

안드로이드 프로그래밍

2025년 2학기

함수

함수 선언 (fun)

- 함수는 fun 키워드를 사용하여 선언
- 함수의 기본 구조

```
fun 함수이름(인자이름: 인자타입): 반환타입 {  
    // 함수 본문  
    return 반환값  
}
```

- 함수이름: 함수를 호출할 때 사용하는 이름
- 인자 (Parameter): 함수가 작업을 수행하는 데 필요한 값. 이름: 타입 형식으로 선언
- 반환타입: 함수가 작업을 마치고 반환하는 값의 타입
 - 만약 함수가 아무것도 반환하지 않으면 반환 타입을 생략하거나 Unit을 명시
 - Unit은 자바의 void와 비슷하지만 실제로는 객체

함수 선언 (fun)

- 예제: 기본적인 함수 선언

```
// 두 정수를 더하고 결과를 반환하는 함수
fun add(a: Int, b: Int): Int {
    return a + b
}

// 아무것도 반환하지 않는 함수 (반환 타입 Unit은 생략 가능)
fun printMessage(message: String): Unit {
    println(message)
}

fun main() {
    val sum = add(5, 3)
    println("5 + 3 = $sum") // 출력: 5 + 3 = 8

    printMessage("Hello, Kotlin!") // 출력: Hello, Kotlin!
}
```


기본 인자값(default parameter), 이름 붙은 인자(named argument)

- 코틀린은 함수의 호출을 더 유연하고 명확하게 만들어주는 두 가지 편리한 기능을 제공
- 기본 인자값 (Default Parameters)
 - 함수를 정의할 때 인자에 기본값 설정 ➡ 호출할 때 해당 인자 값을 생략 가능, 기본값이 사용됨
 - 예제: 기본 인자값

```
fun greet(name: String, greeting: String = "Hello") {  
    println("$greeting, $name!")  
}  
  
fun main() {  
    greet("Alice")           // greeting 인자 생략, 기본값 "Hello" 사용  
    greet("Bob", "Hi")       // greeting 인자 값 지정, "Hi" 사용  
}  
  
/* 출력:  
Hello, Alice!  
Hi, Bob!  
*/
```

기본 인자값(default parameter), 이름 붙은 인자(named argument)

- 이름 붙은 인자 (Named Arguments)

- 인자 이름과 함께 값을 전달하는 방식  함수의 가독성을 크게 높임
 - 인자의 순서를 지키지 않고도 값을 전달
 - 특히 기본 인자값과 함께 사용하면 특정 인자만 값을 바꾸고 싶을 때 매우 유용

- 예제: 이름 붙은 인자

```
fun buildUserProfile(name: String, age: Int, city: String = "Seoul") {  
    println("이름: $name, 나이: $age, 도시: $city")  
}  
  
fun main() {  
    // 순서에 상관없이 인자 이름을 명시하여 전달  
    buildUserProfile(age = 30, name = "Charlie", city = "Busan")  
  
    // 기본 인자값과 함께 사용  
    buildUserProfile(name = "David", age = 25) // city는 기본값 "Seoul" 사용  
}
```

```
이름: Charlie, 나이: 30, 도시: Busan  
이름: David, 나이: 25, 도시: Seoul
```

단일 표현식 함수 (Single-Expression Functions)

- 함수 본문이 단일 표현식으로 이루어져 있다면, 중괄호({})와 return 키워드를 생략하고 등호(=)를 사용하여 간결하게 함수를 정의할 수
 - 코드를 훨씬 깔끔하게. 코틀린의 타입 추론 기능 덕분에 반환 타입도 대부분 생략 가능
- 예제: 단일 표현식 함수

```
// 일반적인 함수 정의
fun maxOf(a: Int, b: Int): Int {
    return if (a > b) a else b
}

// 단일 표현식 함수로 변환 (타입 추론)
fun maxOfSimple(a: Int, b: Int) = if (a > b) a else b

// 두 숫자를 더하는 함수
fun sum(a: Int, b: Int) = a + b

fun main() {
    val result1 = maxOf(10, 20)
    println("Max is: $result1") // 출력: Max is: 20

    val result2 = sum(5, 7)
    println("Sum is: $result2") // 출력: Sum is: 12
}
```

