Classes and Objects

Classes and Inheritance

Classes

Basic Definition

```
class Invoice { /*...*/ }
```

```
class Person(
    val firstName: String,
    val lastName: String,
    var age: Int, // trailing comma
) { /*...*/ }
```

If class is empty

```
class Empty
```

Use Constructors

```
class Person constructor(firstName: String) { /*...*/ }
```

Classes(Kotlin vs Python)

```python
class Lunit:
    def init(self, name, age, is_married):
        self.name = name
        self.age = age
        self.is_married = is_married

        print(f"Hello, {name}!")
```

Definition Class
in Python

```kotlin
class Lunit (var name: String, var age: Int, var isMarried: Boolean) {
    init {
        println("Hello, $name!")
    }
}
```

Definition Class
in Kotlin

Classes

Multiple Constructor

```
class Sample constructor(val name: String) {
    constructor(name: String, age: Int): this(name)
    constructor(name: String, age: Int, birthday: String): this(name, age)
}
```

```
class Sample {
    constructor(name: String) {
        println("name: $name")
    }
    constructor(name: String, age: Int): this(name) {
        println("name: $name, age: $age")
    }
    constructor(name: String, age: Int, birthday: String): this(name, age) {
        println("name: $name, age: $age, birthday: $birthday")
    }
}

fun main() {
    var sample = Sample( name: "kotlin",  age: 27,  birthday: "2020-12-01")
}
```

```
name: kotlin
name: kotlin, age: 27
name: kotlin, age: 27, birthday: 2020-12-01
```

Classes(Kotlin vs Python)

```python
class Sample:
    def init(self, name, age=0, birthday=""):
        if age == 0 and birthday == "":
            constructor_a(name)
        elif birthday == "":
            constructor_b(name, age)
        else:
            constructor_c(name, age, birthday)

    def constructor_a(self, name):
        self.name = name
        print(f"name: {name}")

    def constructor_b(self, name, age):
        self.constructor_a(name)
        self.age = age
        print(f"name: {name}, age: {age}")

    def constructor_c(self, name, age, birthday):
        self.constructor_b(name, age)
        self.birthday = birthday
        print(f"name: {name}, age: {age}, birthday: {birthday}")
```

Multiple Constructor in Python

```kotlin
class Sample {
    constructor(name: String) {
        println("name: $name")
    }
    constructor(name: String, age: Int): this(name) {
        println("name: $name, age: $age")
    }
    constructor(name: String, age: Int, birthday: String): this(name, age) {
        println("name: $name, age: $age, birthday: $birthday")
    }
}

fun main() {
    var sample = Sample( name: "kotlin",  age: 27,  birthday: "2020-12-01")
}
```

Multiple Constructor in Kotlin

Classes

Secondary Constructor

```
class Person {
    var children: MutableList<Person> = mutableListOf()
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

```
class Constructors {
    init {
        println("Init block")
    }

    constructor(i: Int) {
        println("Constructor")
    }
}
```

Non-automatic Declared-constructor

```
class DontCreateMe private constructor () { /*...*/ }
```

Inheritance

Basic Inheritance

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

```
class MyView : View {
    constructor(ctx: Context) : super(ctx)

    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, attrs)
}
```

Class Overriding : method

```
open class Shape {
    open fun draw() { /*...*/ }
    fun fill() { /*...*/ }
}

class Circle() : Shape() {
    override fun draw() { /*...*/ }
}
```

```
open class Rectangle() : Shape() {
    final override fun draw() { /*...*/ }
}
```

Inheritance

Class Overriding : property

```
interface Shape {
    val vertexCount: Int
}

class Rectangle(override val vertexCount: Int = 4) : Shape // Always has 4 vertices

class Polygon : Shape {
    override var vertexCount: Int = 0  // Can be set to any number later
}
```

Class Inheritance Flow

