
Application Development for Mobile Computer

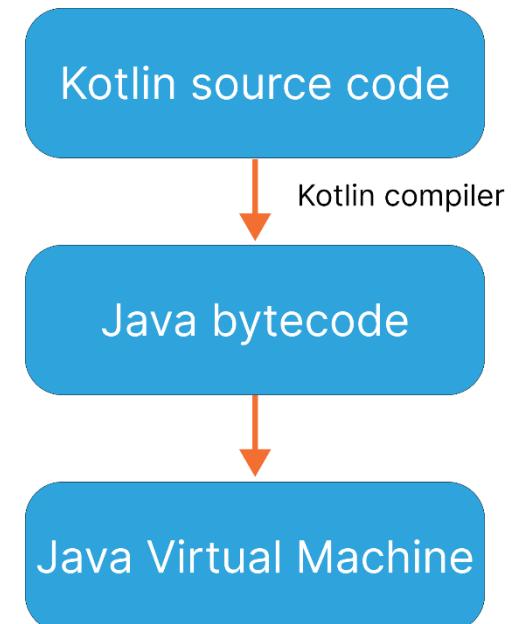
<Week 3>

Youn Kyu Lee

Kotlin

Kotlin

- Developed by JetBrains as an open-source project
- Officially recognized by Google in 2017 as an Android language
- The Kotlin compiler (`kotlinc`) compiles .kt files into Java bytecode
- Advantages
 - Concise syntax with modern language features
 - Built-in support for null safety
 - Fully interoperable with Java
 - Simplifies asynchronous programming using coroutines



Kotlin

- Kotlin file structure(User.kt)

```
package com.example.test3
```

Package

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

Import

```
var data = 10
```

Variable

```
fun formatDate(date: Date): String {  
    val sdformat = SimpleDateFormat("yyyy-mm-dd")  
    return sdformat.format(date)  
}
```

Fuction

```
class User {  
    var name = "hello"  
  
    fun sayHello() {  
        println("name : $name")  
    }  
}
```

Class

Kotlin

- Example with a matching package path(Test.kt)

```
package com.example.test3

import java.util.*

fun main() {
    data = 20
    formatDate(Date())
    User().sayHello()
}
```

- Example with a mismatched package path

```
package ch3

import com.example.test3.User
import com.example.test3.data
import com.example.test3.formatDate
import java.util.*
```

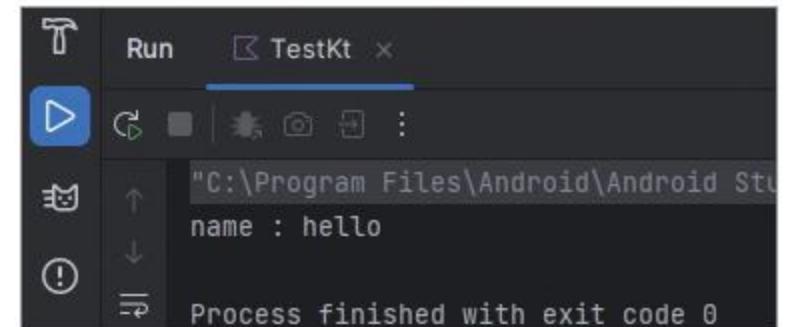
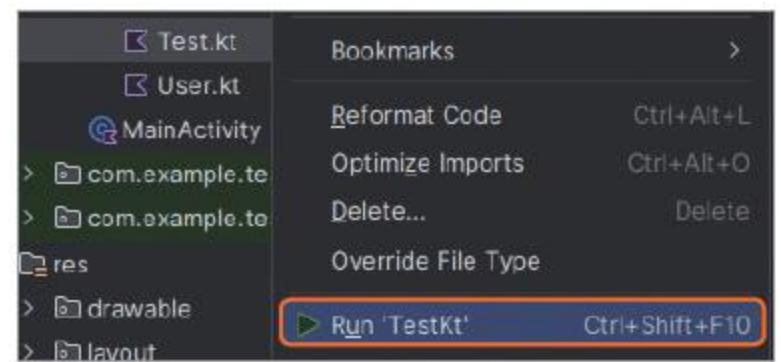
Since it's in a different package,
you need to import it.

```
fun main() {
    data = 20
    formatDate(Date())
    User().sayHello()
}
```