고차 함수 (Higher-Order Functions), 람다 표현식 (Lambda Expressions)

- 함수를 값처럼 다룰 수 있게 해주는 개념
- 고차 함수 (Higher-Order Functions)
 - 다른 함수를 인자로 받거나, 함수를 반환하는 함수
 - 함수형 프로그래밍의 핵심 개념으로, 코드를 더 유연하고 추상적으로 작성할 수 있게
- 람다 표현식 (Lambda Expressions)
 - 이름이 없는 익명 함수
 - 주로 고차 함수에 인자로 전달될 때 사용되며, { ... } 형태로 작성

고차 함수 (Higher-Order Functions), 람다 표현식 (Lambda Expressions)

- 고차 함수 (Higher-Order Functions)
 - 함수가 변수처럼 다뤄질 수 있는 '일급 객체'라는 특성
 - 함수 타입은 (매개변수 타입) → 반환 타입 형태로 선언

```
// 고차 함수: (Int) -> Int 타입의 함수를 파라미터로 받음
fun processAnswer(operation: (Int) -> Int) {
    println(operation(42))
}

processAnswer { number -> number + 1 } // 출력: 43
```

- 자바에서는 void가 타입으로 사용될 수 없지만, 코틀린에서는 Unit이 객체이므로 () → Unit와 같이 명확한 반환 타입을 갖는 함수 타입으로 표현

고차 함수 (Higher-Order Functions), 람다 표현식 (Lambda Expressions)

- 람다 표현식 (Lambda Expressions)
 - 주로 고차 함수의 인자로 전달될 때 사용
 - 중괄호 {} 안에 매개변수 → 함수 본문 형태로 작성

```
// 함수 타입 선언: 두 개의 Int를 받아 Int를 반환하는 함수
val sum: (Int, Int) -> Int = { a, b -> a + b }

// 고차 함수: (Int) -> Int 타입의 함수를 파라미터로 받음
fun processAnswer(operation: (Int, Int) -> Int) {
    println(operation(42,48))
}

fun main() {
    processAnswer(sum)
}
```

고차 함수 (Higher-Order Functions), 람다 표현식 (Lambda Expressions)

- 람다 표현식 (Lambda Expressions)
 - 람다를 더 간결하게 작성할 수 있는 다양한 문법적 축약 기능 제공
 - it 키워드: 람다의 매개변수가 하나뿐이고 컴파일러가 타입을 추론할 수 있는 경우, it이라는 암시적 이름을 사용하여 파라미터 선언을 생략

```
val numbers = listOf(1, 2, 3, 4, 5)  
  
// 람다의 매개변수가 하나이므로 'it'을 사용  
val evens = numbers.filter { it % 2 == 0 }
```

고차 함수 (Higher-Order Functions), 람다 표현식 (Lambda Expressions)

- 실용적인 고차 함수와 람다의 응용
 - 코틀린 표준 라이브러리는 filter, map, forEach와 같은 수많은 고차 함수를 제공
 - 예를 들어, 리스트에서 0보다 큰 값만 필터링하는 작업을 수행한다고 가정. 명령형 프로그래밍 방식에서는 for 루프와 if 조건문을 조합하여 새로운 리스트를 생성해야

```
val numbers = listOf(-3, 1, -5, 2, 0, 4)
val positiveNumbers = mutableListOf<Int>()
for (num in numbers) {
    if (num > 0) {
        positiveNumbers.add(num)
    }
}
```

```
val numbers = listOf(-3, 1, -5, 2, 0, 4)
val positiveNumbers = numbers.filter { it > 0 }
```

고차 함수 (Higher-Order Functions), 람다 표현식 (Lambda Expressions)

- 실용적인 고차 함수와 람다의 응용

```
val numbers = listOf(-3, 1, -5, 2, 0, 4)
val positiveNumbers = numbers.filter { it > 0 }
```

