학습목표

- 기본적인 Kotlin 문법을 익힌다.

차례

01 Kotlin의 특징

① Java와 100% 상호 호환되므로 Java 코드를 완전히 대체 가능

② Java보다 문법이 간결함

③ 프로그램의 안정성을 높여줌

④ var 또는 val 예약어를 통해 데이터 형식을 선언하지 않고 변수를 선언할수 있음

02 Kotlin 프로그램 작성법

① IntelliJ IDEA 환경에서 개발

② Kotlin 사이트(https://play.kotlinlang.org/)에 접속하여 별도의 설치 없이 개발

02 Kotlin 프로그램 작성법

- **<실습 3-1> IntelliJ IDEA 환경에서 Kotlin 개발하기**
  - (1) https://www.jetbrains.com/idea/download/에 접속하여 'Community'를 다운로드(무료로 다운 가능).

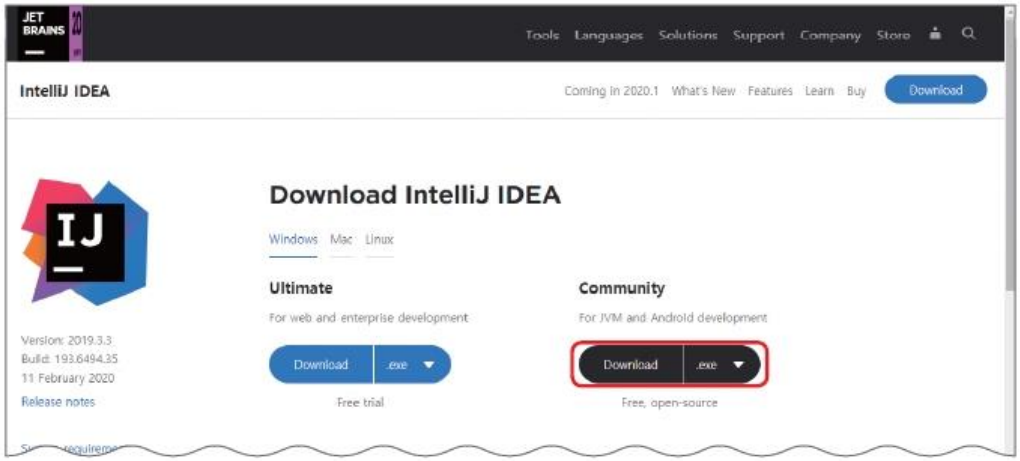


그림 3-1 IntelliJ IDEA 다운로드

02 Kotlin 프로그램 작성법

- (2) 다운로드한 파일을 실행하여 설치.
  - 모두 기본값으로 두고 <Next>를 클릭하여 설치를 진행

- (3) [Installation Options] 창에서는 아무것도 선택하지 않고 <Next>를 클릭

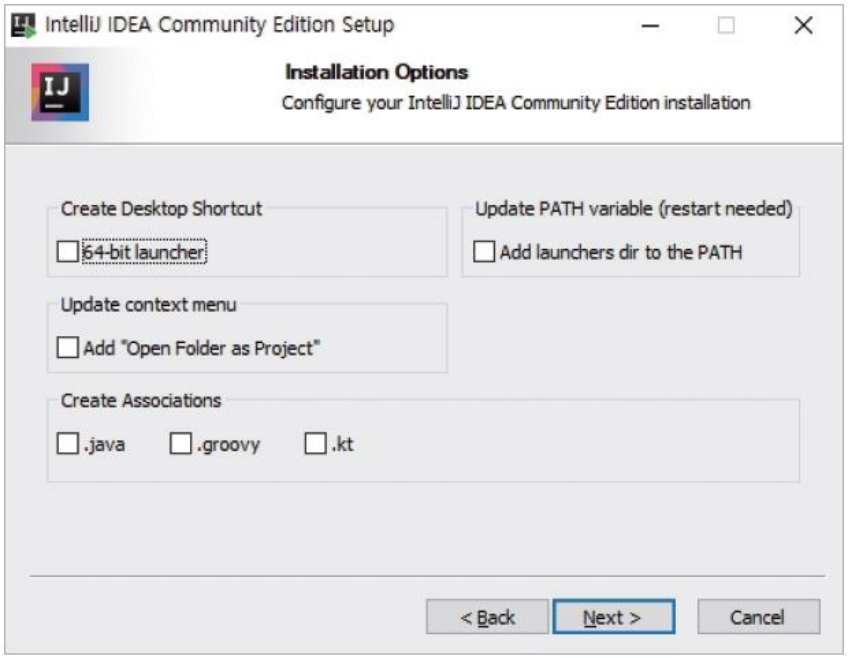


그림 3-3 IntelliJ 설치 옵션 선택 창

02 Kotlin 프로그램 작성법

- (4) 설치가 완료되면 [Windows 시작]-[JetBrains]-[IntelliJ IDEA Community Edition]을 선택하여 실행
  - [Import IntelliJ IDEA Settings From...] 창이 나오면 'Do not import settings'를 선택하고 <OK>를 클릭.

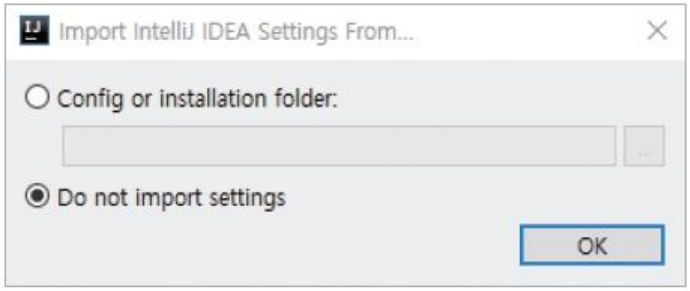


그림 3-4 IntelliJ 이전 환경 가져오기 설정

- (5) [JetBrains Privacy Policy] 창이 나오면 'I confirm that I have read and ...'에 체크하고 <Continue>를 클릭

그림 3-5 IntelliJ IDEA 로고

02 Kotlin 프로그램 작성법

- (6) [Set UI theme] 창이 나오면 'Darcula' 또는 'Light' 모드를 선택하고 왼쪽 아래의 <Skip Remaining and Set Defaults>를 클릭
- (7) 초기화면에서 [Create New Project]를 클릭

