Kotln 참고자료

Kotlin

Kotlin은 JetBrains라는 회사에서 만들어진 프로그래밍 언어로, JVM에서 기반한 언어입니다. IntelliJ를 만든 회사에서 개발하였으며, 자바와 완벽하게 호환되면서도 더 간결한 특징을 가지고 있습니다.

Kotlin 장점

1. **간결한 구문**: Kotlin의 구문은 Java에 비해 더 간결하고 표현력이 풍부하게 설계되어 있어 상용구 코드를 줄여 읽고 쓰기가 더 쉬워집니다.
2. **Null 안전성**: Kotlin은 null 허용 참조와 null 불가 참조를 구별하는 유형 시스템을 도입하여 널 포인터 예외를 방지하는 데 도움을 줍니다.
3. **상호 운용성**: Kotlin은 Java와 원활하게 작동하도록 설계되어 Java 코드를 호출하거나 그 반대로 호출하는 데 문제가 없습니다. 이는 Java에서 Kotlin으로 전환하거나 혼합 언어 환경에서 작업할 때 특히 유용합니다.
4. **함수형 프로그래밍**: Kotlin은 함수형 프로그래밍 기능을 지원하여 기능적이고 간결한 코드를 더 쉽게 작성할 수 있게 해줍니다.
5. **확장 함수**: Kotlin을 사용하면 소스 코드를 수정하지 않고도 기존 클래스에 새 함수를 추가할 수 있습니다. 이는 기존 API를 향상시키는 강력한 기능인 확장 함수를 사용하여 수행됩니다.
6. **스마트 캐스트**: Kotlin의 유형 시스템에는 유형 검사 후 자동으로 변수를 캐스트하는 스마트 캐스트가 포함되어 있어 코드에서 명시적인 캐스팅의 필요성이 줄어듭니다.

Kotlin 에서 출력

Kotlin에서 출력을 할 때, Java와는 다르게 세미콜론(;)을 붙이지 않아도 됩니다. Kotlin에서의 출력 방식은 다음과 같습니다:

```
println("Hello World") // 코틀린에서의 출력
```

```
System.out.println("Hello World"); // 자바에서의 출력
```

Kotlin에서 변수 선언

Kotlin에서 변수를 선언할 때는 `var`와 `val` 키워드를 사용합니다. 이 두 키워드는 변수의 수정 가능 여부를 나타냅니다. 여기에 Kotlin에서의 변수 선언 방식을 설명하겠습니다:

```
val person = Person("짱구", 10)

val log = "사람의 이름은 ${person.name} 이고 나이는 ${person.age} 세 입니다."

prinln(log)
```

- `var`: 가변 변수를 선언합니다. 이 변수의 값은 언제든지 수정할 수 있습니다.
- `val`: 불변 변수를 선언합니다. 이 변수의 값은 한 번 초기화되면 수정할 수 없습니다. 재할당이 불가능합니다.

코틀린에서는 변수를 사용하기 전에 반드시 초기값을 할당해주어야 합니다. 또한, `val`을 사용하여 변수를 선언하면 해당 변수는 한 번만 초기화할 수 있습니다.

Kotlin에서의 변수 선언은 Java와 달리 명시적으로 수정 가능 여부를 표시하므로 코드의 가독성을 높이고 버그를 줄일 수 있습니다.

Kotlin에서 변수 선언

Kotlin에서 변수를 선언할 때는 `var`와 `val` 키워드를 사용합니다. 이 두 키워드는 변수의 수정 가능 여부를 나타냅니다. Kotlin에서의 변수 선언 방식을 설명하겠습니다:

```
long number1 = 10L; //long 타입 변수 선언

final long number2 = 10L; //final 선언

Long number3 = 1_000L; //객체 타입 선언

Person person = new Person("재롱이"); //클래스 선언

<자바에서의 변수 선언>
```

```
fun main() {

    var number1 = 10L //변수 선언 (가변)

        number1 = 5L // 가능
```

```
    val number2 = 20L //변수 선언 (불변)

        number2 = 10L //불가능
}
<코틀린에서의 변수 선언>
```

- `var`: 가변 변수를 선언합니다. 이 변수의 값은 언제든지 수정할 수 있습니다.
- `val`: 불변 변수를 선언합니다. 이 변수의 값은 한 번 초기화되면 수정할 수 없습니다. 재할당이 불가능합니다.

코틀린에서는 변수를 사용하기 전에 반드시 초기값을 할당해주어야 합니다. 또한, `val`을 사용하여 변수를 선언하면 해당 변수는 한 번만 초기화할 수 있습니다.

Kotlin에서의 변수 선언은 Java와 달리 명시적으로 수정 가능 여부를 표시하므로 코드의 가독성을 높이고 버그를 줄일 수 있습니다.

Kotlin 에서 연산자 관련

Java와 Kotlin은 객체 동일성과 객체 동등성을 비교하는 데에 다른 연산자와 메서드를 사용합니다.

	동일성	동등성
자바	==	equals
코틀린	===	==

Java:

- 객체 동일성(Reference Equality)을 비교할 때는 `==` 연산자를 사용합니다.
- 객체 동등성(Structural Equality)을 비교할 때는 `equals` 메서드를 호출합니다.

Kotlin:

- 객체 동일성을 비교할 때는 `===` 연산자를 사용합니다.
- 객체 동등성을 비교할 때는 `==` 연산자를 사용합니다. 이 연산자는 내부적으로 `equals` 메서드를 호출합니다.

따라서 Kotlin에서 `==` 연산자를 사용하면 간접적으로 `equals` 메서드가 호출됩니다. Kotlin에서 이러한 방식을 통해 코드를 더 간결하게 유지하고 가독성을 높이며, 객체 동등성을 비교할 때 명시적으로 `equals`를 호출할 필요가 없습니다.

Kotlin 에서 타입 선언

코틀린에서는 변수를 선언할 때 타입을 명시하거나, 초기값을 지정하여 변수를 생성할 수 있습니다. 아래는 코틀린에서의 타입 선언과 관련된 예제와 설명입니다:

```
var number3: Long = 30L // 타입과 값을 동시에 명시

var number4: Long // 초기값을 지정하지 않고 타입만 명시

var number5 //타입과 초기값을 모두 명시하지 않아 컴파일러가 타입을 추론할 수 없음

var number6: Int //초기값을 지정하지 않았으므로 사용할 수 없음
```

- 변수를 선언할 때 초기값을 함께 지정할 수 있으며, 이 경우 컴파일러는 변수의 타입을 추론합니다.
- 변수를 선언할 때 초기값을 지정하지 않고 타입만 명시하면, 나중에 초기값을 할당할 수 있습니다.
- 변수를 선언할 때 타입과 초기값을 모두 명시하지 않으면 컴파일러는 타입을 추론할 수 없으므로 에러가 발생합니다.
- 변수를 선언한 후 초기값을 지정하지 않으면 나중에 사용할 때 에러가 발생합니다.