Kotlin

Testing Kotlin

```
• Kotlin source file • Test.kt
fun main() {
    println("hello world")
}
```



Kotlin

Var

- A variable is a named storage in memory (temporary storage space).
- Declare a variable and assign a value at the same time.
 - Kotlin uses type inference: the type (String, Int, Boolean, etc.) is determined automatically when the value is assigned.

```
var variable_name = value
```

- Variables can be declared without assigning a value initially.
 - In this case, you must specify the type explicitly by placing a colon (:) after the variable name.

```
var variable_name: type  
variable_name = value
```

Kotlin

Data Type

Category	Data Type	Description	Value Range & Examples
Numeric	Double	Stores decimal values	64-bit floating-point Approx. $-1.7\text{E}+308 \sim 1.7\text{E}+308$
	Float	Same as Double but smaller range	32-bit floating-point Approx. $-3.4\text{E}+38 \sim 3.4\text{E}+38$
	Int	Stores integers without decimals	32-bit integer $-2,147,483,648 \sim 2,147,483,647$
	Long	Stores larger integers than Int	64-bit integer $-2^{63} \sim 2^{63} - 1$
	Short	Stores integer values	16-bit integer $-32,768 \sim 32,767$
	Byte	Stores integer values	8-bit integer $-128 \sim 127$
Character	Char	Single character in single quotes	'A'
	String	Multiple characters	"This is a string."
Boolean	Boolean	Logical values	true or false

Kotlin

Val

- Unlike var, a value assigned to val cannot be changed.
- Declared in the same way as variables, but prefixed with val to make it read-only.

```
val variable_name = value
```

- Useful for storing immutable values that can later be combined with other values.

```
val roadName = "Gukjegeumyung-ro"
val address = roadName + " 8-gil"
```

- Since variables defined with val cannot be reassigned, the following input will cause an error.

```
val language = "kotlin"
language = "java"
```

Const

- Constants are mainly used to store fixed values that serve as a reference.
- Declared by adding the `const` keyword in front of a read-only variable (`val`).
- Similar to `val` (read-only), but the value is **determined at compile time**.
- Only primitive types (e.g., `Int`, `Long`) and `String` can be assigned to constants.

```
const val PI = 3.141592
```

Kotlin

Coding Convention

- Class Names
 - Each project follows specific coding rules for naming.
 - Class names generally follow the Camel Case convention.
 - The first letter of each word is capitalized, while the rest are lowercase.

```
class MainActivity
```

- Function and Variable Names
 - Also follow Camel Case; The first letter of the first word is lowercase; The first letter of subsequent words is capitalized.

```
fun onCreateActivity( )
```

```
var intValue: Int
```

Coding Convention

- Constant Names
 - Written in all uppercase letters.
 - If a constant name consists of two or more words, use snake case with underscores (_) to separate words

```
const val HOW_ARE_YOU: String = "How are you?"
```

- Indentation
 - When a new code block begins, apply consistent indentation using either spaces or the tab key.

Kotlin

If

- Using if Statements
 - Executes code block only if condition is true
 - Comparison operators: >, <, >=, <=, ==, !=
 - Logical operators: &&, ||, !
- Forms of if
 - Basic if – executes code when condition is true
 - if ~ else – executes one block if true, otherwise the other
 - if ~ else if ~ else: evaluates next condition only if previous is false
 - Assigning if result to variable: the last expression in the block becomes the value

```
if (condition) {  
    The code block that executes when the condition is true.  
}
```

Code block

When

- Extended form of if that adds range comparison, similar to switch but more powerful
- Basic when
 - Write parentheses () after when
 - Place the variable to be compared inside the parentheses
- Using Commas
 - If multiple values share the same result, separate them with commas
- Range Comparison
 - Use in to compare a range of values
 - Can implement functionality similar to <=, >= in if statements
- when Without Parameters
 - Parentheses can be omitted
 - Works like an if statement