그림 3-6 새 프로젝트 생성 1

02 Kotlin 프로그램 작성법

- (8) [New Project] 창에서 왼쪽의 [Kotlin]에 이어 오른쪽의 [JVM | IDEA]를 선택한 후 <Next>를 클릭

그림 3-7 새 프로젝트 생성 2

- (9) [Project name]에 'HelloWorld'를 입력하고 <Finish>를 클릭

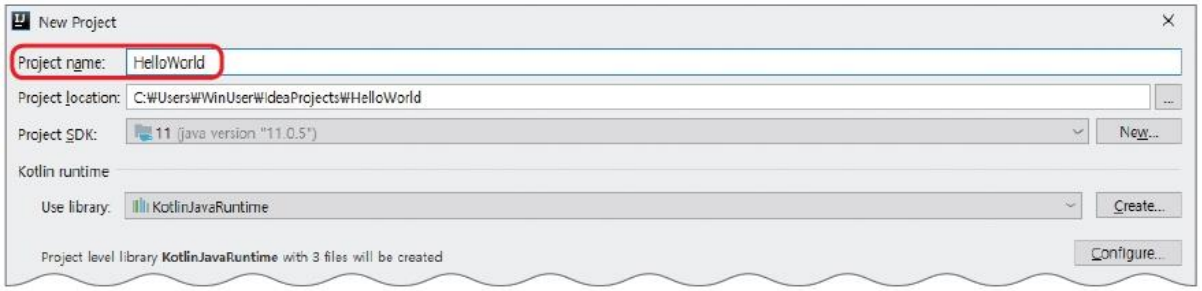


그림 3-8 새 프로젝트 생성 3

02 Kotlin 프로그램 작성법

- (10) 왼쪽의 [HelloWorld]-[src]에서 마우스 오른쪽 버튼을 클릭하고 [New]-[Kotlin File/Class]를 선택
  - 이름에 'HelloWorld'를 입력하고 [File]을 더블클릭

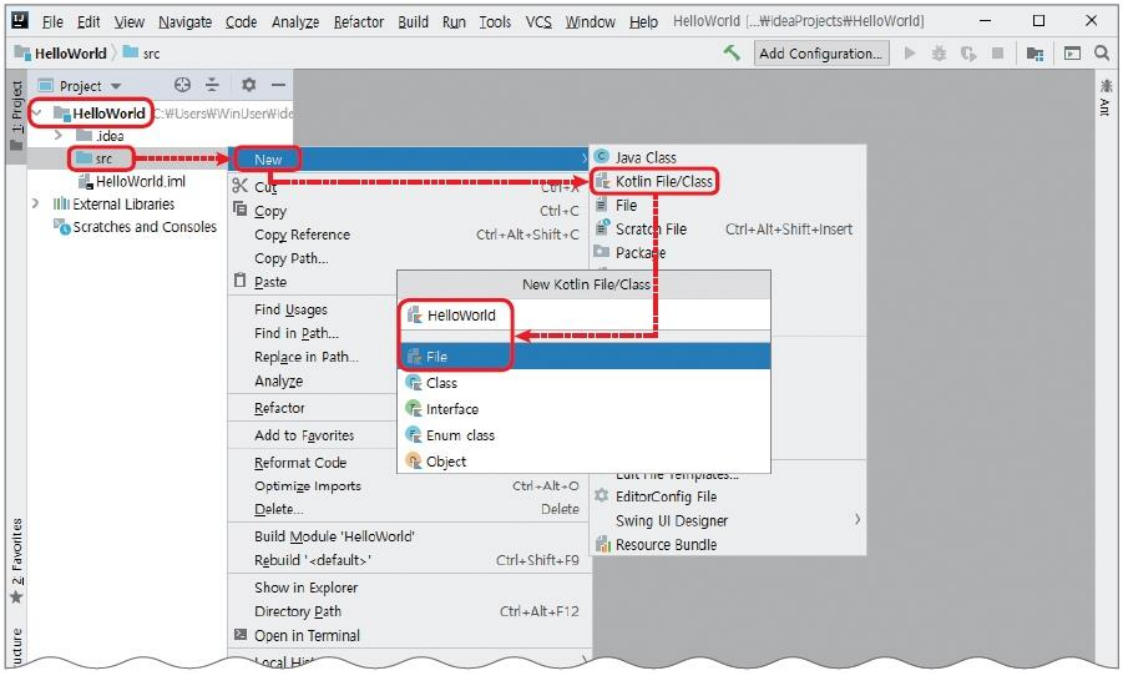


그림 3-9 소스코드 추가

02 Kotlin 프로그램 작성법

- (11) 빈 화면에 다음과 같이 간단한 예제 입력

예제 3-1 HelloWorld.kt

```
1  fun main() {
2      println("안드로이드를 위한 Kotlin 연습")
3  }
```