코틀린에서는 변수를 사용하기 전에 초기값을 할당하거나 타입을 명시해주어야 하며, 초기값을 나중에 할당하려면 변수를 먼저 선언한 후에 할당해야 합니다.

Kotlin 에서의 기본 타입과 참조 타입

```
long number1 = 10L;

Long number3 = 1_000L;
```

```
var number1:Long =10L

var number3:Long =1_000L
```

Kotlin은 기본 타입과 참조 타입을 구분하지 않습니다. Java와는 다르게 Kotlin에서는 모든 데이터 타입이 참조 타입으로 취급됩니다. 그러나 Kotlin 컴파일러는 코드를 최적화하여 필요한 경우에 기본 타입(Primitive Type)으로 변환하여 성능을 향상시킵니다.

예를 들어, Kotlin에서 `Long` 타입 변수를 선언하고 값을 할당하면 내부적으로는 `long` (기본 타입)으로 처리됩니다. Kotlin은 이러한 타입 변환을 자동으로 처리하여 개발자가 명시적으로 타입을 변환할 필요가 없도록 합니다.

따라서 Kotlin에서는 기본 타입과 참조 타입을 명시적으로 구분하지 않고, 기본 타입의 특징과 참조 타입의 편리함을 함께 제공합니다. 이로 인해 코드 작성이 더 간결해지고 가독성이 향상되며, 동시에 성능을 유지할 수 있습니다.

Kotlin 에서의 null

Kotlin은 기본적으로 null을 허용하지 않는 안전한 언어입니다. 변수가 null을 가질 수 있는 경우에는 해당 변수의 타입 뒤에 `?`를 붙여야 합니다. 아래는 예제와 설명입니다:

```
var number1 = 10L

// 아래 주석 처리된 코드는 에러를 발생시킵니다.
// var number3 = 1_000L
// number3 = null

var number: Long? = 1_000L
number = null // 이렇게 null을 할당할 수 있습니다.
```

- Kotlin에서는 변수를 선언할 때 기본적으로 null을 허용하지 않습니다.
- 변수가 null을 가질 수 있는 경우에는 타입 뒤에 `?`를 붙여야 합니다. 이렇게 하면 해당 변수에 null 값을 할당할 수 있습니다.

이러한 접근 방식은 Kotlin에서 안전한 null 처리를 촉진하며, `NullPointerException`을 줄이는 데 도움을 줍니다.

Kotlin 에서의 객체 인스턴스

Kotlin에서 객체를 인스턴스화할 때는 `new` 키워드를 사용하지 않습니다. 다음은 Kotlin에서 객체를 생성하는 예제입니다:

```
var person= Person("김종훈")
```

Kotlin에서는 클래스의 생성자를 호출하여 객체를 생성할 수 있으며, `new` 키워드를 사용할 필요가 없습니다. 이렇게 코드를 작성하면 객체가 생성되고 초기화됩니다.

Kotlin 에서의 null 체크

Kotlin에서는 기본적으로 타입에 null 값을 포함하지 않기 때문에, 타입 뒤에 `?`를 붙여 명시적으로 null을 허용하도록 해야 합니다. 아래는 Kotlin에서의 null 체크와 관련된 코드 예제입니다:

```
public boolean startsWithsWith1(String str) {

if (str == null) {
throw new IllegalArgumentException("Null이 들어왔습니다.")
```

```
}
return str.startsWith("A");
}


public Boolean startsWithsWith2(String str) {

if (str == null) {
return null;
}
return str.startsWith("A");
}


public boolean startsWithsWith3(String str) {

if (str == null) {
return false;
}
return str.startsWith("A");
}
```

<자바에서 null 체크 방법>

```
fun startsWithsWith1(str: String?): Boolean {
if (str == null) {
throw IllegalArgumentException("Null이 들어왔습니다.")
}
return str.startsWith("A");
}


fun startsWithsWith2(str: String?): Boolean? {

if (str == null) {
return null;
}
return str.startsWith("A");
}



fun startsWithsWith3(str: String?): Boolean {

if (str == null) {
return false;
}
```

```
return str.startsWith("A");
}
```

<코틀린에서 null 체크 관련 코드>

- Kotlin에서는 변수나 함수의 파라미터, 반환 값 등을 선언할 때 해당 값이 null을 허용하는지 여부를 `?`를 사용하여 명시적으로 표현해야 합니다.
- `str: String?`에서 `?`는 `str` 변수가 null일 수 있다는 것을 나타냅니다.
- `Boolean?`는 `Boolean` 타입이거나 null일 수 있는 값을 나타냅니다.
- 이러한 접근 방식은 Kotlin에서 안전한 null 처리를 강조하며 NullPointerException을 방지하는 데 도움을 줍니다.

Kotlin Safe Call 연산자

```
val str: String? = "ABC"

// 아래 코드는 컴파일 에러를 발생시킵니다.
str.length // 불가능

// Safe Call 연산자를 사용하면, str이 null이 아닌 경우에만 length를 호출합니다.
val length = str?.length // 가능

// str이 null이면 length에는 null이 할당됩니다.
println(length) // 출력: null
```

- Safe Call 연산자(`?.`)를 사용하면 변수가 null인지 아닌지를 검사하고, null이면 연산을 중단하고 그대로 null을 반환합니다.
- Safe Call 연산자를 사용하면 코드가 더 안전하게 null 처리됩니다. 이를 통해 `NullPointerException`을 방지하고 안정성을 높일 수 있습니다.

Kotlin의 Safe Call 연산자는 nullable한 변수를 다룰 때 유용하게 사용되며, 안전한 코드 작성에 기여합니다.

Kotlin Elvis 연산자

Kotlin에서 Elvis 연산자(`?:`)는 변수가 `null`인지 아닌지를 검사하고, 변수가 `null`이 아닌 경우 왼쪽 식을 실행하며, 변수가 `null`인 경우에는 오른쪽 식을 실행하는 연산자입니다. 아래는 Kotlin Elvis 연산자의 예제와 설명입니다:

```
val str: String? = "ABC"

// Elvis 연산자를 사용하면, str이 null이 아니면 str.length를 반환하고, null이면 0을 ㅂ
```

```
val length = str?.length ?: 0

println(length) // 출력: 3
```

- Elvis 연산자(`?:`)는 변수가 `null`인지 아닌지를 검사하고, 변수가 `null`이 아닌 경우 왼쪽 식을 실행합니다.
- 변수가 `null`인 경우에는 오른쪽 식을 실행하며, 그 결과를 반환합니다.