```
open class Base(val name: String) {

    init { println("Initializing Base") }

    open val size: Int =
        name.length.also { println("Initializing size in Base: $it") }
}

class Derived(
    name: String,
    val lastName: String,
) : Base(name.capitalize().also { println("Argument for Base: $it") }) {

    init { println("Initializing Derived") }

    override val size: Int =
        (super.size + lastName.length).also { println("Initializing size in Derived: $it") }
}
```

Inheritance

Call Super Class

```
open class Rectangle {
    open fun draw() { println("Drawing a rectangle") }
    val borderColor: String get() = "black"
}

class FilledRectangle : Rectangle() {
    override fun draw() {
        super.draw()
        println("Filling the rectangle")
    }

    val fillColor: String get() = super.borderColor
}
```

```
class FilledRectangle: Rectangle() {
    override fun draw() {
        val filler = Filler()
        filler.drawAndFill()
    }

    inner class Filler {
        fun fill() { println("Filling") }
        fun drawAndFill() {
            super@FilledRectangle.draw() // Calls Rectangle's implementation of draw()
            fill()
            println("Drawn a filled rectangle with color ${super@FilledRectangle.borderColor}")
        }
    }
}
```

Inheritance

Overriding Multiple Class

```
open class Rectangle {
    open fun draw() { /* ... */ }
}

interface Polygon {
    fun draw() { /* ... */ } // interface members are 'open' by default
}

class Square() : Rectangle(), Polygon {
    // The compiler requires draw() to be overridden:
    override fun draw() {
        super<Rectangle>.draw() // call to Rectangle.draw()
        super<Polygon>.draw() // call to Polygon.draw()
    }
}
```

Abstract Class

```
open class Polygon {
    open fun draw() {}
}

abstract class Rectangle : Polygon() {
    abstract override fun draw()
}
```

Properties and Fields

Properties

Declare

```
class Address {
    var name: String = "Holmes, Sherlock"
    var street: String = "Baker"
    var city: String = "London"
    var state: String? = null
    var zip: String = "123456"
}
```

```
var <propertyName>[: <PropertyType>] [= <property_initializer>]
    [<getter>]
    [<setter>]
```

Getters and Setters

```
val isEmpty: Boolean
    get() = this.size == 0
```

```
var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value)
    }
```

Properties

Backing Fields

```
var counter = 0 // Note: the initializer assigns the backing field directly
    set(value) {
        if (value >= 0) field = value
    }
```

Note that "Field" is automatically declared.

Backing Properties

```
private var _table: Map<String, Int>? = null
public val table: Map<String, Int>
    get() {
        if (_table == null) {
            _table = HashMap() // Type parameters are inferred
        }
        return _table ?: throw AssertionError("Set to null by another thread")
    }
```

Properties

Compile-Time Constants

```
const val SUBSYSTEM_DEPRECATED: String = "This subsystem is deprecated"

@Deprecated(SUBSYSTEM_DEPRECATED) fun foo() { ... }
```

For use compile-time constants :

1. Top-level or member of an object or a companion object
2. Initialized with String or primitive type value
3. Cannot use custom getter

Late-Initialized Properties and Variables

```
public class MyTest {
    lateinit var subject: TestSubject

    @SetUp fun setup() {
        subject = TestSubject()
    }

    @Test fun test() {
        subject.method()  // dereference directly
    }
}
```

Interfaces

Interfaces

Basic Interfaces

```
interface MyInterface {
    fun bar()
    fun foo() {
      // optional body
    }
}
```

Interfaces Inheritance
with Properties

```
interface Named {
    val name: String
}

interface Person : Named {
    val firstName: String
    val lastName: String

    override val name: String get() = "$firstName $lastName"
}

data class Employee(
    // implementing 'name' is not required
    override val firstName: String,
    override val lastName: String,
    val position: Position
) : Person
```

Interfaces

Resolve overriding conflicts

```
interface A {
    fun foo() { print("A") }
    fun bar()
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() { print("bar") }
}

class D : A, B {
    override fun foo() {
        super<A>.foo()
        super<B>.foo()
    }

    override fun bar() {
        super<B>.bar()
    }
}
```

Functional Interfaces

SAM : Single Abstract Method

Declare