If vs When

- Year Data
 - Example: 2019, 2020, 2021, 2022, 2023 ...
 - Number of values is large and range is not fixed
 - Use if when the range is broad and values cannot be specifically limited
- Day of the Week Data
 - Example: Mon, Tue, Wed, Thu, Fri, Sat, Sun
 - Number of values is fixed and specific (7)
 - Use when when the range is limited and values are specific

```
when (dayOfWeek) {  
    "Mon" → "Study English."  
    "Tue" → "Go to the bike club meeting."  
    "Wed" → "Meet friends."  
    "Thu" → "Play the piano."  
    "Fri" → "Stay up all night coding."  
    "Sat" → "Do the laundry."  
    else → "Clean the house."  
}
```

Array

- Before storing values, the array size must be allocated or determined by the number of initial values.
- The size must be fixed in advance and cannot be increased or decreased later.
- Arrays can be stored in variables like other data types.

```
var variable = Array(size)
```

- Character Array Allocation
 - Allocate empty array space
- Assigning Array with Values
 - Assign array space directly with values
- Inserting Values into Array
 - Use the assignment operator = or the set function
- Retrieving Values from Array
 - Access values using their index

Collection

- In addition to Array, Kotlin provides data types that can store multiple values.
- A collection is also called a dynamic array because, unlike arrays, its size is not fixed at creation and it can hold an arbitrary number of elements.
- Collections include List, Set, Map.
- Elements in Collections
 - The basic unit of a collection is an element.
 - The structure of elements depends on the collection type.

List element = value of the list

Map element = key and value of the map
- Use size to retrieve the number of elements.

Kotlin

List

- A collection where each stored value has an index
- Allows duplicate values
- Use the prefix `mutable` in front of the List type
- Functions for List
 - Create a list: `mutableListOf()`
 - Add a value: `add()`
 - Access a value: `get()`
 - Modify a value: `set()`
 - Remove a value: `removeAt()`
 - Create an empty list: `mutableListOf<Type>()`

Set

- A List that does not allow duplicates
 - Similar structure to List, but cannot be accessed by index
 - Does not support the get() function
- Initialize an empty set and add values

```
var variable = emptySet<Type>()
```

- Using Set
 - Since Set has no index-based access, values at specific positions cannot be retrieved directly
- Deleting from Set
 - Because Set does not allow duplicates, values can be deleted directly by value

Map

- A collection of key–value pairs
- Unlike List, both the key type and the value type must be specified when creating a Map
- Creating a Map
 - Use generics to specify the data types of the key and value
- Creating an Empty Map and Adding Values
 - Use the put() function with a key and a value to add entries
- Using a Map
 - Use the get() function with a key to retrieve a value
- Modifying a Map
 - If a value with the same key already exists, put() updates the value while keeping the key
- Deleting from a Map
 - Use the remove() function with a key to delete the entry

Immutable Collection

- Same as existing Collections, but without the mutable prefix
- Once created, values cannot be changed

```
var list = mutableListOf("1", "2")
```



```
var list = listOf("1", "2")
```

- Functions like add() or set() are not supported
 - Modification, addition, or removal of elements is not allowed
- Example Use Case
 - Declaring the 7 days of the week as an Immutable List ensures the values remain constant

```
val DAY_LIST = listOf("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")
```

- Immutable Collections should be declared with val (Variable names should be written in uppercase)

Kotlin

for

- Used to repeat code a specific number of times

```
for (variable in startValue..endValue) {  
    // code block  
}
```

- for in .. : General form of for loop
- until : Repeats up to but excluding the last number
- step : Skips numbers by a step value
- downTo : Decreases values in reverse order
- Iterate through elements in an Array or Collection

```
val items = arrayOf("a", "b", "c")  
for (item in items) {  
    // code block  
}
```