02 Kotlin 프로그램 작성법

- (12) 빈 화면에서 마우스 오른쪽 버튼을 클릭하고 [Run 'HelloWorldKt']를 선택하면 아래 쪽 콘솔에 실행 결과가 나타남

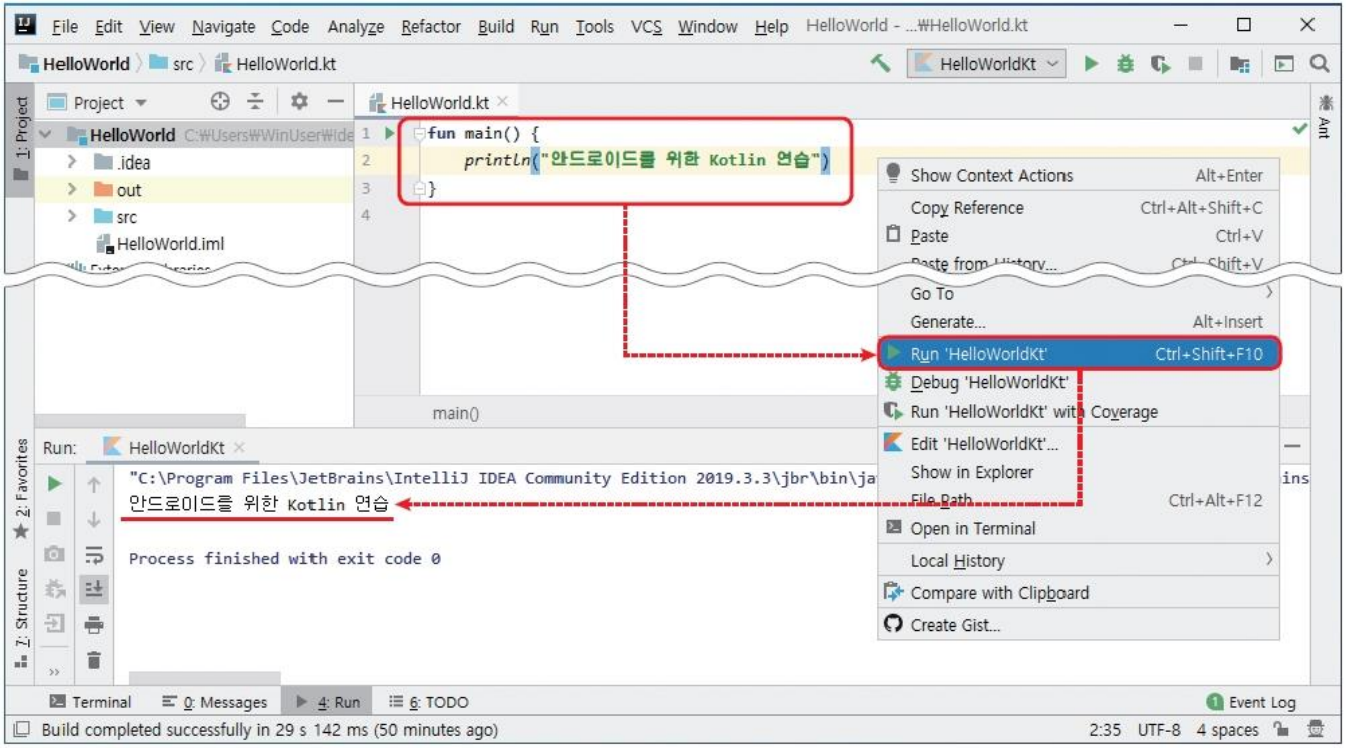


그림 3-10 코드의 컴파일과 실행

01 변수와 데이터 형식

- 정수형 변수, 실수형 변수, 문자형 변수, 문자열 변수를 선언하고 값을 대입한 후 출력하는 예제

예제 3-2 exam01.kt

```
1  fun main() {
2      var var1 : Int = 10
3      var var2 : Float = 10.1f
4      var var3 : Double = 10.2
5      var var4 : Char = '안'
6      var var5 : String = "안드로이드"
7      println(var1)
8      println(var2)
9      println(var3)
10     println(var4)
11     println(var5)
12 }
```

```
10
10.1
10.2
안
안드로이드
```

01 변수와 데이터 형식

- **Kotlin에서 많이 사용되는 기본적인 데이터 형식**

분류	데이터 형식	설명
문자형	Char	2byte를 사용하며 한글 또는 영문 1개만 입력
	String	여러 글자의 문자열을 입력
정수형	Byte	1byte를 사용하며 –128~+127까지 입력
	Short	2byte를 사용하며 –32768~+32767까지 입력
	Int	4byte를 사용하며 약 –21억~+21억까지 입력
	Long	8byte를 사용하며 상당히 큰 정수까지 입력 가능
실수형	Float	4byte를 사용하며 실수를 입력
	Double	8byte를 사용하며 실수를 입력. Float보다 정밀도가 높음
불리언형	Boolean	true 또는 false를 입력

01 변수와 데이터 형식

Kotlin의 변수 선언 방식

1) 암시적 선언

- 변수의 데이터 형식을 지정하지 않고, 대입되는 값에 따라 자동으로 변수의 데이터 형식이 지정
  - 단, 초기화하지 않는 경우에는 데이터 형식을 반드시 명시해야 함.

- 예제 3-2의 2~6행을 아래와 같이 수정 가능

```
var var1 = 10
var var2 = 10.1f
var var3 = 10.2
var var4 = '안'
var var5 = "안드로이드"
```

01 변수와 데이터 형식

- **Kotlin의 변수 선언 방식**
  - 2) var(variable)
    - 일반 변수를 선언할 때 사용
    - 필요할 때마다 계속 다른 값을 대입 가능
  - 3) val(value)
    - 변수 선언과 동시에 값을 대입하거나, 초기화 없이 선언한 후에 한 번만 값을 대입 가능
    - 한 번 값을 대입하고 나면 값을 변경할 수 없음

```
var myVar : Int = 100
myVar = 200 // 정상

val myVal : Int = 100
myVal = 200 // 오류
```

01 변수와 데이터 형식

- **데이터 형식 변환**
  - 캐스팅 연산자 사용
  - Kotlin에서 제공하는 toInt( )나 toDouble( ) 등의 정적 메소드 사용

```
var a : Int = "100".toInt()
var b : Double = "100.123".toDouble()
```

01 변수와 데이터 형식

- **null 사용**
  - Kotlin은 기본적으로 변수에 null 값을 넣지 못함
    - 변수를 선언할 때 데이터 형식 뒤에 **?**를 붙여야 null 대입 가능

```
var notNull : Int = null  // 오류
var okNull : Int? = null  // 정상
```

- **변수가 null 값이 아니라고 표시해야 하는 경우**
  - **!!**로 나타냄
    - 이 경우 null 값이 들어가면 오류 발생

```
var ary = ArrayList<Int>(1)  // 1개짜리 배열 리스트
ary!!.add(100) // 값 100을 추가
```

02 조건문: if, when

- **if 문**
  - 조건이 true인지 false인지에 따라서 어떤 작업을 할 것인지를 결정
  - 이중 분기라고도 부름
- **if 문의 형식**

```
if(조건식) {
    // 조건식이 true일 때 이 부분 실행
}
```

- **if-else 문의 형식**

```
if(조건식) {
    // 조건식이 true일 때 이 부분 실행
} else {
    // 조건식이 false일 때 이 부분 실행
}
```

02 조건문: if, when

- **when 문**
  - 여러 가지 경우에 따라서 어떤 작업을 할 것인지를 결정
  - 다중 분기라고도 부름
- **when 문의 형식**

```
when (식) {
    값1 ->  // 값1이면 이 부분 실행
    값2 ->  // 값2이면 이 부분 실행
    ⋮
    else ->  // 어디에도 해당하지 않으면 이 부분 실행
}
```

02 조건문: if, when

예제 3-3 exam02.kt

```
1  fun main() {
2      var count : Int = 85
3      if (count >= 90) {
4          println("if문: 합격 (장학생)")
5      } else if (count >= 60) {
6          println("if문: 합격")
7      } else {
8          println("if문: 불합격")
9      }
10
11     var jumsu : Int = (count / 10) * 10
12     when (jumsu) {
13         100 -> println("when문: 합격(장학생)")
14         90 -> println("when문: 합격(장학생)")
15         80, 70, 60 -> println("when문: 합격")
16         else -> println("when문: 불합격")
17     }
18 }
```

```
if문: 합격
when문: 합격
```