- filter 함수는 내부적으로 컬렉션을 순회하며, 람다식이 true를 반환하는 요소들만을 새로운 컬렉션에 담아 반환
 - 이 방식은 개발자가 '어떻게(how)' 데이터를 처리할지(순회 및 조건 검사)가 아닌, '무엇을(what)' 할지(양수만 필터링)에 집중하게 하여 코드의 의도를 명확하게 드러냄

고차 함수 (Higher-Order Functions), 람다 표현식 (Lambda Expressions)

- 실용적인 고차 함수와 람다의 응용

함수명	역할	람다 형태	예제 코드	결과
filter	조건을 만족하는 요소로 이루어진 새 리스트 반환	(T) -> Boolean	listOf(1, 2, 3).filter { it > 1 }	``
map	각 요소에 변환 함수를 적용한 새 리스트 반환	(T) -> R	listOf(1, 2, 3).map { it * 10 }	``
forEach	각 요소에 대해 주어진 작업을 수행	(T) -> Unit	listOf("A", "B").forEach { println(it) }	A B
find	조건을 만족하는 첫 번째 요소 반환 (없으면 null)	(T) -> Boolean	listOf(1, 2, 3).find { it % 2 == 0 }	2
any	조건을 만족하는 요소가 하나라도 있는지 확인	(T) -> Boolean	listOf(1, 2, 3).any { it > 2 }	true
count	조건을 만족하는 요소의 개수 반환	(T) -> Boolean	listOf(1, 2, 3, 4).count { it % 2 == 0 }	2

고차 함수 (Higher-Order Functions), 람다 표현식 (Lambda Expressions)

- 람다 표현식 (Lambda Expressions)
 - 람다를 더 간결하게 작성할 수 있는 다양한 문법적 축약 기능 제공
 - it 키워드: 람다의 매개변수가 하나뿐이고 컴파일러가 타입을 추론할 수 있는 경우, it이라는 암시적 이름을 사용하여 파라미터 선언을 생략
 - 후행 람다 (Trailing Lambda): 함수의 마지막 인자가 람다일 경우, 소괄호 () 밖으로 람다를 뺄 수 있음.
 - 마치 코틀린이 제공하는 새로운 제어문처럼 보이게 하여 코드 가독성을 크게 향상시킴

```
val numbers = listOf(1, 2, 3, 4, 5)

// 람다의 매개변수가 하나이므로 'it'을 사용
val evens = numbers.filter { it % 2 == 0 }

// 후행 람다 문법 적용
val people = listOf("Alice", "Bob")
people.forEach { println("Name: $it") }
```

고차 함수 (Higher-Order Functions), 람다 표현식 (Lambda Expressions)

- 예제: 고차 함수와 람다 표현식

```
// 고차 함수 정의: 함수를 인자로 받음
fun operateOnList(list: List<Int>, operation: (Int) -> Int): List<Int> {
    val result = mutableListOf<Int>()
    for (item in list) {
        result.add(operation(item))
    }
    return result
}

fun main() {
    val numbers = listOf(1, 2, 3, 4, 5)

    // 람다 표현식 { it * 2 } 를 operateOnList 함수에 전달
    // it은 람다의 유일한 인자를 나타내는 암묵적인 이름입니다.
    val doubled = operateOnList(numbers, { it * 2 })
    println("Doubled list: $doubled") // 출력: Doubled list: [2, 4, 6, 8, 10]

    // 코틀린은 람다가 함수의 마지막 인자일 경우, 괄호 밖으로 뺄 수 있는 문법을 제공
    val filtered = numbers.filter { it > 3 }
    println("Filtered list: $filtered") // 출력: Filtered list: [4, 5]

    // 람다에 여러 인자가 있을 경우, 인자 이름을 직접 명시할 수 있음
    val sum = { a: Int, b: Int -> a + b }
    println("Sum of 5 and 10: ${sum(5, 10)}") // 출력: Sum of 5 and 10: 15
}
```


