Kotlin Programming II

Asst. Prof. Monlica Wattana, Ph.D
Department of Computer Science,
Khon Kaen University

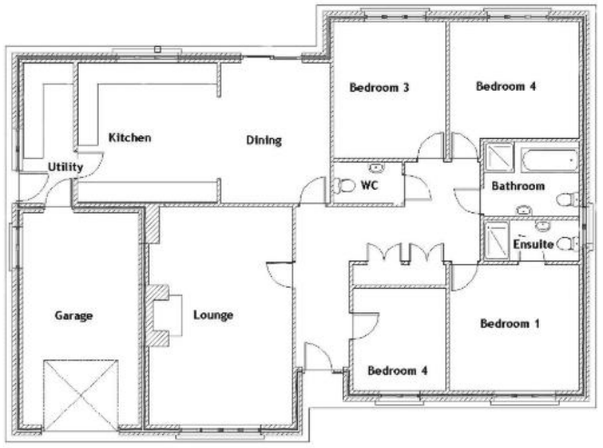
SC 362 007 Mobile Device Programming

Outline

- Class and Object
- Constructors
- Inheritance
- Data Class
- Companion Objects
- Extension function
- Lambda

Class & Object

- Kotlin supports object-oriented programming.
- A class is a blueprint for the object.
- A sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc.
- Based on these descriptions and build the house. House is the object.



Define a class in Kotlin

These objects share two characteristics:

- state
- behavior

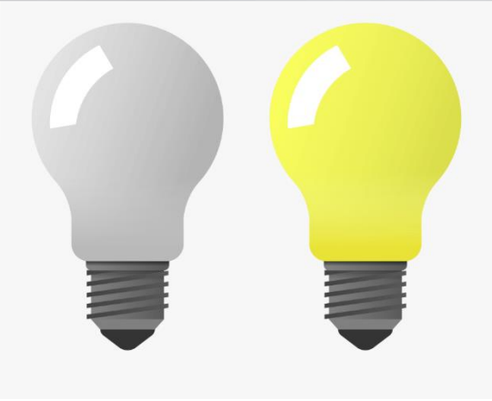
- To define a class in Kotlin, class keyword is used:

```
class ClassName {
    // state or property
    // behavior or member function
    ... .. ...
}
```

Define a class in Kotlin

Examples: Lamp is an object

- It can be in *on* or *off* state.
- You can *turn on* and *turn off* lamp (behavior).
- It can show status: *on* or *off* state (behavior).



```
class Lamp {
    // state or property (data member)
    private var isOn: Boolean = false

    // behavior or member function
    fun turnOn() {  isOn = true  }

    // behavior or member function
    fun turnOff() {  isOn = false }

    // behavior or member function
    fun displayLightStatus() {
        if (isOn == true)
            println("lamp is on.")
        else
            println("lamp is off.")
    }
}
```

Define an object in Kotlin

- When class is defined, only the specification for the object is defined; no memory or storage is allocated.
- To access members defined within the class, we need to create objects.

```
fun main() {
    val lamp1 = Lamp()  // create object of Lamp class
    ////   Turn on
    lamp1.turnOn()
    lamp1.displayLightStatus()
    ////   Turn off
    lamp1.turnOff()
    lamp1.displayLightStatus()
}
```

Result

Kotlin Constructors

A constructor is a concise way to initialize class properties.

There are two constructors:

1. **Primary constructor -** concise way to initialize a class
2. **Secondary constructor -** allows putting additional initialization logic

Kotlin Constructors

1. Primary Constructor

   The primary constructor is part of the class header.

```
class Person(val firstName: String, var age: Int)
{
    // class body
}
```

Kotlin Constructors

Ex. Primary Constructor

```
class Person(val firstName: String, var age: Int) {
}
fun main() {
    val person1 = Person("Joe", 25)
    println("First Name = ${person1.firstName}")
    println("Age = ${person1.age}")
}
```

Result

Kotlin Constructors

Note: Ex. Primary Constructor

```
class Person( firstName: String, age: Int) {
}
fun main() {
    val person1 = Person("Joe", 25)
    println("First Name = ${person1.firstName}")//ERROR
    println("Age = ${person1.age}") //ERROR
}
```