02 조건문: if, when

- **when 문의 처리 방법**
  - 각각의 값에 따라 처리 : [예제 3-3] 방식
  - 범위로 처리 : in 키워드 사용
    - [예제 3-3]의 11~17행을 다음과 같이 수정해도 동일한 결과를 얻을 수 있음

```
var jumsu : Int = count
when (jumsu) {
    in 90 .. 100 -> println("when문: 합격(장학생)")
    in 60 .. 89 -> println("when문: 합격")
    else -> println("when문: 불합격")
}
```

03 배열

▪ 배열

- 여러 개의 데이터를 하나의 변수에 저장하기 위해 사용

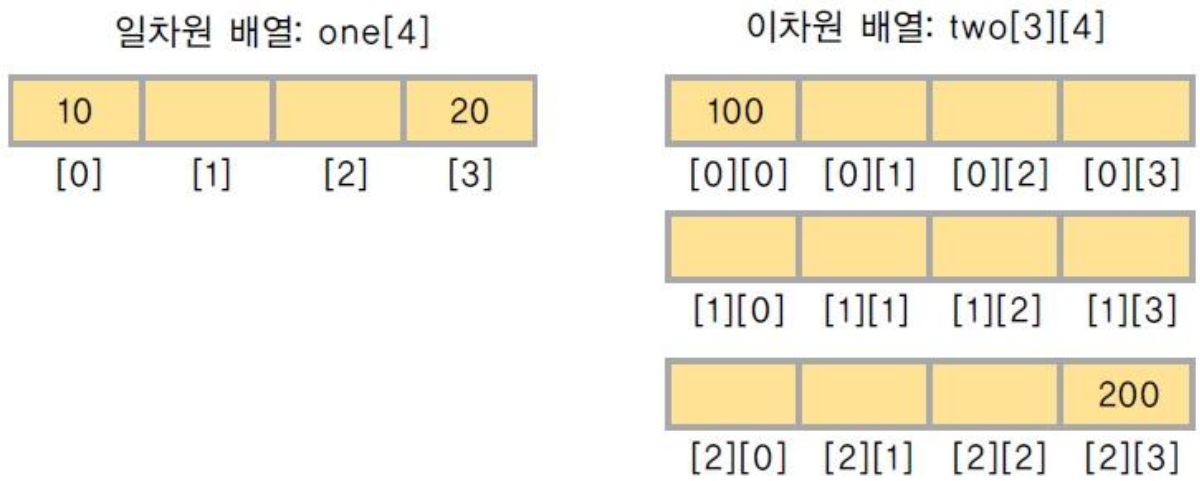


그림 3-11 배열의 개념

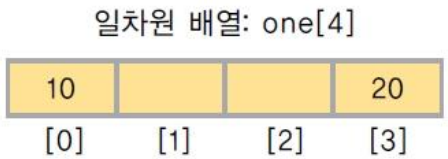
03 배열

▪ 일차원 배열 선언 형식

```
var 배열명 = Array<데이터 형식>(개수, {초깃값})
```

```
var 배열명 = Array<데이터 형식>(개수) {초깃값}
```

– 일차원 배열(one[4])을 선언하고 값을 대입하는 방법




```
var one = Array<Int>(4, {0})
one[0] = 10
one[3] = 20
```

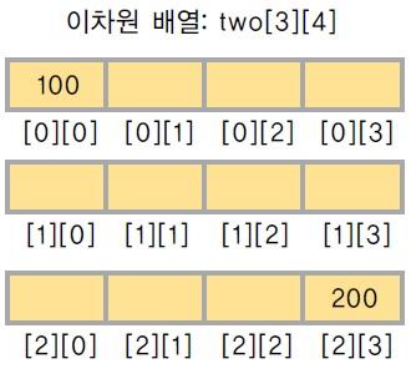
03 배열

▪ 이차원 배열 선언 형식

```
var 배열명 = Array<배열 데이터 형식>(행 개수, {배열 데이터 형식(열 개수)})
```

```
var 배열명 = Array<데이터 형식>(개수) {초깃값}
```

– 3×4 이차원 배열(two[3][4])을 선언하고 값을 대입하는 방법




```
var two = Array<IntArray>(3, {IntArray(4)})
two[0][0] = 100
two[2][3] = 200
```

03 배열

- 배열을 선언하면서 값을 바로 대입하는 것도 가능함

```
var three : IntArray = intArrayOf(1,2,3)
```

- ArrayList

```
var one = ArrayList<Int>(4)
one.add(10)
one.add(20)
var hap = one.get(0) + one.get(1)
// 첫 번째 값 + 두 번째 값
```

04 반복문: for, while

- **for 문**

```
for (변수 in 시작..끝 step 증가량) {
    // 이 부분을 반복 실행
}
```

  - 배열의 개수만큼 변수에 대입하여 반복하는 방법

```
for (변수 in 배열명.indices) {
    // 이 부분을 반복 실행
}
```

04 반복문: for, while

- 배열의 모든 값을 출력하는 방법

```
var one : IntArray = intArrayOf(10,20,30,40)
for (i in one.indices) {
    println(one[i])
}
```

- 첨자(i) 없이 바로 배열의 값을 하나씩 처리하는 방법

```
for (변수 in 배열명) {
    // 이 부분에서 변수 사용
}
```

  - 배열의 내용이 하나씩 변수에 대입된 후 for 문 내부 실행
    - » **결국 배열의 개수만큼 for 문이 반복됨**

04 반복문: for, while

- **while 문**

```
while (조건식) {
    // 조건식이 true인 동안 이 부분을 실행
}
```

- **break 문**
  - 반복문을 빠져나올 때 사용

- **continue 문**
  - 반복문의 조건식으로 건너뛸 때 사용

04 반복문: for, while

예제 3-4 exam03.kt

```
1  fun main() {
2      var one : IntArray = intArrayOf(10,20,30)
3      for (i  in  one.indices) {
4          println(one[i])
5      }
6      for (value  in one) {
7          println(value)
8      }
9
10     var two : Array<String> = arrayOf("하나", "둘", "셋")
11     for (i in 0..2 step 1) {
12         println(two[i])
13     }
14     var k : Int = 0
15     while (k < two.size) {
16         println(two[k])
17         k++
18     }
19 }
```

```
10
20
30
10
20
30
하나
둘
셋
하나
둘
셋
```

05 메소드와 전역변수, 지역변수

- **메소드**
  - 기본 메소드인 main( ) 함수 외에 사용자가 메소드를 추가로 생성할 수 있음
  - 메소드를 호출할 때 파라미터를 넘길 수 있음
  - 메소드에서 사용된 결과를 return 문으로 돌려줄 수도 있음.