Kotlin

while

- A statement used to repeat code until a condition is satisfied
- Can be considered as a repeatable form of if

```
while (condition) {  
    // code block  
}
```

- Standard while Loop
 - Unlike the for loop, if an index needs to increase or decrease, it must be handled manually in the code
- do ~ while Loop
 - Executes the code inside the do block at least once, regardless of the condition
 - Differs from the standard while loop because it still executes once even if the initial value does not satisfy the condition

Controlling Loops

- Used when a loop must exit early or skip to the next iteration under certain conditions
- `break`
 - Exits the loop immediately
 - When `break` is encountered inside a loop, the loop terminates
 - Used when a condition requires leaving the `for` block entirely
- `continue`
 - Skips the remaining code in the current iteration
 - Control jumps back to the beginning of the loop for the next iteration

```
for (except in 1..10) {  
    if (except > 3 && except < 8) {  
        continue  
    }  
    Log.d("continue", "current index is $except.")  
}
```

Function

- Defined using the fun keyword
- Can receive values as parameters (inputs)

```
fun function_name(type of parameter: type): return type {  
    return value  
}
```

- Function with parameters and return value
- Function without return value
- Function with return value but without parameters

Function Parameters

- Defined in the form name: Type
- Multiple parameters are separated by commas
- All parameter values are immutable
 - Parameters can be considered as having an implicit val

```
fun function_name((omit val) name1: String, name2: Int, name3: Double) { executable code }
```

- Default Parameter Values
 - Use the = operator when defining a parameter to set a default value

```
fun function_name(name1: String, name2: Int = 157, name3: Double) { executable code }
```

- Named Arguments
 - When a function has many parameters and the meaning of values may be unclear
 - Values can be assigned by explicitly specifying the parameter name, regardless of order

Kotlin

Class

- A way to group functions and variables together under a single name for easier use

```
Class class {  
    var variable  
    fun function() {  
        // code block  
    }  
}
```

- Example: String Class

- Return string length: length()
- Concatenate strings: plus()
- Compare strings: compare()

```
Class String {  
    var length: Int  
    fun plus(other: Any) {  
        // code block  
    }  
    fun compare(other: Any) {  
        // code block  
    }  
}
```

Writing a Class

- Define a class name and use the class keyword in front of it
 - Use curly braces {} (class scope) to define the boundaries of the class

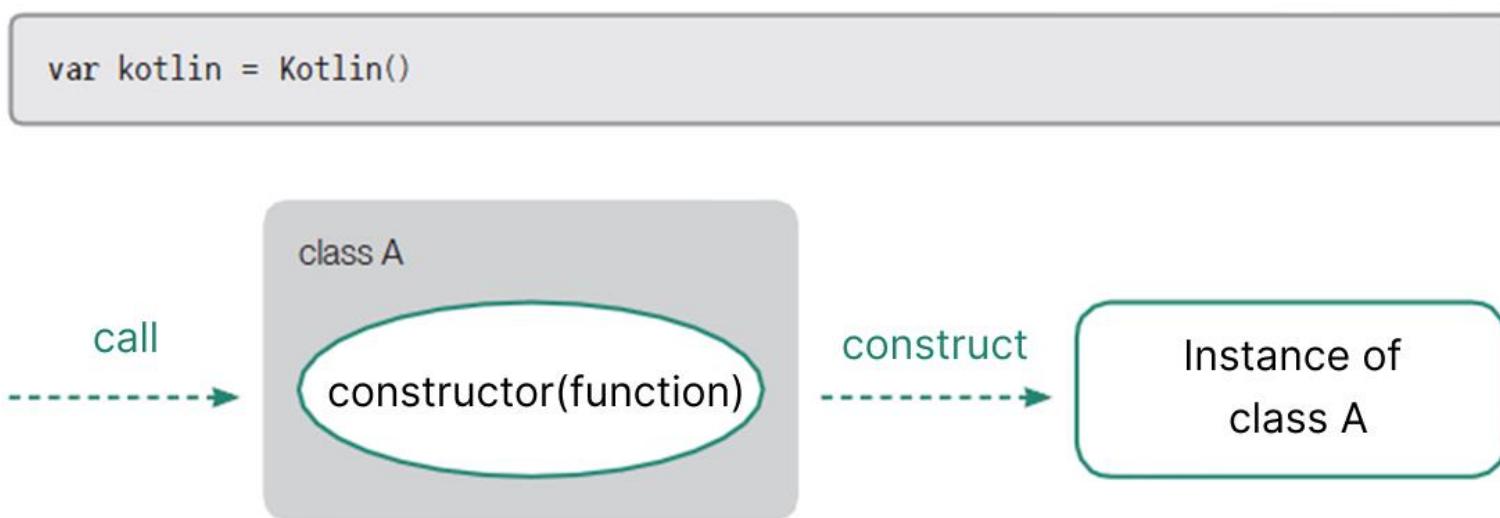
```
Class ClassName {  
    // class scope  
}
```