Result ERROR

Kotlin Constructors

1. Primary Constructor

- Initializer Blocks

```
class Person(fName: String, personAge: Int) {
    val firstName: String
    var age: Int
    // initializer block
    init {
            firstName = fName.capitalize()
             age = personAge
            println("First Name = $firstName")
            println("Age = $age")
    }
}
```

```
fun main() {
    val person1 = Person("alice", 21)
}
```

Result

```
First Name = Alice
Age = 21
```

Kotlin Constructors

1. Primary Constructor

Default Value in Primary Constructor

```
class Person(_firstName: String = "UNKNOWN", _age: Int = 0) {
    val firstName = _firstName.capitalize()
    var age = _age
    // initializer block
    init {
        println("First Name = $firstName and Age = $age ")
            }
}
```

```
fun main() {
    println("person1 : ")
    val person1 = Person("Alice", 21)
    println("person2  : ")
    val person2 = Person("Bob")
    println("person3 : ")
    val person3 = Person(_age = 25)
}
```

Result

Kotlin Constructors

2. Secondary Constructor

To extend a class that provides multiple constructors that initialize the class in different ways.

```
class Person(var firstName: String) {
      var age: Int? = null
      var phoneNumber: String? = null
   // Secondary Constructor
   constructor(firstName: String, age: Int): this(firstName)  {
       this.age = if(age > 0) age else 0
   }
   // Secondary Constructor
   constructor(firstName: String, age: Int, phoneNumber: String): this(firstName, age)  {
          this.phoneNumber = phoneNumber
   }
}
```

Kotlin Constructors

2. Secondary Constructor (cont.)

```
fun main() {
   // Calls the primary constructor (Age will be null in this case)
    val person1 = Person("Bill")
    println("Person1 :  ${person1.firstName}  ")
    // Calls the secondary constructor
     val person2 = Person("Jeff", 15)
     println("Person2 :  ${person2.firstName},  Age : ${person2.age}")
    // Calls the secondary constructor
    val person3 = Person("Toby", 40,"056-9440042")
    println("Person3 :  ${person3.firstName},  Age : ${person3.age},  Phone  Number : ${person3.phoneNumber}")
}
```

Result

```
Person1 :  Bill
Person2 :  Jeff,  Age : 15
Person3 :  Toby,  Age : 40, Phone  Number : 056-9440042
```

Getters and Setters

- Getters are used for getting value of the property.
- Setters are used for setting value of the property.
  - In Kotlin, getters and setters are optional

```
class Person {
    var name: String = "defaultValue"
}
```

```
class Person {
    var name: String = "defaultValue"

    // getter
    get() = field

    // setter
    set(value) {
      field = value
    }
}
```

Getters and Setters

```
class Girl {
    var age: Int = 0
    get() = field
    set(value) {
        field = if (value < 18)
            18
        else if (value >= 18 && value <= 30)
            value
        else
            value-3
    }
    var actualAge: Int = 0  }
```