Kotlin의 Elvis 연산자는 변수가 `null`인 경우에 대한 기본 값을 설정할 때 유용하게 사용됩니다. 이를 통해 안전한 기본값 설정 및 코드의 가독성을 향상시킬 수 있습니다.

Kotlin Safe Call과 Elvis 연산자 예시

Kotlin에서 Safe Call(`?.`)과 Elvis(`?:`) 연산자를 함께 사용하여 예외 처리를 간결하게 할 수 있습니다.

```
fun startsWithsWith1(str: String?): Boolean {
return str?.startsWith("A") ?: throw IllegalArgumentException("Null이 들어왔습니다
}

fun startsWithsWith2(str: String?): Boolean? {
return str?.startsWith("A")
}


fun startsWithsWith3(str: String?): Boolean {
return str?.startsWith("A") ?: false
}
```

- `startsWithsWith1` 함수는 `str`이 `null`이면 예외를 던지고, 그렇지 않으면 `startsWith` 메서드를 호출하여 결과를 반환합니다.
- `startsWithsWith2` 함수는 `str`이 `null`이면 `null`을 반환하고, 그렇지 않으면 `startsWith` 메서드를 호출하여 결과를 반환합니다.
- `startsWithsWith3` 함수는 `str`이 `null`이면 `false`를 반환하고, 그렇지 않으면 `startsWith` 메서드를 호출하여 결과를 반환합니다.

Kotlin의 Safe Call과 Elvis 연산자를 함께 사용하면 예외 처리를 간결하게 할 수 있으며, 코드의 가독성을 향상시킬 수 있습니다.

Kotlin non-null 단정자

Kotlin에서 Non-null 단정자(`!!`)는 변수에 할당된 값이 `null`이 아님을 확실히 단언하므로, 컴파일러가 null 검사를 수행하지 않도록 합니다. 그러나 만약 변수가 실제로 `null`이라면 `NullPointerException`이 발생합니다. 따라서 Non-null 단정자는 변수가 확실히 `null`이 아닌 경우에만 사용해야 합니다. 아래는 예제와 설명입니다:

```
var str1: String? = null

// Non-null 단정자를 사용하여 str1이 null이 아님을 단언합니다.
var len = str1!!.length

// str1이 실제로 null이므로 아래 코드는 NullPointerException을 발생시킵니다.
println(len)
```

- Non-null 단정자(`!!`)를 사용하면 변수에 할당된 값이 확실히 `null`이 아님을 단언합니다.
- 그러나 Non-null 단정자를 사용할 때는 해당 변수가 실제로 `null`이 아닌 경우에만 사용해야 합니다.
- 변수가 `null`인 경우에 Non-null 단정자를 사용하면 `NullPointerException`이 발생합니다.

Kotlin에서 Non-null 단정자는 변수가 null이 아님을 확실히 알고 있을 때만 사용해야 하며, null 안전성을 보장하기 위해 주의해서 사용해야 합니다.

Java → Kotlin 변환 시 주의사항

인텔리제이(IntelliJ IDEA)를 사용하여 Java 코드를 Kotlin으로 변환할 때, 주의해야 할 몇 가지 사항이 있습니다. 특히, null 관련 정보가 없는 경우에 대한 변환에 주의해야 합니다. 아래는 Java에서 Kotlin으로의 변환 시 주의사항과 설명입니다:

1. **@Nullable 및 @NonNull 어노테이션 처리:** Java 코드에 `@Nullable` 또는 `@NonNull`과 같은 어노테이션을 사용하여 null 관련 정보를 나타내고 있다면, Kotlin으로 변환될 때 `?` 또는 `!!`를 적절하게 사용합니다. 예를 들어, `@Nullable`이 있는 필드는 Kotlin에서 nullable 한 타입으로 변환될 것이며, `@NonNull`이 있는 필드는 non-null 타입으로 변환될 것입니다.
2. **자바에서는 기본 타입:** Java에서 사용되는 기본 타입(예: `int`, `boolean`)은 Kotlin에서 자동으로 Kotlin 기본 타입으로 변환됩니다. 따라서 필요한 경우 Kotlin 기본 타입을 사용하고, null 처리를 위해 기본 타입을 nullable로 선언해야 하는 경우 `?`를 추가해야 합니다.
3. **컬렉션 타입 변환:** Java에서 Kotlin으로 변환할 때, 컬렉션 타입도 자동으로 변환됩니다. Java의 `List`는 Kotlin에서 `List`로 유지되며, `Map`은 `Map`으로 변환됩니다.
4. **Getter 및 Setter 메서드 변환:** Java의 Getter와 Setter 메서드는 Kotlin의 프로퍼티로 변환됩니다. 따라서 필드에 직접 접근하는 대신 프로퍼티를 사용하게 됩니다.
5. **스태틱 멤버 변환:** Java의 `static` 멤버는 Kotlin의 `companion object`로 변환됩니다.
6. **컴파일러 경고:** Java에서 Kotlin으로의 변환 후 컴파일 시 경고가 발생할 수 있습니다. 이는 Java와 Kotlin의 차이로 인해 발생할 수 있으며, 경고를 해결하기 위해 코드를 수정해

야 할 수 있습니다.

7. **Kotlin의 null 안전성 이점 활용:** Kotlin은 null 안전성을 강조하는 언어이므로, Java 코드를 Kotlin으로 변환할 때 null 처리에 주의를 기울여야 합니다. Nullable 타입을 올바르게 다루고, Elvis 연산자 및 Safe Call 연산자를 활용하여 NullPointerException을 방지할 수 있습니다.

변환된 Kotlin 코드를 검토하고 필요한 경우 null 관련 정보를 추가하고, 안전성을 확보하기 위한 수정을 수행해야 합니다. 코드 변환은 자동화될 수 있지만, 완벽한 변환을 위해서는 수동으로 일부 수정이 필요할 수 있습니다.

Kotlin 기본 타입 변환

Kotlin에서는 타입 변환을 위해 `to변환타입()` 형식의 함수를 사용합니다. 변수가 `null` 값을 가질 수 있는 경우, 변환 시에 적절한 처리를 해야 합니다. 아래는 Kotlin에서의 기본 타입 변환과 null 처리에 관한 예제와 설명입니다:

```
val number1 = 3 // Int로 선언됩니다.
val number2: Long = number1.toLong()

val number3: Int? = 3 // Nullable한 Int로 선언됩니다.
val number4: Long = number3?.toLong() ?: 0L
```

- `number1`은 선언 시 타입이 지정되지 않았지만, 기본적으로 `Int`로 처리됩니다.
- `number2`는 `number1`을 `toLong()` 함수를 사용하여 `Long`으로 변환한 결과입니다.
- `number3`는 `Int?`로 선언되어 Nullable한 Int 값을 가집니다.
- `number4`는 Safe Call 연산자(`?.`)와 Elvis 연산자(`?:`)를 사용하여 `number3`을 `Long`으로 변환하며, 값이 `null`인 경우 0L로 기본값을 설정합니다.