- **변수**
  - 전역변수(global variable)
    - 전역변수는 모든 메소드에서 사용됨
  - 지역변수(local variable)
    - 메소드 내부에서만 사용됨

05 메소드와 전역변수, 지역변수

예제 3-5 exam04.kt

```
1  var myVar : Int = 100
2  fun main() {
3      var myVar : Int = 0
4      println (myVar)
5
6      var sum : Int = addFunction(10,20)
7      println(sum)
8  }
9
10 fun addFunction(num1: Int, num2: Int) : Int {
11     var hap : Int
12     hap = num1 + num2 + myVar
13     return hap
14 }
```

```
0
130
```

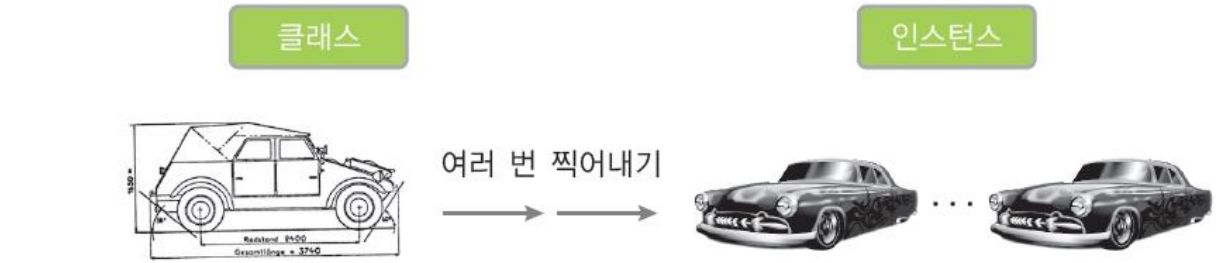
07 연산자

표 3-2 주로 사용되는 Kotlin 연산자

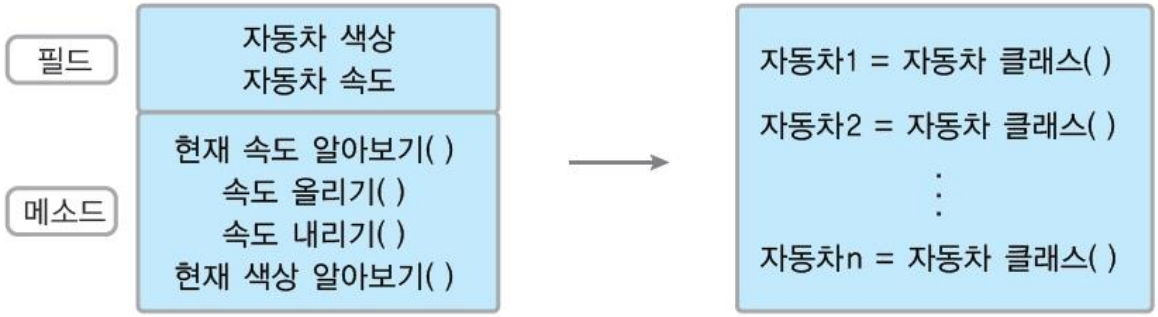
연산자	설명
+, −, *, /, %	사칙 연산자로 %는 나머지값을 계산한다.
+, −	부호 연산자로 변수, 수, 식 앞에 붙일 수 있다.
=	대입 연산자로 오른쪽을 왼쪽에 대입한다.
++, −−	1씩 증가 또는 감소시킨다.
==, ===, !=, !==, <, >, <=, >=	비교 연산자로 결과는 true 또는 false이며, if 문이나 반복문의 조건식에 주로 사용된다.
&&, ||, !	논리 연산자로 and, or, not을 의미한다.
and, or, xor, inv()	비트 연산자로, 비트 단위로 and, or, exclusive or, not 연산을 한다.
shr, shl	시프트 연산자로, 비트 단위로 왼쪽 또는 오른쪽으로 이동한다.
+=, −=, *=, /=	복합 대입 연산자로, 'a+=b'는 'a=a+b'와 동일하다.
toByte(), toShort(), toInt(), toLong(), toFloat(), toDouble(), toChar()	데이터 형식을 강제로 변환하는 함수이다. 예를 들어 'var a : Int = (3.14).toInt()'는 Double 형인 3.14 값을 Int 형으로 강제로 변환하여 a에 대입한다. 즉 a에 3이 대입된다.

01 클래스 정의와 인스턴스 생성

- **클래스(class)**
  - 변수(필드)와 메소드로 구성
  - 객체지향 관점에서의 클래스 :
    - 실세계의 객체들이 가질 수 있는 상태와 행동



(a) 클래스와 인스턴스의 개념



(b) 자동차 객체의 클래스와 인스턴스

그림 3-12 클래스의 형식

01 클래스 정의와 인스턴스 생성

- **[그림 3-12]의 자동차 클래스를 구현한 예제**
  - 클래스 이름은 Car로 지정

예제 3-7 Car.kt–기본 구조

```
1  class Car {
2      var color : String = ""
3      var speed : Int = 0
4
5      fun upSpeed(value: Int) {
6          if (speed+value >= 200)
7              speed = 200
8          else
9              speed = speed + value
10     }
11     fun downSpeed(value: Int) {
12         if (speed-value <= 0)
13             speed = 0
14         else
15             speed = speed - value
16     }
17 }
```

01 클래스 정의와 인스턴스 생성

- [예제 3-7]에서 정의한 Car 클래스를 인스턴스로 생성

예제 3-8 exam06.kt

```
1  class Car {
2      ~~~ 생략([예제 3-7]과 동일) ~~~
3  }
4
5  fun main() {
6      var myCar1 : Car = Car()
7      myCar1.color = "빨강"
8      myCar1.speed = 0
9
10     var myCar2 : Car = Car()
11     myCar2.color = "파랑"
12     myCar2.speed = 0
13
14     var myCar3 : Car = Car()
15     myCar3.color = "초록"
16     myCar3.speed = 0
17
18     myCar1.upSpeed(50)
19     println("자동차1의 색상은 " + myCar1.color
20         + "이며, 속도는 " + myCar1.speed + "km입니다.");
21
22     myCar2.upSpeed(20)
23     ~~~ 생략(myCar2 내용 출력) ~~~
24
25     myCar3.upSpeed(250)
26     ~~~ 생략(myCar3 내용 출력) ~~~
27 }
```

```
자동차1의 색상은 빨강이며, 속도는 50km입니다.
자동차2의 색상은 파랑이며, 속도는 20km입니다.
자동차3의 색상은 초록이며, 속도는 200km입니다.
```