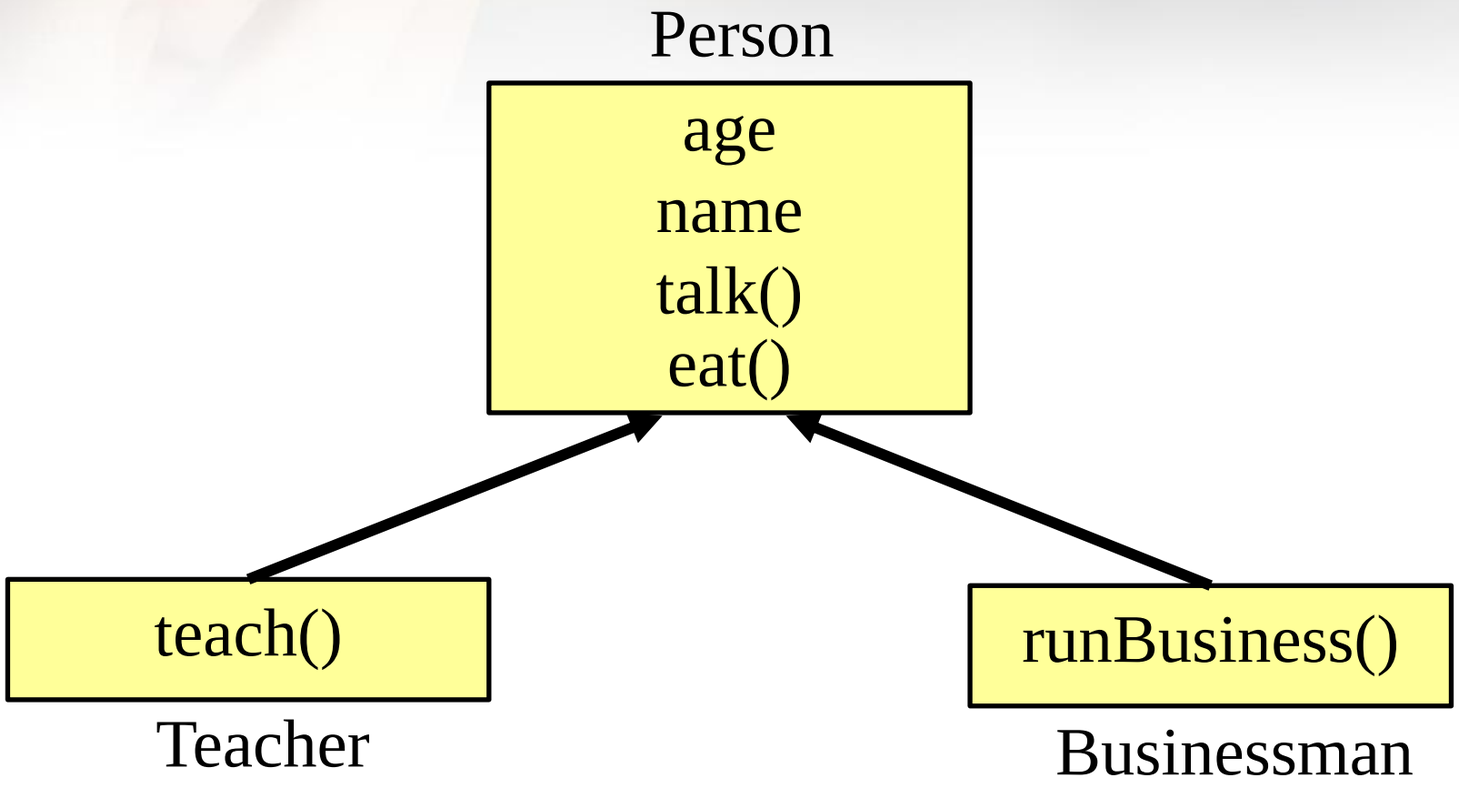
Result

```
fun main() {
    val maria = Girl()
    maria.actualAge = 15
    maria.age = 15
    println("Maria: actual age = ${maria.actualAge}")
    println("Maria: fake age = ${maria.age}")
    val angela = Girl()
    angela.actualAge = 35
    angela.age = 35
    println("Angela: actual age = ${angela.actualAge}")
    println("Angela: fake age  = ${angela.age}")
}
```

Inheritance

- Inheritance is one of the key features of object-oriented programming.
- It allows the user to create a new class (child class) from an existing class (base class).
- The child class inherits all the features from the base class and can have additional features of its own.
- Keyword: open and override

Inheritance



Kotlin inheritance

```
open class Person(age: Int, name: String) {
    init {
        println("My name is $name.")
        println("My age is $age")
    }
}
class Teacher(age: Int, name: String): Person(age, name) {
    fun teach () {
        println("I teach in primary school.")
    } }
class Businessman(age: Int, name: String): Person(age, name) {
    fun runBusiness() {
        println(“I have my company.")
    } }
```

```
fun main() {
    val t1 = Teacher(25, "Jack")
    t1.teach ()
    println()
    val f1 = Businessman (29, "Christiano")
    f1. runBusiness()
}
```

Result

Overriding Member Function

```
open class Person() {
    open fun displayAge(age: Int) {
        println("My age is $age.")
    }
}
class Girl: Person() {
    override fun displayAge(age: Int) {
        println("My fake age is ${age - 5}.")
    }
}
fun main() {
    val girl = Girl()
    girl.displayAge(31)
}
```