Kotlin에서는 타입 변환과 null 처리를 안전하게 다룰 수 있도록 다양한 기능을 제공하며, 코드 안정성을 높일 수 있습니다.

Kotlin 타입 캐스팅

Kotlin에서 타입 캐스팅은 `is`와 `as` 연산자를 사용하여 수행합니다. 아래는 Kotlin에서의 타입 캐스팅과 스마트 캐스트에 관한 예제와 설명입니다:

```
//1.자바에서의 타입 캐스팅
public static void printAgeIfPersion(Object obj){
if(obj instanceof Person){
        Person person  = (Person) obj;
```

```
        System.out.println(person.getAge());
    }
}

//2.코틀린에서의 타입 캐스팅
fun prinAgeIfPerson(obj:Any){
    if(obj is Person){
            val person = obj as Person
            println(person.age)
    }
}

//2번 코틀린 코드를 스마스 캐스팅을 통해서 생략한 코드

fun prinAgeIfPerson(obj:Any){
    if(obj is person){
        println(obj.age)
    }
}

//2번 코틀린 코드에 null이 들어올수 있을 때
fun prinAgeIfPerson(obj:Any?){
    if(obj is person){
        val person = obj as? person
            println(person?.age)
    }
}
```

- `is` 연산자를 사용하여 객체의 타입을 체크할 수 있습니다. `obj is Person`은 `obj`가 `Person` 타입인지를 검사합니다.
- `as` 연산자를 사용하여 객체를 원하는 타입으로 캐스팅할 수 있습니다. `val person = obj as Person`은 `obj`를 `Person` 타입으로 변환합니다.
- `as?`를 사용하여 안전한 타입 캐스팅을 수행할 수 있습니다. `val person = obj as? Person`은 `obj`를 `Person` 타입으로 변환하며, 만약 실패하면 `null`을 반환합니다.

스마트 캐스트는 Kotlin의 강력한 기능 중 하나로, 타입 체크 이후에 자동으로 객체를 원하는 타입으로 변환해주어 코드를 간결하게 만듭니다.

Kotlin에서만 존재하는 Type

Kotlin은 Java와 달리 몇 가지 고유한 타입을 제공합니다. 이러한 타입은 Kotlin의 특징을 이해하고 코드를 더 명확하게 작성하는 데 도움을 줍니다.

1. **Any**
    - `Any`는 Kotlin의 최상위 타입으로, 모든 객체의 부모입니다.

- 모든 기본 타입(`Primitive Type`)의 최상위 타입도 `Any` 입니다. 따라서 `long`과 `Long`을 구분하지 않습니다.
- `Any` 자체로는 `null`을 포함할 수 없습니다. `null`을 포함하려면 `Any?`로 표현합니다.
- `Any`에는 `equals`, `hashCode`, `toString` 등의 메서드가 존재합니다.

2. **Unit**
   - `Unit`은 Java의 `void`와 동일한 역할을 합니다. Kotlin에서는 타입 추론이 가능하므로 함수의 반환 타입으로 `Unit`을 생략할 수 있습니다.
   - `void`와 달리 `Unit`은 그 자체로 타입 인자로 사용 가능합니다.
   - 함수형 프로그래밍에서 `Unit`은 단 하나의 인스턴스만을 갖는 타입을 의미하며, 실제로 존재하는 타입입니다.
3. **Nothing**
   - `Nothing`은 함수가 정상적으로 끝나지 않았다는 사실을 표현하는 데 사용됩니다.
   - 주로 예외를 던지는 함수나 무한 루프를 가진 함수 등에서 활용됩니다.
   - `Nothing`은 함수가 예외를 던지거나 무한 루프에 빠지기 때문에 함수의 반환 타입으로 사용됩니다.

Kotlin은 이러한 고유한 타입을 활용하여 코드를 더 간결하고 안전하게 작성할 수 있도록 도와줍니다.

Kotlin 배열

배열 예시

```
fun main() {

    // Int형으로 1, 2, 3, 4 배열 생성
    val intArr: Array<Int> = arrayOf(1, 2, 3, 4)

    // 타입 생략 가능
    val intArr2 = arrayOfNulls<Int>(5)

    // Any는 모든 데이터 타입을 포함할 수 있는 최상위 타입
    val anyArr: Array<Any> = arrayOf(1, "awd", 3.2, 4)

    println(intArr[0]) // 1 출력
    println(intArr2[1]) // null 출력
    println(anyArr[1]) // "awd" 출력
}

출력
1
null
```

```
awd
```

- Kotlin에서 배열은 `Array` 클래스를 사용하여 생성합니다.
- 배열의 요소 타입은 `<Type>` 형식으로 명시합니다. 예를 들어, `Array<Int>`는 `Int` 타입의 배열을 나타냅니다.
- 요소의 타입을 생략할 수 있으며, Kotlin은 컴파일러를 통해 타입을 추론합니다.
- `arrayOfNulls` 함수를 사용하여 지정된 크기의 null로 초기화된 배열을 생성할 수 있습니다.
- `Any`는 모든 데이터 타입을 포함할 수 있는 최상위 타입으로, 배열에 다양한 타입의 요소를 포함할 수 있습니다.

Kotlin에서 배열을 사용할 때 자바와 비슷한 문법을 사용하지만, 타입 추론과 표현력 있는 문법을 활용하여 코드를 간결하게 작성할 수 있습니다.

Kotlin 조건문

Kotlin 조건문 예시

```
fun main(){
    var a = 7
    if(a > 6){
        println(a)
    } else{
        print("exit")
    }
}

출력:
7
```

- Kotlin의 `if` 조건문은 Java와 유사하게 동작합니다.
- 조건식이 참인 경우 `if` 블록이 실행되고, 거짓인 경우 `else` 블록이 실행됩니다.
- Kotlin에서는 블록 내의 마지막 표현식이 해당 블록의 결과값으로 자동으로 반환되므로 `return` 키워드를 사용하지 않아도 됩니다.

Kotlin의 조건문은 Java와 비슷하게 사용할 수 있으며, Kotlin의 표현력 있는 문법을 활용하여 더 간결하고 가독성 있는 코드를 작성할 수 있습니다.