```
fun interface KRunnable {
   fun invoke()
}
```

with lambda

```
fun interface IntPredicate {
   fun accept(i: Int): Boolean
}

val isEven = IntPredicate { it % 2 == 0 }

fun main() {
   println("Is 7 even? - ${isEven.accept(7)}")
}
```

Visibility Modifiers

Visibility Modifiers

There are four visibility modifiers, private, protected, internal and public.

objects level	private	protected	internal	public(basic)
in package	in same file	-	in same module	all objects
in class and interface	in same class	in same class and subclass	in same module	all objects
in constructors	in same class	-	-	all objects
in local	-	-	-	-
in modules	-	-	in same module	

Extensions

Extensions

Declare

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {
    val tmp = this[index1] // 'this' corresponds to the list
    this[index1] = this[index2]
    this[index2] = tmp
}
```

```
val list = mutableListOf(1, 2, 3)
list.swap(0, 2) // 'this' inside 'swap()' will hold the value of 'list'
```

extensions resolved
static

```
open class Shape

class Rectangle: Shape()

fun Shape.getName() = "Shape"

fun Rectangle.getName() = "Rectangle"

fun printClassName(s: Shape) {
    println(s.getName())
}

printClassName(Rectangle())
```

Extensions

Nullable Receiver

```
fun Any?.toString(): String {
    if (this == null) return "null"
    // after the null check, 'this' is autocast to a non-null type, so the toString() below
    // resolves to the member function of the Any class
    return toString()
}
```

Extension Property

```
val <T> List<T>.lastIndex: Int
    get() = size - 1
```

Extension Companion

```
class MyClass {
    companion object { }  // will be called "Companion"
}

fun MyClass.Companion.printCompanion() { println("companion") }

fun main() {
    MyClass.printCompanion()
}
```

Extensions

Scope of Extension

```
package org.example.declarations

fun List<String>.getLongestString() { /*...*/}
```

```
package org.example.usage

import org.example.declarations.getLongestString

fun main() {
    val list = listOf("red", "green", "blue")
    list.getLongestString()
}
```

Extensions

Extension as Members

```
class Host(val hostname: String) {
    fun printHostname() { print(hostname) }
}

class Connection(val host: Host, val port: Int) {
    fun printPort() { print(port) }

    fun Host.printConnectionString() {
        printHostname()
        print(":")
        printPort()
    }

    fun connect() {
        host.printConnectionString()
    }
}

fun main() {
    Connection(Host( hostname: "kotl.in"),  port: 443).connect()
}
```

Data Classes

Data Classes

Declare

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

Point :

1. Class always have equals(), hashCode(), toString(), copy()
2. When inheritance basic functions, basic functions must muted.
3. For inheritance componentN(), function must be open and return compatible types.
4. Explicit implement of functions(componentN, copy) is not allowed.

Caution :

1. At least one primary constructor with at least one parameter
2. All parameter in constructor needs var or val
3. Class cannot be abstract, sealed or inner
4. (before 1.1) Class may only implement interfaces

Data Classes

Properties

can use already declared

compare

```
data class Person(val name: String) {
    var age: Int = 0
}
```

```
val person1 = Person("John")
val person2 = Person("John")
person1.age = 10
person2.age = 20
```

```
person1 == person2: true
person1 with age 10: Person(name=John)
person2 with age 20: Person(name=John)
```

Copy

```
fun copy(name: String = this.name, age: Int = this.age) = User(name, age)
```

```
val jack = User(name = "Jack", age = 1)
val olderJack = jack.copy(age = 2)
```

Sealed Classes

Sealed Classes

Declare

```
sealed class Expr
data class Const(val number: Double) : Expr()
data class Sum(val e1: Expr, val e2: Expr) : Expr()
object NotANumber : Expr()
```

Point :

1. Class can have one of the types but cannot have others.
2. Run like extension of enum classes.
3. Subclass of sealed class can have multiple instance.