클래스와 객체 지향

클래스와 생성자

- 클래스는 객체를 생성하기 위한 청사진, 그 인스턴스는 객체
- 클래스 정의: 객체가 가져야 할 속성(데이터)과 함수(행동)를 명시하는 작업
- 생성자(constructor): 클래스의 인스턴스가 생성될 때 자동으로 호출되는 특별한 멤버 함수
- 주요 생성자(Primary Constructor)와 보조 생성자(Secondary Constructor)의 두 가지 유형의 생성자를 지원

클래스와 생성자

- 주요 생성자 (Primary Constructor)

- 클래스 헤더에 직접 선언, 클래스 이름 뒤에 괄호로 정의
- 클래스의 속성을 초기화하는 데 필요한 파라미터를 받는 것
- 클래스 내에 별도의 본문(body)을 가질 수 없기 때문에, 초기화 시점에 수행해야 할 로직은 init 블록에 작성

```
class Person(val name: String, val age: Int) {  
    // 주요 생성자에는 로직을 포함할 수 없음  
    init {  
        // init 블록은 객체 생성 시점에 실행되는 초기화 로직을 담음  
        println("A new Person object is being created with name: $name")  
        println("The age is: $age")  
    }  
}
```

클래스와 생성자

- 주요 생성자 (Primary Constructor)
 - 코틀린의 역할 분리(Separation of Concerns)를 극단적으로 추구하는 설계 철학을 반영
 - 주요 생성자는 객체의 상태(속성)를 선언하는 역할에만 집중, 실제 초기화 로직은 init 블록으로 분리
 - 코드는 더 간결해지고, 가독성이 높아짐
 - 주요 생성자의 파라미터는 init 블록과 클래스 본문 내의 속성 초기화 모두에 사용됨
 - 특히 val 또는 var 키워드를 사용하여 주요 생성자 내에서 직접 속성을 선언하고 초기화할 수 있음
 - `this.property = property`와 같은 상용구 코드 제거

클래스와 생성자

- 보조 생성자 (Secondary Constructors)
 - 클래스 본문 내부에 constructor 키워드를 사용하여 선언
 - 보조 생성자는 자체 본문을 가질 수 있으며, 그 안에 커스텀 초기화 로직을 포함
 - 한 클래스는 여러 개의 보조 생성자를 가질 수 있음
 - 매우 중요한 규칙
 - 클래스에 주요 생성자가 존재할 경우, 모든 보조 생성자는 반드시 `this()` 키워드를 통해 주요 생성자에 위임(delegate)해야 한다.
 - 모든 객체 생성 경로가 궁극적으로 주요 생성자로 수렴되도록 보장하여, 객체가 부분적으로만 초기화되는 것을 원천적으로 방지하고 코드의 일관성과 안정성을 높임

클래스와 생성자

- 보조 생성자 (Secondary Constructors)

```
class Student(val name: String, val age: Int) {  
    var studentId: String = ""  
  
    // 주요 생성자  
    init {  
        println("Student object initialized.")  
    }  
  
    // 보조 생성자: 이름과 나이를 받아 주요 생성자에 위임  
    constructor(name: String) : this(name, 20) {  
        println("Secondary constructor with name only called.")  
        this.studentId = "S_001"  
    }  
}  
  
fun main() {  
    val student1 = Student("Alice", 22)  
    val student2 = Student("Bob")  
}
```

Student("Bob")은 먼저 Student(name: String) 보조 생성자를 호출하고, 이 생성자는 다시 this(name, 20)을 통해 주요 생성자에 위임한다. 이 과정이 완료된 후 보조 생성자의 본문 로직이 실행됨

Data Class (안드로이드 모델 객체로 자주 사용됨)

- data 클래스는 주로 데이터를 담기 위한(hold) 목적으로 사용되는 특별한 유형의 클래스
- 개발자가 직접 작성해야 하는 반복적인 상용구 코드(boilerplate code)를 컴파일러가 자동으로 생성
 - 생산성을 크게 향상

Data Class (안드로이드 모델 객체로 자주 사용됨)