Kotlin에서의 조건문과 반환

Kotlin에서는 `if` 조건문을 표현식으로 사용하여 더 간결하게 코드를 작성할 수 있습니다. 아래는 예제와 설명입니다:

```
// 기존 방식
fun getPassOrFail(score: Int): String {
    if (score >= 50) {
        return "P"
    } else {
        return "F"
    }
}
```

```
// Kotlin에서의 간결한 방식
fun getPassOrFail(score: Int): String {
    return if (score >= 50) {
        "P"
    } else {
        "F"
    }
}
```

- Kotlin에서는 `if` 조건문을 표현식으로 사용할 수 있습니다. 이는 조건식의 결과에 따라 `if` 블록 또는 `else` 블록 중 하나가 반환됩니다.
- 함수에서 `if` 표현식을 직접 반환할 수 있으므로, `return` 키워드를 사용하지 않고 간결하게 코드를 작성할 수 있습니다.
- Kotlin에서는 삼항 연산자가 없습니다. 대신 `if` 표현식을 활용하여 값을 반환합니다.

Kotlin의 `if` 표현식을 적극적으로 활용하면 코드를 간결하게 유지할 수 있으며, Java와 비교하여 가독성이 향상됩니다.

Kotlin에서의 조건문과 반환

```
// 기존 방식
fun getPassOrFail(score: Int): String {
    if (score >= 50) {
        return "P"
    } else {
        return "F"
    }
}

// Kotlin에서의 간결한 방식
fun getPassOrFail(score: Int): String {
    return if (score >= 50) {
        "P"
```

```
    } else {
        "F"
    }
}
```

- Kotlin에서는 `if` 조건문을 표현식으로 사용할 수 있습니다. 이는 조건식의 결과에 따라 `if` 블록 또는 `else` 블록 중 하나가 반환됩니다.
- 함수에서 `if` 표현식을 직접 반환할 수 있으므로, `return` 키워드를 사용하지 않고 간결하게 코드를 작성할 수 있습니다.
- Kotlin에서는 삼항 연산자가 없습니다. 대신 `if` 표현식을 활용하여 값을 반환합니다.

Kotlin의 `if` 표현식을 적극적으로 활용하면 코드를 간결하게 유지할 수 있으며, Java와 비교하여 가독성이 향상됩니다.

Kotlin에서의 조건문과 범위 표현

```
//자바에서의 조건문
public void validateScoreIsNotNegative(int score){
if(0 <= score && score <= 100){
      // 범위 내의 값일 때 처리
    }
}
```

```
//코틀린에서의 조건문
fun validateScoreIsNotNegative(score: Int) {
    if (score in 0..100) {
        // 범위 내의 값일 때 처리
    }
}
```

- Kotlin에서는 `in` 연산자를 사용하여 범위를 표현할 수 있습니다. `0..100`은 0부터 100까지의 범위를 나타냅니다.
- 조건문을 간결하게 표현할 수 있으며, 가독성이 높아집니다.
- 범위 내에 속하는 값을 검사할 때 사용합니다. `score`가 0부터 100 사이인지 여부를 간단하게 확인할 수 있습니다.

Kotlin의 범위 표현은 코드를 더 직관적으로 만들어주므로 다양한 상황에서 활용할 수 있습니다.

Kotlin에서의 `when` 표현식

```
when (조건을검사할 값) {

    case1 -> 동작1
    case2 -> 동작2
    else -> 동작3
}
```

- `when`은 Kotlin의 다양한 조건 처리를 위한 표현식입니다.
- 조건을 검사할 값이 `when` 키워드 뒤에 위치하며, 이 값이 각 분기에서 비교됩니다.
- 여러 개의 분기 조건문을 지원하며, 다양한 타입의 데이터를 비교할 수 있습니다.

Kotlin의 `when` 표현식은 간결하면서도 다양한 상황에서 활용할 수 있으며, 코드의 가독성을 높이는데 도움이 됩니다.

Kotlin `when` 표현식의 다양한 활용 예시

```
fun getGradeWithSwitch(score: Int): String {
    return when (score / 10) {
    9 -> "A"
    8 -> "B"
    7 -> "C"
    else -> "D"
    }
}


fun getGradeWithSwitch2(score: Int): String {
    return when (score) {
    in 90..99 -> "A"
    in 80..89 -> "B"
    in 70..79 -> "C"
    else -> "D"
    }
}

fun judgeNumber(score: Int) {
    when (score) {
    in -1..1 -> println("-1 , 0, 1에 속하는 숫자입니다.")
    else -> println("아닙니다")
    }
}


fun judgeNumber2(score: Int) {
    when {
```

```
    score == 0 -> println("주어진 숫자는 0입니다.")
    score % 2 == 0 -> println("주어진 숫자는 짝수입니다.")
    else -> println("주어진 숫자는 홀수입니다.")
    }
}
```

- 첫 번째 `when` 예시에서는 `score`를 10으로 나눈 결과에 따라 학점을 반환합니다.
- 두 번째 `when` 예시에서는 `score`가 범위에 따라 학점을 반환합니다.
- 세 번째 `when` 예시에서는 `score`가 범위에 따라 주어진 숫자에 속하는지 여부를 판단합니다.
- 네 번째 `when` 예시에서는 `score`의 조건에 따라 주어진 숫자의 특성을 출력합니다.

Kotlin의 `when` 표현식은 다양한 상황에서 조건을 처리하고 코드를 간결하게 유지하는 데 도움이 됩니다.

Kotlin `when` 표현식의 값 비교 예시

```
fun main() {
    val x = 5// 처음 x값에 5를 할당한다.

    when (x) { //when 문에서 x가 무슨 값을 가지느냐에 따라 결과가 달라진다.
        1 -> println("x는 1이다")  //만약 x가 1일 경우, 'x는 1이다'가 출력된다.
        2, 3 -> println("x는 2 또는 3이다")//x가 2나 3일 경우,
        in 4..10 -> println("x는 4와 10 사이에 있다")//4부터 10까지의 범위에 속하는경
        else -> println("x는 다른 수이다")//모든 조건에 해당하지 않는 경우
    }
}
```

[출력결과]

```
x는 4와 10 사이에 있다
```

- 이 예시에서는 `when` 문을 사용하여 변수 `x`의 값에 따라 다른 결과를 출력합니다.
- `when`의 조건 중 하나에 해당하는 경우 해당 결과가 출력됩니다.
- `in` 키워드를 사용하여 범위에 속하는지 확인할 수 있습니다.
- `else` 블록은 모든 조건에 해당하지 않는 경우에 실행됩니다.