Caution :

1. Class is abstract class.
2. All constructor of class must be private.
3. Inheritance class of sealed class can use anywhere.
4. Sealed classes and subclasses will be located same file.

Sealed Classes

```
fun main() {
    var color = "red"
    var font = when(color) {
        "red" -> { "apple" }
        "green" -> { "glass" }
        "blue" -> { "water" }
        else -> { "???" }
    }

    println(font)
}
```

Data Checking
with String

```
sealed class Color() {
    object Red: Color()
    object Green: Color()
    object Blue: Color()
}

fun main() {
    var color : Color =  Color.Red
    var font = when(color) {
        Color.Red -> { "apple" }
        Color.Green -> { "glass" }
        Color.Blue -> { "water" }
    }

    println(font)
}
```

Data Checking with
Sealed Class

Generics

Generics

Declare

```
class Box<T>(t: T) {
    var value = t
}
```

```
val box: Box<Int> = Box<Int>(1)
```

when inferred types

```
val box = Box(1)
```

Variance

Point :

In Java, generic types are **invariant**. So, List<String> **is not a subtype of** List<Object>.

To solve this problem, Java uses **? wildcard**. This is not complete solution, however.

When use ?, read done correct, but cannot write cause unknown subtype.

Declare

Declaration-Site Variance

```
interface Source<out T> {
    fun nextT(): T
}

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs
    // ...
}
```

Use-Site Variance

```
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0)
    val y: Comparable<Double> = x
}
```

Type Projections

How to use

```
class Array<T>(val size: Int) {
    fun get(index: Int): T { ... }
    fun set(index: Int, value: T) { ... }
}
```

```
val ints: Array<Int> = arrayOf(1, 2, 3)
val any = Array<Any>(3) { "" }
copy(ints, any)
```

wrong

```
fun copy(from: Array<Any>, to: Array<Any>) {
    assert(from.size == to.size)
    for (i in from.indices)
        to[i] = from[i]
}
```

correct

```
fun copy(from: Array<out Any>, to: Array<Any>) { ... }
```

Generic Functions

Declare

```
fun <T> singletonList(item: T): List<T> {
    // ...
}
```

how to use

```
val l = singletonList<Int>(1)
```

Generic Constraints

```
fun <T : Comparable<T>> sort(list: List<T>) {  ... }
```

upper bound

Nested and Inner Classes

Nested and Inner Classes

Nested Classes

```
class Outer {
    private val bar: Int = 1
    class Nested {
        fun foo() = 2
    }
}

val demo = Outer.Nested().foo()
```

Nested Interfaces

```
interface OuterInterface {
    class InnerClass
    interface InnerInterface
}

class OuterClass {
    class InnerClass
    interface InnerInterface
}
```

Inner Classes

```
class Outer {
    private val bar: Int = 1
    inner class Inner {
        fun foo() = bar
    }
}

val demo = Outer().Inner().foo()
```

Enum Classes

Enum Classes

Declare

```
enum class Color(val rgb: Int) {
        RED(0xFF0000),
        GREEN(0x00FF00),
        BLUE(0x0000FF)
}
```

implement interfaces

```
enum class IntArithmetics : BinaryOperator<Int>, IntBinaryOperator {
    PLUS {
        override fun apply(t: Int, u: Int): Int = t + u
    },
    TIMES {
        override fun apply(t: Int, u: Int): Int = t * u
    };

    override fun applyAsInt(t: Int, u: Int) = apply(t, u)
}
```

use values in enum

```
enum class RGB { RED, GREEN, BLUE }

inline fun <reified T : Enum<T>> printAllValues() {
    print(enumValues<T>().joinToString { it.name })
}

printAllValues<RGB>() // prints RED, GREEN, BLUE
```

Objects

Objects

Declare

```
object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) {
        // ...
    }

    val allDataProviders: Collection<DataProvider>
        get() = // ...
}
```

inheritance

```
object DefaultListener : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { ... }

    override fun mouseEntered(e: MouseEvent) { ... }
}
```