- Primary Constructor
 - Works like the header of a class
 - Defined with the constructor keyword (can be omitted in some cases)
- Secondary Constructor
 - Declared inside the class scope using the constructor keyword
 - Can be overloaded if the number or type of parameters differs
- Default Constructor
 - If no constructor is defined, a parameterless primary constructor is provided by default

Kotlin

Using a Class

- Add parentheses () after the class name to call the constructor and create an instance
- The created instance can be stored in a variable and used



- Accessing Functions and Variables in a Class
 - Using a class means accessing the variables and functions defined inside it
 - Through the constructor, the created instance stored in a variable can access internal members using the dot operator (.)

Object

- An object allows you to use properties and methods inside its block without instantiating the class with a constructor
 - Properties and methods inside the object block can be accessed directly with the class name and dot operator
 - Unlike classes, only one instance of an object exists throughout the entire app
- Companion Object
 - A companion object is used to add object functionality inside a regular class

Data Class

- Used for simple value storage
 - Created by adding the data keyword in front of the class definition
 - Parameters are defined after the class name
 - Unlike regular constructors, each parameter must explicitly specify var or val to indicate mutable or immutable

```
data class class_name (val parameter1: type, var parameter2: type)
```

- Defining and Creating a Data Class
 - Use the data keyword before class
 - The var (or val) keyword in constructor parameters cannot be omitted
 - Parameters defined with val are read-only, like normal variable declarations
 - `toString()` in a data class returns the stored values, while in a regular class it returns the instance address
- Works like a normal class: calling the constructor executes the init block and methods can be used

Inheritance (1)

- Inheritance supports the reuse of classes
 - It provides another way of using class resources, similar to accessing methods and properties with the dot operator (.) after creating a class
 - With inheritance, methods and properties of the parent class can be used as if they are part of the child class
- Class Inheritance
 - Use the open keyword to declare a parent class and allow its methods and properties to be inherited
 - Since inheritance means that the child contains the parent instance, the parent constructor must be called by adding parentheses () after the parent class name

```
open class ParentClass {  
    // code  
}  
  
class ChildClass : ParentClass() {  
    // code  
}
```

Inheritance (2)

- Inheriting a Class with Constructor Parameters
 - If the parent class has constructor parameters, values can be passed through the child class constructor
- Using Parent Properties and Methods
 - Properties and methods defined in the parent class can be used directly in the child class
- Inherited properties and methods can be redefined using override
 - When redefinition is needed, use the override keyword
 - Methods and properties must be marked with open in the parent class to allow overriding
- Extensions in Kotlin
 - Kotlin supports extensions for classes, methods, and properties

Design Tools (1)

- Managing files during coding—classifying, naming, and organizing them into directories—is all part of design
- Package
 - A package is a directory structure used to manage classes and source files
 - Since multiple files can be created in one package, related files should be placed together for easier management
- Abstraction
 - When the code design is not yet clear, only method names are written while the implementation is deferred
 - Use the abstract keyword to declare abstraction
- Interface
 - An interface is like an abstract class that contains only method names without implementation code