Kotlin의 `when` 표현식을 사용하면 다양한 조건을 처리할 수 있으며 코드를 간결하게 유지할 수 있습니다.

when **타입 비교**

다음 코드에서는 `when` 문을 사용하여 변수의 타입을 비교합니다.

```
fun main() {
    val data: Any = "Hello, World!"

    when (data) {
        is String -> println("data is a String: $data")
        is Int -> println("data is an Int: $data")
        is Boolean -> println("data is a Boolean: $data")
        else -> println("data is something else: $data")
    }
}
```

[출력결과]

```
data is a String: Hello, World!
```

- `data` 변수는 `Any` 타입으로 선언되어 다양한 타입의 값이 할당될 수 있습니다.
- 현재는 "Hello, World!"라는 문자열이 `data`에 할당되어 있습니다.
- `when` 문은 `data` 변수의 타입을 확인하고 분기 처리합니다.
- `is` 키워드를 사용하여 변수의 타입을 확인합니다.
- 만약 `data`의 타입이 `String`이면 "data is a String: Hello, World!"가 출력됩니다.
- 만약 `data`의 타입이 `Boolean`이면 "data is a Boolean: Hello, World!"가 출력됩니다.
- 위 코드는 `data` 변수가 `String` 타입이기 때문에 "data is a String: Hello, World!"가 출력됩니다.

Kotlin 반복문

`for-each` 문 예시

```
val numbers = listOf(1, 2, 3)

for (i in numbers) {
    println(numbers)
}
```

- Kotlin에서 `for-each` 루프는 `in` 키워드를 사용하여 구현됩니다.
- 위 예시에서는 `numbers` 리스트의 각 요소를 반복하며 각 요소를 `number` 변수에 할당하여 출력합니다.
- 이것은 자바의 `for-each`와 유사한 방식으로 작동합니다.

Kotlin의 `for-each` 문을 사용하면 반복 작업을 보다 간편하게 수행할 수 있습니다.

기본 `for` 문

```
for(i: Int in 1..10)
    print(i)
```

- Kotlin에서 기본 `for` 문은 `in` 키워드를 사용하여 범위를 지정하여 반복을 수행합니다.
- 위 예시에서는 1부터 10까지의 범위에서 `i` 변수를 반복하며 각 값을 출력합니다.

`for` 문은 기본적으로 1씩 증가하며 반복합니다. 그러나 `step` 키워드를 사용하여 증가 단계를 지정할 수 있습니다. 이를테면, 2씩 증가하고 싶다면 다음과 같이 작성할 수 있습니다:

변수를 이용한 Kotlin `for` 문

```
val len: Int = 10
for(i in 1..len)
    print(i)
```

- Kotlin에서는 변수를 사용하여 반복문을 작성할 수 있습니다.
- 위 예시에서는 `len` 변수에 10을 할당하고, `for` 문에서 1부터 `len`까지의 범위를 반복하며 각 값을 출력합니다.

변수를 사용하여 반복 범위를 동적으로 설정할 수 있어 유연한 반복 작업을 수행할 수 있습니다.

`until`을 이용한 Kotlin `for` 문

```
for(i in 1 until len) //len이 10이므로 1부터 9까지 반복
    print(i)
```

- Kotlin에서 `until` 키워드를 사용하면 마지막 숫자 전까지의 범위를 반복할 수 있습니다.
- 위 예시에서는 `len` 변수가 10이므로 1부터 9까지의 범위에서 `i` 변수를 반복하며 각 값을 출력합니다.

`until`을 사용하면 마지막 숫자를 포함하지 않고 범위를 반복할 수 있으므로 유용합니다.

`step`을 이용한 Kotlin `for` 문

```
for (i: Int in 1..10 step 2) // 1, 3, 5, 7, 9
    print(i)

for (i in 10 downTo 1 step 1) // 10부터 1까지 역순으로 출력
    print(i)
```

- Kotlin에서 `step` 키워드를 사용하면 증가 값을 설정할 수 있습니다.
- 첫 번째 예시에서는 1부터 10까지 2씩 증가하는 범위에서 `i` 변수를 반복하며 홀수를 출력합니다.
- 두 번째 예시에서는 `downTo` 키워드를 사용하여 10부터 1까지 1씩 감소하는 역순 범위에서 `i` 변수를 반복하며 출력합니다.

`step`을 사용하여 원하는 증가 값을 설정하거나 역순으로 반복하는 것이 가능합니다.

`downTo`를 이용한 Kotlin `for` 문

```
for(i in 10 downTo 1) // 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
    print(i)

for(i in 10 downTo 1 step(2)) //10, 8, 6, 4, 2
    print(i)
```

- Kotlin에서 `downTo` 키워드를 사용하면 역순으로 반복할 수 있습니다.
- 첫 번째 예시에서는 10부터 1까지 역순으로 `i` 변수를 반복하며 출력합니다.
- 두 번째 예시에서는 `step` 키워드를 사용하여 2씩 감소하면서 역순으로 반복하는 예시입니다.

`downTo`를 사용하면 역순으로 반복할 수 있으며, `step`을 함께 사용하여 증가폭을 조절할 수도 있습니다.

배열과 리스트를 사용한 for문

배열을 탐색하는 Kotlin `for` 문

```
val arr: Array<Int> = arrayOf(1, 2, 3, 4, 5) //배열 선언

for(i in arr){
    print(i)
}
```

- Kotlin에서 배열을 사용하여 `for` 문을 사용할 때, 배열의 각 요소를 순서대로 탐색하며 작업을 수행할 수 있습니다.
- 위의 예시에서는 정수 배열 `arr`을 선언하고, `for` 문을 사용하여 배열의 각 요소를 출력하는 예시입니다.

배열을 거꾸로 탐색하는 Kotlin `for` 문

```
for(i in arr.reversed()){
    print(i)
}
```

- Kotlin에서 배열을 거꾸로 탐색하려면 `reversed` 함수를 사용합니다.
- 위의 예시에서는 정수 배열 `arr`을 거꾸로 탐색하며 각 요소를 출력하는 예시입니다.

인덱스와 원소 값을 함께 사용하는 Kotlin `for` 문

```
val nameArr: Array<String> = arrayOf("Kim", "Lee", "Park") //배열 선언

    for((index, name) in nameArr.withIndex()){
        println("${index+1}번째 성은 ${name}입니다.")
    }
```

- Kotlin에서는 `withIndex()` 함수를 사용하여 배열의 인덱스와 해당 요소 값을 함께 사용할 수 있습니다.
- 위의 예시에서는 문자열 배열 `nameArr`을 순회하면서 각 성(요소)과 해당 성의 인덱스를 출력하는 예시입니다.