- 자동 생성 함수
 - data 클래스는 주요 생성자에 정의된 모든 속성들을 기반으로 유용한 함수들을 자동으로 생성
 - equals() 및 hashCode(): 객체의 속성 값들을 기준으로 동등성(equality)을 비교하는 equals() 함수, 해시 기반 컬렉션(예: HashSet, HashMap)에서 사용되는 hashCode() 함수
 - 데이터 객체를 컬렉션에 안전하게 보관하고 관리
 - toString(): "ClassName(prop1=value1, prop2=value2)"와 같은 가독성 높은 문자열을 반환
 - 디버깅 시 객체의 상태를 파악하는 데 유용
 - copy(): 기존 객체의 속성 중 일부를 변경하여 새로운 인스턴스를 생성
 - 불변성(immutability)을 유지하면서 객체의 상태를 업데이트하는 현대적인 프로그래밍 패턴을 지원
 - componentN(): 주요 생성자의 각 속성에 대응하는 component1(), component2() 등의 함수를 생성
 - 구조 분해 선언(destructuring declarations)을 가능하게

Data Class (안드로이드 모델 객체로 자주 사용됨)

- 요구 사항과 제약 사항
 - 주요 생성자는 최소 하나 이상의 파라미터를 가져야 한다.
 - 주요 생성자의 모든 파라미터는 `val` 또는 `var`로 선언되어야 한다.
 - `data` 클래스는 `abstract`, `open`, `sealed`, 또는 `inner`로 선언할 수 없다.
 - 주요 생성자 외부에 선언된 속성들은 자동 생성 함수들에 포함되지 않는다.
 - 이로 인해 두 객체가 `equals()` 비교를 통해 동일하다고 판단되더라도, 클래스 본문에 선언된 속성들의 값은 다를 수 있다.
 - 객체의 '식별'을 위한 핵심 데이터와 부가적인 상태를 분리

Data Class (안드로이드 모델 객체로 자주 사용됨)

```
data class User(val name: String, val age: Int) {
    var isRegistered: Boolean = false
}

fun main() {
    val user1 = User("John", 42)
    val user2 = User("John", 42)

    user1.isRegistered = true
    user2.isRegistered = false

    // equals()는 isRegistered 속성을 무시하므로 true를 반환한다.
    println("user1 == user2: ${user1 == user2}") // Output: true

    // copy() 함수를 사용하여 불변 객체를 수정
    val olderUser = user1.copy(age = 43)
    println("Older user: $olderUser") // Output: User(name=John, age=43)

    // 구조 분해 선언
    val (name, age) = user1
    println("User name is $name and age is $age")
}
```

Data Class (안드로이드 모델 객체로 자주 사용됨)

- data 클래스는 특히 안드로이드 개발에서 모델 객체로 자주 사용
 - 선언적 UI(Declarative UI) 패러다임과 밀접한 관련
 - Jetpack Compose와 같은 프레임워크는 상태(State)의 변화를 감지하여 UI를 업데이트하며, 이때 불변 객체로 상태를 관리하는 것이 권장
 - data 클래스의 `copy()` 함수는 불변 객체의 일부 속성만 변경하여 새로운 객체를 생성하는 데 최적화되어 있어, 상태 업데이트를 안전하고 효율적으로 수행하도록 돕는다.

Data Class (안드로이드 모델 객체로 자주 사용됨)

일반 클래스와 데이터 클래스

비교

특징	일반 class	data class
목적	객체의 상태와 행동을 모두 정의	주로 데이터를 담는 컨테이너
equals(), hashCode()	기본적으로 참조(reference) 비교	주요 생성자 속성 기반으로 값 비교
toString()	메모리 주소 기반의 문자열 반환	"ClassName(prop1=value1,...)" 형식의 문자열 반환
copy()	지원하지 않음 (직접 구현 필요)	자동으로 생성되어 불변성 유지에 도움
componentN()	지원하지 않음	자동으로 생성되어 구조 분해 선언 가능
상용구 코드	다수 필요	최소화

상속 (Inheritance) - open와 override

- 코틀린의 상속 모델은 자바와 근본적으로 다른 철학
- 자바에서는 모든 클래스와 멤버(함수, 속성)가 기본적으로 상속 및 오버라이딩에 열려(open) 있는 반면,
- 코틀린에서는 모든 것이 기본적으로 final
- 이처럼 상속을 기본적으로 금지하는 설계는 "안정성"과 "의도적인 설계"를 장려한다.