Design Tools (2)

- Visibility Modifiers
 - Classes, interfaces, functions, and variables can all have visibility modifiers

Modifier	Scope of Access
private	Not accessible from other files
internal	Accessible only within the same module
protected	Same as private, but accessible from child classes in inheritance
public	Accessible from all files without restriction

- Applying Visibility Modifiers
 - When applied, they restrict the use of the corresponding class, member property, or method

Kotlin

null

- Incorrect handling of null can cause program malfunctions and threaten stability
- Kotlin provides null safety mechanisms to ensure safe handling of null
- Allowing null in Variables
 - If a type does not have a ? after it, null cannot be assigned
 - Add ? after the type to allow null
- Allowing null in Function Parameters
 - Like the Bundle parameter in Android's onCreate() method, function parameters can also allow or disallow null
 - If a parameter allows null, a null check must be performed in the function before using it
- Allowing null in Return Types
 - Add ? after the return type to allow null values to be returned

Safe Call

- Provides a concise way to perform null checks
- When ?. is used after a nullable variable:
 - If the variable is null, the method or property after ?. is not called
 - If the variable is not null, the method or property is executed normally

```
fun testSafeCall(str: String?): Int? {  
    // If str is null, it does not check length and returns null.  
    var resultNull: Int? = str?.length  
    return resultNull  
}
```

Replacing Null Values

- Use the Elvis Operator (?:) to set a default value when the original variable is null
- If the variable is null, the default value is returned
- If the variable is not null, its value is returned

```
fun testElvis(str: String?): Int {  
    // When using ?: on the right side of length, if it is null, the value on the right side of ?: is returned.  
    var resultNotNull: Int = str?.length?:0  
    return resultNotNull  
}
```

Nullable(?), Safe Call(?.), Elvis Operator(?:)

Nullable	<ul style="list-style-type: none">Notation: Add ? after the variable type
	<ul style="list-style-type: none">Purpose: Allow null as a value
	<ul style="list-style-type: none">Example: var nullable: Type?
Safe call	<ul style="list-style-type: none">Notation: Add ?. after the variable name
	<ul style="list-style-type: none">Purpose: Skip the property or command if the variable is null
	<ul style="list-style-type: none">Example: var result = variable?.length or variable?.property?.something
Elvis operator	<ul style="list-style-type: none">Notation: Add ?: after the variable name
	<ul style="list-style-type: none">Purpose: Use the value after ?: as a default when the variable is null
	<ul style="list-style-type: none">Example: var result = variable ?: 0 or variable?.property ?: 0

lateinit

- Prevents overuse of nullable (?) handling in class code
 - Used when a class property must be declared as nullable first, but initialized later (e.g., in the constructor or another method)
 - By using lateinit, safe calls (?.) are avoided, improving code readability
- Normally, a variable is declared with ? and initialized with null
 - Later, the value is assigned in another method of the class
 - However, accessing such variables requires frequent safe calls (?.), reducing readability
- Features of lateinit
 - Can only be used with class properties declared with var
 - null is not allowed
 - Cannot be used with primitive types such as Int, Long, Double, Float

lazy

- Lazy Initialization with `lazy`
 - Provides lazy initialization for read-only variables (`val`)
 - Unlike `lateinit` (which can be reassigned), a variable declared with `lazy` cannot be changed
 - Declare the variable with `val` and use the `by lazy { }` syntax to specify the initialization code
- Features of `lazy`
 - The variable is initialized at the time of its first call, using the value inside `by lazy { }`
 - Initialization code is written together at declaration, so no separate initialization is required later

Scope functions

- Functions that help simplify and shorten code, also called scope functions
- `run`
 - Within the scope function, the target object is referenced as `this`
 - Works like calling functions inside a class
 - Methods and properties can be called directly without `this`
- `let`
 - Within the scope function, the target object is referenced as `it`
 - it cannot be omitted, but can be replaced with another name (e.g., `target`)

Q & A

aiclasscau@gmail.com