Result

Overriding Properties

```
open class Person() {
    open var age: Int = 0
        get() = field
        set(value) {
            field = value
        }
}
class Girl: Person() {
    override var age: Int = 0
        get() = field
        set(value) {
            field = value - 5
        }
}
```

```
fun main() {
    val girl = Girl()
    girl.age = 31
    println("My fake age is ${girl.age}.")
}
```

Result

Calling Members of Base Class from Child Class

```
open class Person() {
    open fun displayAge(age: Int) {
            println("My actual age is $age.")
    }
}
class Girl: Person() {
    override fun displayAge(age: Int) {
            // calling function of base class
             super.displayAge(age)
             println("My fake age is ${age - 5}.")
    }
}
```

```
fun main() {
    val girl = Girl()
    girl.displayAge(31)
}
```

Result

Kotlin Visibility Modifiers

- private - visible (can be accessed) from inside the class only.
- public - visible everywhere. (default)
- protected - visible to the class and its subclass.
- internal - any client inside the module can access them.

Kotlin Visibility Modifiers

Example

```
open class Base() {
    var a = 1             // public by default
   private var b = 2      // private to Base class
   protected open val c = 3 // visible to the Base and the Child class
   internal val d = 4     // visible inside the same module

    protected fun e() { }     // visible to the Base and the Child class
}
class Derived: Base() {
   // a, c, d, and e() of the Base class are visible
   // b is not visible
   override val c = 9      // c is protected
}
```

Kotlin Visibility Modifiers

Example (cont.)

```
fun main() {
    val base = Base()

    // base.a and base.d are visible
    // base.b, base.c and base.e() are not visible

    val derived = Derived()
    // derived.c is not visible
}
```

Kotlin Abstract Class

An abstract class cannot create objects.

However, can inherit subclasses from it.

```
abstract class Person {
    var age: Int = 40

    fun displayID(id: Int) {
        println("My ID is $id.")
    }
    abstract fun displayJob(description: String)
}
```

** displayJob() doesn't have any implementation and must be overridden in its subclasses.

Kotlin Abstract Class

```
class Teacher(name: String): Person(name) {
     val myName = name
     override fun displayJob(description: String) {
        println(description)
    }
}
fun main() {
    val jack = Teacher("Jack Smith")
    println("My name is ${jack.myName}.")
    jack.displayJob("I'm a mathematics teacher.")
    jack.displayID (1234567890)
}
```

Result

Kotlin Interfaces

- Interfaces can contain declarations of abstract methods and method implementations.
- To different from abstract classes is that interfaces cannot store state.
- A property declared in an interface can either be abstract, or it can provide implementations for accessors.
- Properties declared in interfaces can't have backing fields, and therefore accessors declared in interfaces can't reference them

Kotlin Interfaces

```
interface MyInterface {
    var prop: Int // abstract
    val propertyImplementation: String
        get() = "Implementation property "
    fun funInterface() {
        println(prop)
        println(propertyImplementation)
    }
}
class Child : MyInterface {
    override var prop: Int = 25
}
```

```
fun main() {
    var a = Child()
    a.prop = 30
    //a.propertyImplementation = "Test"  /// Error
    println(a.prop)
    println(a.propertyImplementation)
    a.funInterface()
}
```

Result

Kotlin Nested and Inner Class

Define a class within another class known as nested class.

```
class Outer {
    val a = "Outside Nested class."
    class Nested {
        val b = "Inside Nested class."
        fun callMe() = "Function call from inside Nested class."
    } }
fun main() {
    // accessing member of Nested class
    println(Outer.Nested().b)
    // creating object of Nested class
    val nested = Outer.Nested()
    println(nested.callMe())
}
```

Result

Kotlin Nested and Inner Class

- Inner Class

```
class Outer {
    val a = "Outside Nested class."
    inner class Inner {
        fun callMe() = a
    }
}
fun main() {
    val outer = Outer()
    println("Using outer object: ${outer.Inner().callMe()}")
    val inner = Outer().Inner()
    println("Using inner object: ${inner.callMe()}")
}
```

Result

Kotlin Data Class

- Create a class solely to hold data.
- For this class, the compiler automatically generates:
  - copy() function, equals() and hashCode() pair, and toString() form of the primary constructor
  - componentN() functions (N: 1, 2, 3,…)
- The requirements:
  - The primary constructor must have at least one parameter.
  - The parameters of the primary constructor must be marked as either val (read-only) or var (read-write).
  - The class cannot be open, abstract, inner.