리스트를 사용한 Kotlin `for` 문

```
val list : List<String> = listOf("a", "b", "c") //리스트 선언

for(i in list){
```

```
    print(i)
}
```

- Kotlin에서는 리스트를 사용하여도 배열과 동일하게 `for` 문을 사용할 수 있습니다.
- 위의 예시에서는 문자열 리스트 `list`를 순회하면서 각 요소를 출력하는 예시입니다.

count를 사용한 Kotlin `for` 문

```
for(i in 0 until list.count()){
    print(list[i])
}
```

- Kotlin에서는 리스트의 길이를 `list.size`를 사용하여 `for` 문으로 탐색할 수 있습니다.
- 위의 예시에서는 문자열 리스트 `list`의 길이를 사용하여 각 요소를 출력하는 예시입니다.

while 문

while문

```
var a: Int = 1
while(a <= 10){
    print("${a++} ") //1, 2, 3, 4, 5, 6, 7, 8, 9, 10
}
```

- `while` 문은 조건을 먼저 검사하고 조건이 참일 때 루프를 실행합니다.
- 변수 `a`가 1부터 시작하고, 조건 `a <= 10`이 참일 동안 반복하여 `a`의 값을 출력하고 1씩 증가시킵니다. 결과적으로 1부터 10까지의 숫자가 출력됩니다

do-while문

```
var b: Int = 10
do{
    print("${b--} ")    //10, 9, 8, 7, 6, 5, 4, 3, 2, 1
}while(b > 0)
```

- `do-while` 문은 일단 루프를 한 번 실행한 다음에 조건을 검사합니다. 따라서 조건이 거짓이더라도 최소 한 번은 루프가 실행됩니다.
- 변수 `b`가 10부터 시작하고, 조건 `b > 0`이 참일 동안 반복하여 `b`의 값을 출력하고 1씩 감소시킵니다. 결과적으로 10부터 1까지의 숫자가 출력됩니다.

함수 (Functions)

함수는 코드를 모듈화하고 재사용 가능한 블록 단위로 그룹화하는 데 사용됩니다. Kotlin에서 함수를 선언하려면 `fun` 키워드를 사용하며, 일반적으로 다음과 같은 구조를 가집니다:

```
fun functionName(parameters: ParameterTypes): ReturnType {
    // 함수 본문
    // 실행될 코드
    return result // 함수가 반환하는 값 (optional)
}
```

여기에서 설명을 나눠보겠습니다:

- `fun`: Kotlin에서 함수를 선언하기 위한 키워드입니다.
- `functionName`: 함수의 이름을 지정하는 부분입니다. 함수 이름은 식별자 규칙을 따라야 합니다.
- `parameters`: 함수의 입력 매개변수(파라미터)를 나타내며, 필요한 경우 여러 개의 파라미터를 콤마로 구분하여 선언할 수 있습니다.
- `ParameterTypes`: 각 파라미터의 데이터 타입을 나타냅니다.
- `ReturnType`: 함수가 반환하는 값의 데이터 타입을 나타냅니다. 함수가 값을 반환하지 않는 경우 `Unit` 타입을 사용하거나 반환 타입을 생략할 수 있습니다.
- 함수 본문: 중괄호 `{}` 안에 함수의 동작을 정의하는 부분으로, 실행될 코드 블록을 포함합니다.
- `return result`: 함수가 값을 반환할 때 사용되며, 필요한 경우 값을 반환할 수 있습니다. 반환 값은 함수의 `ReturnType`과 일치해야 합니다.

함수 예시 설명

기본 함수

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

- 함수의 이름은 max이다. a, b를 입력하면 누가 더 큰지 비교하여 더 큰 것을 리턴 하는 함수이다.

기본 함수 호출

```
    var a = 1

    var b = 2

    val max = max(a,b)
```

- 함수를 호출하려면 함수의 이름 뒤에 `호출 연산자()`를 사용한다.

함수 선언 단순화

return 단순화

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

`max()` 함수는 두 개의 정수 `a`와 `b`를 입력으로 받아 더 큰 값을 반환합니다. 하지만 이 함수의 본문은 단순한 if-else 표현식을 사용하고 있어 불필요한 로컬 변수 선언이 있습니다.

`{}`를 `=`로 변환

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

위의 코드는 `max()` 함수를 더 간결하게 표현하는 방법을 보여줍니다. 중괄호(`{}`) 블록 대신 할당 연산자(`=`)를 사용하여 함수의 본문을 표현하고, 이 본문에서 `if-else` 표현식을 사용하여 두 수 중 더 큰 값을 반환합니다.

반환 타입 생략

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

`=` 을 쓰게 되면 반환 타입도 생략 가능합니다. Kotlin은 표현식의 결과를 통해 반환 타입을 추론할 수 있습니다.

중괄호 생략

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

중괄호 `{}` 또한 다 생략 가능하며, 위의 코드처럼 한 줄로 표현할 수 있습니다. 이때 중괄호가 생략되면 반환 타입이 unit(자바로 치면 void)가 아니면 명시적으로 작성해야 합니다.

이렇게 Kotlin에서는 함수를 더 간결하게 선언하고 사용할 수 있습니다.

Default Parameter

디폴트 파라미터(Default Parameter)는 Kotlin에서 함수를 선언할 때 매개변수에 기본값을 지정하는 방법을 제공합니다. 이를 통해 함수를 호출할 때 일부 매개변수를 생략하고, 기본값이 설정된 매개변수만 사용할 수 있습니다.

예시를 통해 Default Parameter를 설명한 내용을 수정하여 보겠습니다:

```
//파라미터에 값을 지정
fun repeat(string: String, num: Int = 3, useNewLine: Boolean = true) {
    for (i in 1..num) {
        if (useNewLine) {
            println(string)
        } else {
            print(string)
        }
    }
}
```

위의 코드에서 `repeat` 함수는 세 개의 매개변수를 가지고 있습니다. 그 중 `num`과 `useNewLine` 매개변수에는 기본값이 지정되어 있습니다. 이렇게 매개변수에 기본값을 지정하면 함수를 호출할 때 해당 매개변수를 생략할 수 있습니다.

예를 들어, 아래와 같이 함수를 호출할 수 있습니다:

```
repeat("Hello") // num은 기본값 3, useNewLine은 기본값 true를 사용
```

기본값이 지정된 매개변수에 대해서는 호출 시 값을 제공하지 않으면 기본값이 사용되며, 필요한 경우에만 값을 지정하여 함수를 호출할 수 있습니다.

Argument Parameter

Kotlin에서는 함수 호출 시 매개변수의 이름을 지정하여 Argument Parameter를 지정할 수 있습니다. 이를 통해 기본값을 변경하고자 하는 매개변수만 선택적으로 설정할 수 있습니다.

예를 들어, 아래와 같이 `repeat` 함수를 호출하여 `num`은 그대로 3을 사용하고, `useNewLine`은 `false`로 변경하고 싶다면 다음과 같이 Argument Parameter를 사용할 수 있습니다:

```
repeat("hello", useNewLine =  false)
```

이와 같이 Argument Parameter를 사용하면 함수 호출 시 명확하게 어떤 매개변수에 값을 할당하는지 지정할 수 있어 코드의 가독성을 높일 수 있습니다.

단, Kotlin에서 Java 함수를 사용할 때는 이러한 기능을 사용할 수 없으며, Java 함수의 매개변수 순서대로 값을 전달해야 합니다.

