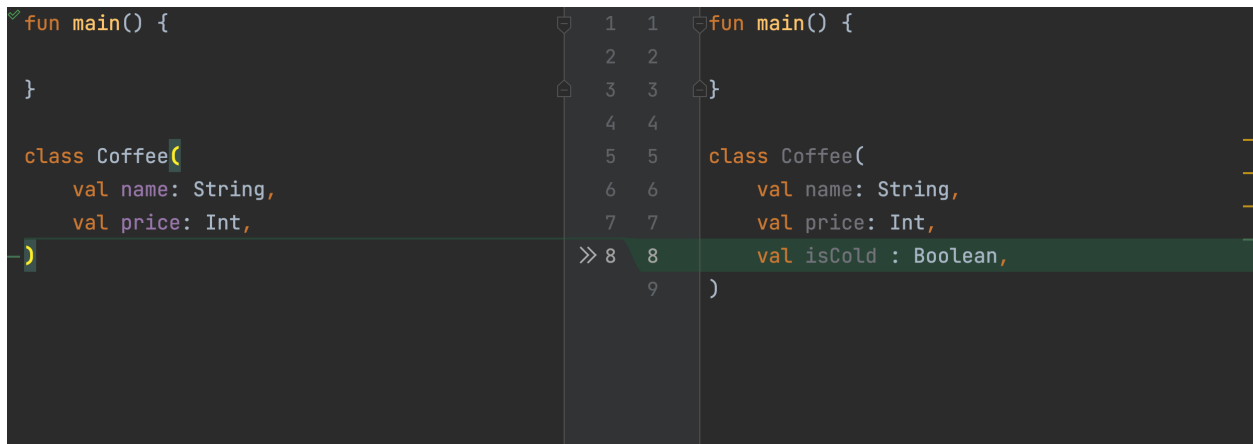


```

1 1 fun main() {
2 2
3 3 }
4 4
5 5 class Coffee(
6 6     val name: String,
7 7     val price: Int,
8 8     val isCold : Boolean,
9 9 )

```

- 후행 심표를 쓰고 diff 했을때



2. 프로퍼티

- 먼저 코틀린의 프로퍼티는 `val`, `var` 키워드를 모두 사용할 수 있다

```
class Coffee(  
    var name: String = "", // 기본 값 추가  
    var price: Int = 0,  
)
```

- 프로퍼티를 수정하거나 사용하려면 참조를 사용하면 된다

```
fun main() {  
    val coffee = Coffee()  
    coffee.name = "아이스 아메리카노"  
    coffee.price = 2000  
  
    println("${coffee.name} 가격은 ${coffee.price}")  
}
```

3. getter, setter

- 코틀린은 var로 선언된 프로퍼티는 `getter`, `setter` 를 자동으로 생성한다
- 아래의 코드는 실제론 필드의 setter를 사용해 값을 할당한다

```
coffee.name = "아이스 아메리카노"
```

- 해당 필드를 사용할시에는 `getter`를 사용한다

```
println("${coffee.name} 가격은 ${coffee.price}")
```

- val로 선언된 프로퍼티는 `getter` 만 존재한다
- 또한 코틀린은 커스텀 `getter`를 만들 수 있다

```
class Coffee(  
    var name: String = "",  
    var price: Int = 0, // trailing comma  
) {  
  
    val brand: String  
        get() = "스타벅스" // 커스텀 getter  
}  
  
fun main() {  
    val coffee = Coffee()  
    coffee.name = "아이스 아메리카노"  
    coffee.price = 2000  
  
    // brand를 포함해 출력  
    println("${coffee.brand} ${coffee.name} 가격은 ${coffee.price}")  
}
```

- var로 선언된 프로퍼티에 한하여 커스텀 setter를 만들 수 있다

```
class Coffee(
    var name: String = "",
    var price: Int = 0, // trailing comma
) {

    val brand: String
    get() {
        return "스타벅스"
    }

    var quantity: Int = 0
    set(value) { // 커스텀 setter
        if (value > 0) { // 수량이 0 이상인 경우에만 값을 할당
            field = value
        }
    }
}

fun main() {
    val coffee = Coffee()
    coffee.name = "아이스 아메리카노"
    coffee.price = 2000
    coffee.quantity = 1 // 주문 수량 추가

    // 수량을 포함해 출력
    println("${coffee.brand} ${coffee.name} 가격은 ${coffee.price} 수량은 ${coffee.quantity}")
}
```

- 코틀린은 getter, setter에서 `field` 라는 식별자를 사용해 필드의 참조에 접근하는 데 이를 `Backing Field` 에 접근한다고 합니다.
- 그럼 Backing Field가 필요한 이유가 무엇일까? 코틀린에서 프로퍼티에 값을 할당할때 실제로 setter를 사용하는데 이때 무한 재귀 즉 `StackOverflow` 가 발생할 수 있다

```
var quantity: Int = 0
    set(value) {
        if (value > 0) {
            quantity = value // 재귀 호출
        }
    }
```

- 또한 코틀린의 프로퍼티는 객체지향적이다. 기본적으로 객체지향에서 객체의 상태는 프로퍼티로 표현하고 행위는 메서드로 표현하는데 자바는 상태를 메서드로 나타냄

```
public class Java_Coffee {

    private boolean isIced;

    public boolean isIced() {
        return isIced;
    }

    public void setIced(boolean iced) {
        isIced = iced;
    }
}

class Barista {

    public static void main(String[] args) {
        Java_Coffee coffee = new Java_Coffee();
        coffee.setIced(true);

        if(coffee.isIced()) { // 상태를 메서드로 표현
            System.out.println("아이스 커피");
        }
    }
}
```

- 코틀린은 프로퍼티를 사용해 상태를 나타낼 수 있기 때문에 자바보다 객체지향적으로 코드를 작성할 수 있다

```
class Coffee(
    var name: String = "",
    var price: Int = 0, // trailing comma
    var iced: Boolean = false,
) {

    val brand: String
    get() {
        return "스타벅스"
    }

    var quantity: Int = 0
    set(value) {
        if (value > 0) {
            quantity = value
        }
    }
}
```

```

        }
    }
}

fun main() {
    val coffee = Coffee()
    coffee.name = "아이스 아메리카노"
    coffee.price = 2000
    coffee.quantity = 1
    coffee.iced = true

    if (coffee.iced) { // 프로퍼티
        println("아이스 커피")
    }
    println("${coffee.brand} ${coffee.name} 가격은 ${coffee.price} 수량은 ${coffee.quantity}")
}

```