Companion

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}
```

```
class MyClass {
    companion object { }
}

val x = MyClass.Companion
```

Objects

Expression

```
window.addMouseListener(object : MouseAdapter() {
    override fun mouseClicked(e: MouseEvent) { /*...*/ }

    override fun mouseEntered(e: MouseEvent) { /*...*/ }
})
```

just use object

```
fun foo() {
    val adHoc = object {
        var x: Int = 0
        var y: Int = 0
    }
    print(adHoc.x + adHoc.y)
}
```

Caution

```
class C {
    // Private function, so the return type is the anonymous object type
    private fun foo() = object {
        val x: String = "x"
    }

    // Public function, so the return type is Any
    fun publicFoo() = object {
        val x: String = "x"
    }

    fun bar() {
        val x1 = foo().x        // Works
        val x2 = publicFoo().x  // ERROR: Unresolved reference 'x'
    }
}
```

Aliases and Inline Classes

Aliases and Inline Classes

Declare
can type alias in class, function

```
class A {
    inner class Inner
}
class B {
    inner class Inner
}

typealias AInner = A.Inner
typealias BInner = B.Inner
```

Inline Classes
cannot have init, backing fields

```
inline class Name(val s: String) {
    val length: Int
        get() = s.length

    fun greet() {
        println("Hello, $s")
    }
}
```

Caution :
Aliases support assignment-compatible, but inline classes are not.

Delegation

Delegation

Declare
inheritance example

```
interface Base {
    fun print()
}

class BaseImpl(val x: Int) : Base {
    override fun print() { print(x) }
}

class Derived(b: Base) : Base by b

fun main() {
    val b = BaseImpl(10)
    Derived(b).print()
}
```

more details

```
interface Base {
    fun printMessage()
    fun printMessageLine()
}

class BaseImpl(val x: Int) : Base {
    override fun printMessage() { print(x) }
    override fun printMessageLine() { println(x) }
}

class Derived(b: Base) : Base by b {
    override fun printMessage() { print("abc") }
}

fun main() {
    val b = BaseImpl(10)
    Derived(b).printMessage()
    Derived(b).printMessageLine()
}
```

Delegate Properties

Delegate to Another Property
can delegate getter and setter

```
class MyClass {
    var newName: Int = 0
    @Deprecated("Use 'newName' instead", ReplaceWith("newName"))
    var oldName: Int by this::newName
}

fun main() {
    val myClass = MyClass()
    // Notification: 'oldName: Int' is deprecated.
    // Use 'newName' instead
    myClass.oldName = 42
    println(myClass.newName) // 42
}
```

Caution :
Only delegate top-level property, properties in same class, properties in another classes

examples

```
class User(val map: Map<String, Any?>) {
    val name: String by map
    val age: Int     by map
}

fun main() {
    val user = User(mapOf(
        "name" to "John Doe",
        "age"  to 25
    ))

    println(user.name)
    println(user.age)
}
```

Delegate Properties

Requirements

val(getValue) :

thisRef : must be the same or a supertype of the property owner.

property : must be of type KProperty<*> or its supertype.

```
class Resource

class Owner {
    val valResource: Resource by ResourceDelegate()
}

class ResourceDelegate {
    operator fun getValue(thisRef: Owner, property: KProperty<*>): Resource {
        return Resource()
    }
}
```

Delegate Properties

Requirements

var(setValue) :

thisRef : must be the same or a supertype of the property owner.

property : must be of type KProperty<*> or its supertype.

value: must be of the same as the property.

```
class Resource

class Owner {
    var varResource: Resource by ResourceDelegate()
}

class ResourceDelegate(private var resource: Resource = Resource()) {
    operator fun getValue(thisRef: Owner, property: KProperty<*>): Resource {
        return resource
    }
    operator fun setValue(thisRef: Owner, property: KProperty<*>, value: Any?) {
        if (value is Resource) {
            resource = value
        }
    }
}
```

Thank you!