Kotlin Data Class

Example

```
data class User(val name: String, val age: Int)

fun main() {
val u1 = User("John", 29)
    println("call toString() :  ${u1.toString()}")
    println("call component1() :  ${u1.component1()}")
    println("call component2() :  ${u1.component2()}")
    println("call hashCode() :  ${u1.hashCode()}")

}
```

Result

Enum Class

Enums are special classes which limit the possible values of an object for that class. The possible values defined for that class are final or unchangeable.

```
enum class Country {
  Japan,  Korea, China, Thailand   }

fun main() {
    val myCountry = Country.Thailand
    println("I am from $myCountry")
}
```

Result

Enum Class

Add properties for each enum value by adding a constructor to the enum class that defines the properties

```
enum class Language(var code: String) {
   JAPANESE("jp"),
   KOREAN("kr"),
   CHINESE(“cn")
   THAI("th")
 }
 fun main() {

 val thaiCode = Language.THAI.code
       println("Show Language code $thaiCode")
}
```

Result

Kotlin Object

Singleton is an object-oriented pattern where a class can have only one instance (object)

```
object Singleton_object {
    private var a: Int = 2
    var b: Int = 3 + a
    fun printObject(): String {
        return "Call function in Singleton object"
    }
}
fun main() {
      //  println("a = ${Singleton_object.a}") ///Error
    println("b = ${Singleton_object.b}")
    val result = Singleton_object.printObject()
    println("result = $result")
}
```

Result

Kotlin Companion Objects

Call method in class by using the class name

```
class Person {
  companion object {
        fun callMe() = println("I'm called from companion object .")
    }
}

fun main() {
    Person.callMe()
}
```

Result

Extension Function

- An extension function extends a class with new functionality.
- An extension function is a member function of a class that is defined outside the class.

Example: Remove First and Last Character of String

```
fun String.removeFirstLastChar(): String =  this.substring(1, this.length - 1)

fun main() {
    val myString= "Hello Everyone"
    val result = myString.removeFirstLastChar()
    println("Result of Hello Everyone: $result")
}
```

Result

```
Result of Hello Everyone: ello Everyon
```

Extension function

- The extension function removeFirstLastChar() is added to the *String* class.
- The class name is the receiver type (*String* class in example). The *this* keyword inside the extension function refers the receiver object.

```
fun String.removeFirstLastChar(): String =  this.substring(1, this.length - 1)
```

receiver type

receiver object

Lambda

- Anonymous functions are 'function literals', i.e. functions that are not declared but passed immediately as an expression.

```
val functionName: (parameter_data_type) -> return_data_type  = {
                    // it is parameter
 }
```

```
fun main() {

   val printAndTriple : (Int) -> Int = {
        it * 3
    }
 println("Call printAndTriple(3) = " + printAndTriple(3))
}
```

Result

```
Call printAndTriple(3) = 9
```

Lambda

- Multi-Parameter

```
fun main() {

    val concat : (String, String) -> String = {
       s1,s2 -> s1+s2
    }

    println("Call concat = " + concat("abc","def"))
```

Result

End Of Chapter

References

- http://www.cems.uwe.ac.uk/~bk2dean/uwe/digitalmedia/mobiledevelopment/lectures/anatomy_of_a_mobile_device.ppt
- https://en.wikipedia.org/wiki/Windows_Phone#/media/File:Windows_10_Logo.svg
- https://www.cs.cmu.edu/~bam/uicourse/830spring09/BFeiginMobileApplicationDevelopment.pdf
- http://cs.joensuu.fi/~zhao/Courses/Location/Dariusz.ppt
- https://www.programiz.com/kotlin-programming/class-objects