가변 인자가 있는 함수

가변 인자 함수는 Kotlin에서 간단하게 선언할 수 있으며, 호출 시에도 간편하게 사용할 수 있습니다.

Kotlin에서 가변 인자 함수를 선언할 때에는 매개변수 앞에 `vararg` 키워드를 사용하여 표시합니다. 이를 통해 함수 내에서 가변 개수의 인자를 배열로 다룰 수 있게 됩니다.

예를 들어, 아래와 같이 `printAll` 함수를 선언할 수 있습니다:

KotiIn에서 가변 인자 함수 선언할 때

```
val array = arrayOf("A", "B", "C")
printAll(*array)

fun printAll(vararg strings: String) {
    for (str in strings) {
        println(str)
    }
}


//vararg 가 가변 인자임을 표시해주는 키워드이다.
```

그리고 가변 인자 함수를 호출할 때는 배열을 직접 넘기는 대신 스프레드 연산자(`*`)를 사용하여 간단하게 값을 전달할 수 있습니다.

Kotlin에서의 클래스

Kotlin에서 클래스를 선언하는 방법과 자바에서의 클래스 선언 방법을 비교하였습니다. Kotlin은 간단한 문법을 제공하여 클래스를 선언하고 필드, 생성자, getter, setter 등을 손쉽게 정의할 수 있습니다.

아래는 Java와 Kotlin에서의 클래스 선언 방법을 간결하게 정리한 내용입니다:

```


package com.lannstark.lec09;

public class JavaPerson {


private final String name;

private int age;
```

```java
public JavaPerson(String name, int age) {
    this.name = name;
    this.age = age;
}

public String getName() {
    return name;
}

public int getAge() {
    return age;
}

public void setAge(int age) {
    this.age = age;
}




}
```

자바에서는 이렇게 클래스를 작성하고 해당 필드에 맞는 getter와 setter를 만들어주고 생성자를 만들어주었다.

```kotlin
// 1단계
// 생성자는 파라미터 처럼 오른쪽 옆에 만든다.
class Person constructor(name:String,age:Int) {

    val name:String = name
    var age: Int = age
}

//2단계
//이미 생성자에서 타입을 알려주기 때문에 필드쪽에서 타입 생략 가능하다
class Person constructor(name:String,age:Int) {

    val name= name
    var age = age

}

//3단계
//constructor도 생략 가능하다
class Person(name:String,age:Int) {
```

```
val name= name
var age = age

}

//4단계
//필드에 있는 것을 생성자쪽으로 옮겨서 생성자와 필드 한번에 선언이 가능하다.
class Person constructor(val name:String, var age:Int) {

}
```

Kotlin은 간결한 문법을 통해 필드, 생성자, getter, setter 등을 자동으로 생성해주기 때문에 코드를 더 간편하게 작성할 수 있습니다. 필요한 경우 직접 getter와 setter를 커스터마이징할 수도 있습니다.

Kotlin에서의 객체 인스턴스 선언 및 프로퍼티 접근

```
fun main() {

    val person =Person("재롱이", 10)

    println(person.name)  //getter
    println(person.age)

    person.age= 15  //setter
    println(person.age)
}


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

<코틀린에서 객체 인스턴스 선언 후 사용>

코틀린에서는 객체 인스턴스를 선언하고 해당 객체의 프로퍼티에 접근하는 방식이 자바와 다릅니다.

- **프로퍼티 접근 (getter):** 코틀린에서는 `. 프로퍼티명` 형식으로 직접 프로퍼티에 접근할 수 있습니다. 별도의 `get` 메서드를 호출할 필요가 없습니다.
- **프로퍼티 값 변경 (setter):** 코틀린에서는 setter 메서드가 명시적으로 존재하지 않더라도 `.프로퍼티명 = 값` 형식으로 프로퍼티의 값을 변경할 수 있습니다.

Kotlin 생성자 검증

```
public class JavaPerson {

private final String name;

private int age;

public JavaPerson(String name, int age) {

if (age <= 0) {
    throw new IllegalArgumentException(String.format("나이는 %s일 수 없습니다", age
}
    this.name = name;
    this.age = age;
    }
}


//getter, setter 생략
```

<자바에서 생성자 검증 및 초기화>

```
class Person (val name:String,  var age:Int){
//클래스가 초기화 되는 시점에 한번 호출 됨
    init {
        if(age<=0){
            throw IllegalArgumentException("나이는 ${age}일 수 없습니다.")
        }
    }
}
```

<코틀린 `init` 블록 사용>

코틀린에서는 `init` 블록을 활용하여 생성자의 검증 및 초기화를 수행할 수 있습니다. `init` 블록은 클래스가 초기화될 때 생성자 호출 직후에 실행되는 블록으로, 주로 생성자 파라미터의 유효성을 검사하거나 초기화 로직을 수행하기 위해 사용됩니다.

Kotlin 추가 생성자

```
fun main() {
    val person =Person("재롱이")
```

```
}

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

//추가 생성자
constructor(name: String): this(name,1)
}
```

<클래스 내에서 `constructor` 사용하여 추가 생성자 생성>

추가 생성자는 클래스 내부에서 `constructor` 을 통해서 작성합니다.

생성자 요약 정리

- 클래스는 주 생성자를 반드시 가지며, 단, 주 생성자에 파라미터가 하나도 없다면 생략 가능합니다.
- 부 생성자는 클래스에 있을 수도 있고 없을 수도 있습니다.
  - 부 생성자는 주 생성자를 `this`로 호출해야 합니다.
  - 부 생성자는 본문(body)을 가질 수 있습니다.

```
class Person (
    val name: String = "김종훈",
    var age: Int = 1
) {
    init {
        if (age <= 0) {
            throw IllegalArgumentException("나이는 ${age}일 수 없습니다.")
        }
    }
}```

하지만 보통은 위 와 같이 `default parameter`를 활용하는 것이 권장되며, `constructor`를


### **커스텀 getter , setter**

사용자가 직접 커스텀 해서 메소드를 정의 할 수도 있다.

밑에 예제는 함수처럼 정의 하는 방식과 프로퍼티처럼 정의 하는 방식은 소개한다.

```kotlin

fun main() {

}

class Person (
```
```

```
val name:String = "김종훈",
var age:Int = 1){

init {
if(age<=0){
    throw IllegalArgumentException("나이는 ${age}일 수 없습니다.")
        }
    }


    fun isAdult(): Boolean{
    return this.age >=20
    }
}
```

커스텀으로 getter ,setter를 작성하려면 해당 클래스내에서 fun을 선언하고 작성해주면 된다.

```
    val isAdult: Boolean
    get() = this.age >= 20;

}
```

이건 프로퍼티처럼 작성하는 방법이다. {} ,return 생략이 된다.