상속 (Inheritance) - open와 override

- final by default의 의미
 - 상속은 부모 클래스의 내부 구현에 대한 의존성을 높여 코드를 복잡하게 만들고
 - 예상치 못한 오버라이딩으로 인해 잠재적인 버그를 유발할 수 있다
 - 코틀린은 이러한 위험을 줄이기 위해, 상속을 "특권"으로 취급
 - 개발자는 클래스나 멤버를 상속 가능하게 만들려면 명시적으로 open 키워드를 사용해야
 - "이 클래스는 상속을 염두에 두고 설계되었고, 하위 클래스가 안전하게 확장될 수 있도록 보장한다"는 명확한 계약을 제시하는 효과

상속 (Inheritance) - open와 override

- open 키워드: 상속 허용
 - 클래스를 상속 가능하게 만들려면 class 키워드 앞에 open을 붙여야 한다.

```
open class Animal {  
    // 이 함수는 open으로 명시하지 않으면 final이 된다.  
    open fun makeSound() {  
        println("Animal makes a sound")  
    }  
}
```

- 클래스의 모든 멤버가 자동으로 오버라이딩 가능해지는 것은 아니다.
- 함수나 속성 또한 개별적으로 open 키워드를 붙여야만 자식 클래스에서 재정의할 수 있다.

상속 (Inheritance) - open와 override

- override 키워드: 재정의
 - 자식 클래스에서 재정의할 때는 반드시 override 키워드를 사용해야

```
class Dog : Animal() {  
    // override 키워드를 사용하여 부모 클래스의 open 함수를 재정의한다.  
    override fun makeSound() {  
        println("Dog barks")  
    }  
}
```

- 재정의된 멤버는 기본적으로 암묵적으로 open 상태. 따라서 Dog 클래스의 자식 클래스도 makeSound() 함수를 오버라이드할 수 있다.
- 만약 오버라이드된 멤버가 더 이상 재정의되지 않도록 막고 싶다면 final override를 사용하면 된다.

접근 제어자 (Access Modifiers) - public, private, protected, internal

- 접근 제어자는 클래스, 생성자, 함수, 속성 등의 가시성(visibility)을 제어하여 코드의 캡슐화(encapsulation)를 돕는 중요한 도구
- 코틀린은 네 가지 접근 제어자를 제공
 - public
 - 코틀린의 기본 가시성
 - public으로 선언된 멤버는 어떤 코드에서든 접근이 가능
 - private
 - 가시성을 가장 엄격하게 제한
 - 클래스 멤버: private로 선언된 멤버는 오직 그 멤버가 선언된 클래스 내부에서만 접근 가능
 - 최상위 선언: 코틀린은 최상위(top-level)에 함수, 속성, 클래스를 선언하는 것을 허용
 - private는 해당 선언이 포함된 파일 내부에서만 접근 가능하도록 제한
 - 특정 파일에 국한된 헬퍼 함수나 변수를 정의하여 불필요한 노출을 막고 코드의 응집도를 높임

접근 제어자 (Access Modifiers) - public, private, protected, internal

- 코틀린은 네 가지 접근 제어자를 제공
 - protected
 - protected 멤버는 선언된 클래스 내부와 그 자식 클래스에서만 접근 가능
 - 자바와 달리, 같은 패키지 내에서는 접근이 허용되지 않음
 - 최상위 선언에는 사용할 수 없음
 - internal
 - internal로 선언된 멤버는 동일한 모듈(module) 내부에서만 접근 가능
 - 모듈은 함께 컴파일되는 코틀린 파일들의 집합 (예: IntelliJ IDEA 모듈, Gradle 소스셋)
 - internal은 논리적인 컴포넌트 단위(모듈)로 캡슐화를 제공