02 생성자

- [예제 3-7]의 Car.kt에 생성자 코드 추가

예제 3-9 Car.kt–생성자 추가

```
1  class Car {
2      var color : String = ""
3      var speed : Int = 0
4
5      constructor(color: String, speed: Int) {
6          this.color = color
7          this.speed = speed
8      }
9      ~~~ 생략([예제 3-7]의 5행 이하와 동일) ~~~
```

02 생성자

- [예제 3-8]의 Car 클래스를 사용한 myCar1, myCar2, myCar3의 내용을 변경해야 함.
  - [예제 3-8]의 6~16행을 아래와 같이 수정.
  - 실행 결과는 [예제 3-8]과 동일.

예제 3-10 exam06.kt–수정

```
1  fun main() {
2      var myCar1 : Car  = Car("빨강", 0)
3      var myCar2 : Car  = Car("파랑", 0)
4      var myCar3 : Car  = Car("초록", 0)
5
6      ~~~ 생략([예제 3-8]의 18행 이하와 동일) ~~~
```

03 메소드 오버로딩

- **메소드 오버로딩(overloading)**
  - 한 클래스 내에서 메소드의 이름이 같아도 파라미터의 개수나 데이터 형식만 다르면 여러 개를 선언할 수 있음

  - [예제 3-9]의 9행에 다음 예제의 10~15행을 추가

예제 3-11 Car.kt–메소드 오버로딩 추가

```
1  class Car {
2      var color : String = ""
3      var speed : Int = 0
4
5      constructor(color: String, speed: Int) {
6          this.color = color
7          this.speed = speed
8      }
9
10     constructor(speed: Int) {
11         this.speed = speed
12     }
13
14     constructor() {
15     }
16
17     ~~~ 생략([예제 3-7]의 5행 이하와 동일) ~~~
```

04 정적 필드, 정적 메소드, 상수 필드

- **정적 필드(static field)**
  - 인스턴스를 생성하지 않고 클래스 자체에서 사용되는 변수
  - companion object { } 안에 작성하여 정적 필드를 만듦

- **정적 메소드(static method)**
  - 메소드 또한 companion object { } 안에 작성하면 됨
  - 인스턴스를 생성하지 않고도 '클래스명.메소드명( )'으로 호출하여 사용 가능

- **상수 필드**
  - 정적 필드에 초깃값을 입력하고 const val로 선언
  - 선언한 후에는 값을 변경할 수 없음
  - 상수 필드는 대문자로 구성하는 것이 일반적임
  - 클래스 안에 상수를 정의할 때 사용

04 정적 필드, 정적 메소드, 상수 필드

예제 3-12 Car.kt-정적 구성 요소 추가

```
1  class Car {
2      var color : String = ""
3      var speed : Int = 0
4      companion object {
5          var carCount : Int = 0
6          const val MAXSPEED : Int = 200
7          const val MINSPEED : Int = 0
8          fun currentCarCount() : Int {
9              return carCount
10         }
11     }
12
13     constructor(color: String, speed: Int) {
14        this.color = color
15        this.speed = speed
16        carCount ++
17     }
18
19     ~~~ 생략([예제 3-11]의 10행 이하와 동일) ~~~
```

04 정적 필드, 정적 메소드, 상수 필드

예제 3-13 exam07.kt

```
1  class Car {
2      ~~~ 생략([예제 3-12]의 Car 클래스와 동일) ~~~
3  }
4  fun main() {
5      var myCar1 : Car  = Car("빨강", 0)
6      var myCar2 : Car  = Car("파랑", 0)
7      var myCar3 : Car  = Car("초록", 0)
8
9      println("생산된 차의 대수(정적 필드) ==> " + Car.carCount)
10     println("생산된 차의 대수(정적 메소드) ==> " + Car.currentCarCount())
11     println("차의 최고 제한 속도 ==> " + Car.MAXSPEED)
12
13     println("PI의 값 ==> " + Math.PI)
14     println("3의 5제곱 ==> " + Math.pow(3.0, 5.0))
15 }
```

```
생산된 차의 대수(정적 필드) ==> 3
생산된 차의 대수(정적 메소드) ==> 3
차의 최고 제한 속도 ==> 200
PI의 값 ==> 3.141592653589793
3의 5제곱 ==> 243.0
```

01 클래스 상속과 메소드 오버라이딩

- **클래스 상속(inheritance)**
  - 기존의 클래스가 가지고 있는 것을 그대로 물려받으면서 필요한 필드나 메소드를 추가로 정의하는 것을 의미함