접근 제어자 (Access Modifiers) - public, private, protected, internal

코틀린 접근 제어자

비교

접근 제어자	가시성 범위	자바와의 차이점
public	모든 코드	동일
private	클래스/파일 내부	최상위 선언의 가시성 범위가 파일로 확장됨
protected	클래스 내부 및 자식 클래스	같은 패키지 내 접근이 허용되지 않음
internal	동일한 모듈 내부	코틀린 고유 (자바의 package-private 대체)

object 키워드 (Singleton, Anonymous Object)

- object 키워드는 코틀린에서 클래스 정의와 동시에 인스턴스 생성을 수행하는 데 사용되는 강력한 기능
- 싱글톤 객체, 익명 객체, 그리고 컴패니언 객체 등 세 가지 주요 패턴을 구현하는 데 활용

object 키워드 (Singleton, Anonymous Object)

- object 선언: 싱글톤 패턴 (Object Declaration: The Singleton Pattern)
 - 싱글톤 패턴은 애플리케이션 내에서 오직 하나의 인스턴스만 존재해야 하는 경우에 사용
 - object 키워드를 클래스 이름 앞에 붙여 선언하면, 컴파일러가 클래스 정의와 동시에 해당 클래스의 유일한 인스턴스를 생성
 - 자바에서 싱글톤을 구현하기 위해 필요했던 복잡한 보일러플레이트 코드를 제거하여, 의도를 매우 명확하게 만듦

```
// 싱글톤 객체 선언
object DatabaseManager {
    fun connect() {
        println("Connecting to the database...")
    }
}

fun main() {
    DatabaseManager.connect() // 인스턴스 생성 없이 직접 접근
}
```

object 키워드 (Singleton, Anonymous Object)

- object 표현식: 익명 객체 (Object Expression: Anonymous Objects)
 - object 표현식은 이름 없이 object 키워드를 사용하여 인터페이스나 추상 클래스를 즉석에서 구현하는 일회용 인스턴스를 생성하는 데 사용
 - 이는 자바의 익명 내부 클래스(anonymous inner class)와 유사하지만, 훨씬 간결한 문법을 제공

```
interface ClickListener {  
    fun onClick()  
}  
  
fun setClickListener(listener: ClickListener) {  
    listener.onClick()  
}  
  
fun main() {  
    // object 표현식으로 ClickListener 인터페이스의 익명 객체 생성  
    setClickListener(object : ClickListener {  
        override fun onClick() {  
            println("Button clicked!")  
        }  
    })  
}
```

object 키워드 (Singleton, Anonymous Object)

- 컴패니언 객체: static 멤버 대체 (Companion Object: Replacing static Members)
 - 코틀린은 자바의 static 멤버 대신 companion object를 사용하여 클래스에 종속된 멤버를 인스턴스 생성 없이 클래스 이름을 통해 직접 접근할 수 있게 한다.
 - 컴패니언 객체는 클래스 내부에 선언되는 특별한 객체
 - 컴패니언 객체는 그 자체로 하나의 객체이므로 이름을 가질 수 있고, 인터페이스를 구현하거나 다른 클래스를 상속받을 수 있다. 이는 자바의 static 멤버로는 불가능한 기능이다.

object 키워드 (Singleton, Anonymous Object)

- 컴패니언 객체: static 멤버 대체 (Companion Object: Replacing static Members)

```
class User private constructor(val name: String) {  
    // 컴패니언 객체: static 멤버와 유사한 역할  
    companion object Factory {  
        fun create(name: String): User {  
            return User(name)  
        }  
    }  
}  
  
fun main() {  
    // 팩토리 메서드를 통해 객체 생성  
    val user = User.Factory.create("Alice")  
    println(user.name) // Output: Alice  
}
```


object 키워드 (Singleton, Anonymous Object)

object 키워드 사용법
비교

용법	목적	인스턴스화	접근 방식
object 선언	싱글톤 패턴 구현	컴파일러가 단일 인스턴스 생성	이름으로 직접 접근
object 표현식	일회성 익명 객체 생성	코드 실행 시 즉석에서 생성	주로 변수 할당 또는 함수 인자
companion object	클래스 레벨 멤버 정의	클래스 로드 시 단일 인스턴스 생성	클래스 이름으로 접근