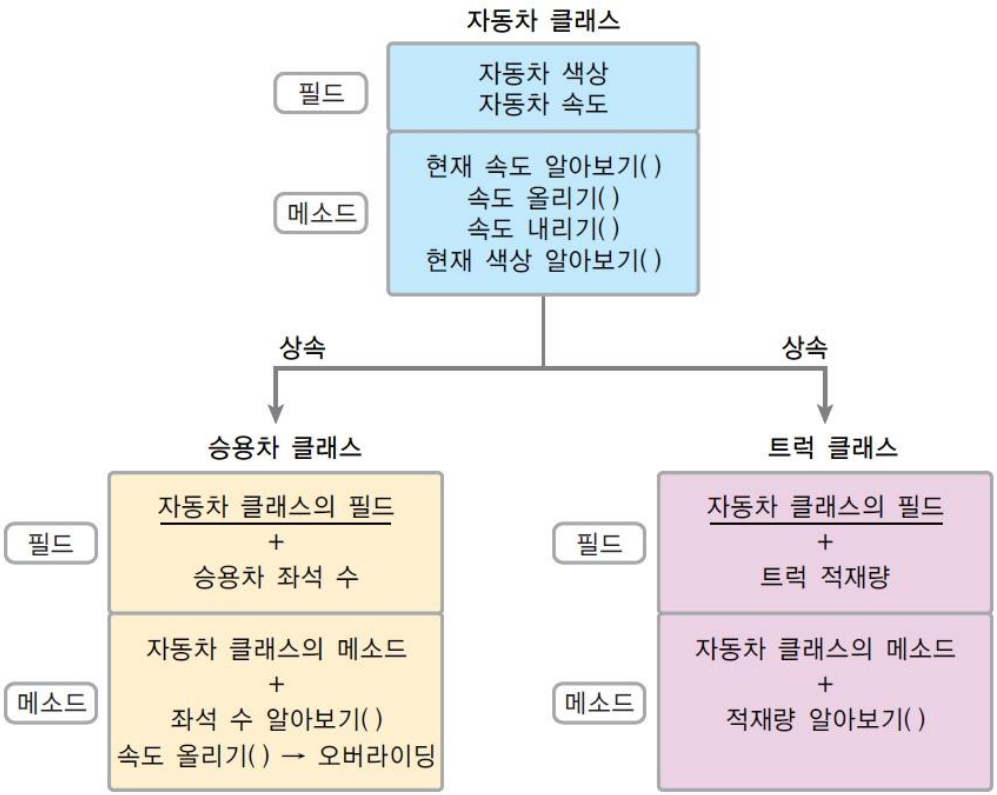


그림 3-13 클래스 상속의 개념

01 클래스 상속과 메소드 오버라이딩

- **슈퍼클래스(superclass, 또는 부모 클래스)**
  - [그림 3-13]에서의 자동차 클래스를 의미함

- **서브클래스(subclass, 또는 자식 클래스)**
  - [그림 3-13]에서의 승용차 클래스나 트럭 클래스를 의미함
  - 서브클래스는 슈퍼클래스의 모든 필드와 메소드를 상속받음
    - → 별도의 선언이나 정의 없이도 슈퍼클래스의 필드와 메소드를 사용할 수 있음

- **메소드 오버라이딩(overriding)**
  - 슈퍼클래스의 메소드를 무시하고 새로 정의하는 것을 의미함
    - 승용차 클래스의 '속도 올리기( )' 메소드는 슈퍼 클래스의 메소드를 재정의함

01 클래스 상속과 메소드 오버라이딩

- [그림 3-13]의 승용차 클래스를 Kotlin 코드로 변경한 예제

예제 3-14 Automobile.kt

```
1  open class Car {
2      ~~~ 생략([예제 3-12]의 Car 클래스와 동일) ~~~
3      open fun upSpeed(value: Int) {
4          if (speed+value >= 200)
5          ~~~ 생략 ~~~
6  }
7
8  class Automobile : Car {
9      var seatNum : Int = 0
10
11     constructor() {
12     }
13     fun countSeatNum() : Int {
14         return seatNum
15     }
16
17     override fun upSpeed(value: Int) {
18         if (speed+value >= 300)
19             speed = 300
20         else
21             speed = speed + value
22     }
23 }
```

01 클래스 상속과 메소드 오버라이딩

- 서브클래스를 사용하는 간단한 예제

예제 3-15 exam08.kt

```
1  ~~~ 생략([예제 3-14]의 Car 및 Automobile 클래스와 동일) ~~~
2
3  fun main() {
4      var auto : Automobile = Automobile()
5      auto.upSpeed(250)
6      println("승용차의 속도는 " + auto.speed + "km입니다.")
7  }
```

```
승용차의 속도는 250km입니다.
```

02 추상 클래스와 추상 메소드

- **추상(abstract) 클래스**
  - 인스턴스화를 금지하는 클래스
  - 추상 클래스는 클래스 앞에 abstract 키워드를 붙여서 지정함
  - 인스턴스화란
    - [예제 3-15]의 4행과 같이 클래스로 인스턴스를 생성하는 것을 의미.

- **추상 메소드**
  - 본체가 없는 메소드
  - 메소드 앞에 abstract 키워드를 붙여서 지정
    - 추상 메소드를 포함하는 클래스는 추상 클래스로 지정해야 함.

02 추상 클래스와 추상 메소드

- 추상 클래스와 추상 메소드를 사용하는 목적
  - 공통적으로 사용되는 기능을 추상 메소드로 선언 해놓고, 추상 클래스를 상속받은 후에 추상 메소드를 오버라이딩해서 사용하기 위함
  - '**구현한다(implement)**'
    - 추상 메소드를 오버라이딩하는 것을 의미함

- 동물 클래스를 추상 클래스로 만들고 추상 메소드인 '이동한다( )'를 포함함

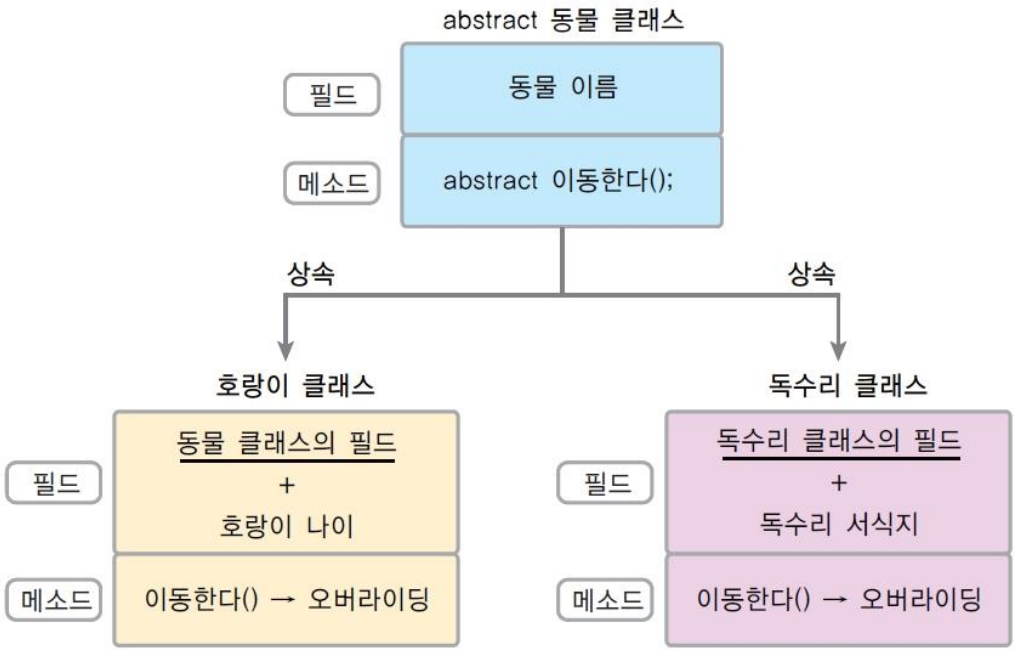


그림 3-14 추상 클래스 상속의 개념

02 추상 클래스와 추상 메소드

- [그림 3-14]를 구현한 예제

예제 3-16 Animal.kt

```
1  abstract class Animal {
2      var name : String = ""
3      abstract fun move()
4  }
5
6  class Tiger : Animal() {
7      var age : Int = 0
8      override fun move() {
9          println("네 발로 이동한다.")
10     }
11 }
12
13 class Eagle : Animal() {
14     var home : String = ""
15     override fun move() {
16         println("날개로 날아간다.")
17     }
18 }
19
20 fun main() {
21     var tiger1 = Tiger()
22     var eagle1 = Eagle()
23
24     tiger1.move()
25     eagle1.move()
26 }
```

```
네 발로 이동한다.
날개로 날아간다.
```

03 클래스 변수의 다형성

- **다형성(polymorphism)**
  - 서브클래스에서 생성한 인스턴스를 자신의 클래스 변수에 대입할 수 있는 것을 의미함
  - 하나의 변수에 여러 종류의 인스턴스를 대입할 수 있음

예제 3-17 exam09.kt

```
1  ~~~ 생략([예제 3-16]의 Animal, Tiger, Eagle 클래스와 동일) ~~~
2  fun main() {
3      var animal : Animal
4
5      animal = Tiger()
6      animal.move()
7
8      animal = Eagle()
9      animal.move()
10 }
```

```
네 발로 이동한다.
날개로 날아간다.
```

04 인터페이스와 다중 상속

- **인터페이스(interface)**
  - 추상 클래스와 성격이 비슷함
  - interface 키워드를 사용하여 정의하고 내부에는 추상 메소드를 선언함
  - 클래스에서 인터페이스를 받아 완성할 때 상속과 마찬가지로 '**: 인터페이스 이름**' 형식을 사용함
  - 인터페이스는 '상속받는다'고 하지 않고 '**구현한다**'고 함
  - Kotlin은 다중 상속을 지원하지 않음
    - 대신 인터페이스를 사용하여 다중 상속과 비슷하게 작성할 수 있음

04 인터페이스와 다중 상속

예제 3-18 exam10.kt

```
1  abstract class Animal {
2      var name : String = ""
3      abstract fun move()
4  }
5
6  interface iAnimal {
7      abstract fun eat()
8  }
9
10 class iCat : iAnimal {
11     override fun eat() {
12         println("생선을 좋아한다.")
13     }
14 }
15
16 class iTiger : Animal(), iAnimal {
17     override fun move() {
18         println("네 발로 이동한다.")
19     }
20     override fun eat() {
21         println("멧돼지를 잡아먹는다.")
22     }
23 }
24
25 class Eagle : Animal() {
26     var home : String = ""
27     override fun move() {
28         println("날개로 날아간다.")
29     }
30 }
31
32 fun main() {
33     var cat = iCat()
34     cat.eat()
35
36     var tiger = iTiger()
37     tiger.move()
38     tiger.eat()
39 }
```

```
생선을 좋아한다.
네 발로 이동한다.
멧돼지를 잡아먹는다.
```

05 람다식

- **람다식(lambda expression)**
  - 함수를 익명 함수(anonymous function) 형태로 간단히 표현 한 것
  - 코드가 간결해져서 가독성이 좋아짐
  - 람다는 { } 안에 매개변수 와 메소드의 모든 내용을 선언

  - 파라미터 2개를 받아서 합계를 출력하는 일반적인 메소드 형식

```
fun addNumber (n1: Int, n2: Int) : Int {
    return n1 + n2
}
```

    - 이를 람다식으로 간단히 표현할 수 있음

```
val addNumber = { n1: Int, n2: Int -> n1 + n2 }
```

05 람다식

▪ 람다식의 특징

① 람다식은 { }로 감싸며 fun 예약어를 사용하지 않음

② { } 안 ->의 왼쪽은 파라미터, 오른쪽은 함수의 내용

③ -> 오른쪽 문장이 여러 개라면 세미콜론(;)으로 구분

④ 내용 중 마지막 문장은 반환값(return)임.

– 이후 장에서는 버튼을 클릭했을 때 실행되는 람다식을 아래 형태로 구현함

```
버튼변수.setOnClickListener {
    // 버튼을 클릭하면 실행될 내용
}
```

06 예외 처리: try~catch

- **예외(exception)**
  - Kotlin 프로그램 실행 중에 발생하는 오류
    - **try~catch** 문을 통해 이 오류를 처리함

예제 3-6 exam05.kt

```
1  fun main() {
2      var num1 : Int = 100
3      var num2 : Int = 0
4      try {
5          println(num1/num2)
6      } catch (e : ArithmeticException)  {
7          println("계산에 문제가 있습니다")
8      }
9  }
```

```
계산에 문제가 있습니다
```

01 패키지

- **패키지**
  - 클래스와 인터페이스가 많아지면 관리하기가 어렵기 때문에 패키지 단위로 묶어서 관리함.
    - 사용자가 생성한 클래스가 포함될 패키지는 *.kt 파일 첫 행에 아래와 같이 지정

```
package 패키지명
```

02 제네릭스

- **제네릭스(Generics)**
  - 데이터 형식의 안전성을 보장하는 데 사용함
    - 제네릭스를 사용하여 strList에 문자열만 들어가도록 설정할 수 있음.
      - 다음 코드의 마지막 행은 컴파일 오류가 발생

```
var strList  =  ArrayList<String>(4)
strList.add("첫 번째")
strList.add("두 번째")
strList.add(3)
```

- 제네릭스는 <String>뿐 아니라 <Int>, <Double> 등을 사용할 수 있음
- 사용자가 정의한 클래스 형에도 사용할 수 있음

03 문자열 비교, 날짜 형식

- **문자열 비교**
  - String 클래스의 equals( ) 메소드 사용

```
var str : String  = "안녕하세요"
if (str.equals("안녕하세요")) {
    // 문자열이 같으면 이곳을 수행
}
```

03 문자열 비교, 날짜 형식

- **날짜 형식**
  - DateFormat 클래스를 상속받은 SimpleDate Format 사용
    - '연월일'이나 '시분초'와 같은 일반적인 표현이 가능함

```
import java.text.DateFormat
import java.text.SimpleDateFormat
import java.util.*

fun main() {
    var now = Date()
    var sFormat : SimpleDateFormat

    sFormat = SimpleDateFormat("yyyyMMdd")
    sFormat = SimpleDateFormat("HH:mm:ss")
    println(sFormat.format(now)) // 23:15:21 형식으로 출력
}
```