CODE SPITZ 82

KOTLIN ELEMENTARY

1 2 3 [✓ 4] 5 6

constructor

basic

```
open public class ClassTest0{
    private val propA:String
    private val propB:String
    public constructor(a:String, b:String){
        println("constructor1")
        propA = a
        propB = b
    }
    public constructor(a:String):this(a, "b"){
        println("constructor2")
    }
}
```

basic

```
open public class ClassTest0{
    private val propA:String
    private val propB:String
    public constructor(a:String, b:String){
        println("constructor1")
        propA = a
        propB = b
    }
    public constructor(a:String):this(a, "b"){
        println("constructor2")
    }
}
```

```
ClassTest0("a", "b")
ClassTest0("a")
```

basic

```
open public class ClassTest0{
    private val propA:String
    private val propB:String
    public constructor(a:String, b:String){
        println("constructor1")
        propA = a
        propB = b
    }
    public constructor(a:String):this(a, "b"){
        println("constructor2")
    }
}
```

```
ClassTest0("a", "b")
ClassTest0("a")
```

basic

```
open public class ClassTest0{
    private val propA:String
    private val propB:String
    public constructor(a:String, b:String){
        println("constructor1")
        propA = a
        propB = b
    }
    public constructor(a:String):this(a, "b"){
        println("constructor2")
    }
}
```

```
ClassTest0("a", "b")
ClassTest0("a")
```

basic

```kotlin
class ClassTest0{
    private val propA:String
    private val propB:String
    constructor(a:String, b:String){
        println("constructor1")
        propA = a
        propB = b
    }
    constructor(a:String):this(a, "b"){
        println("constructor2")
    }
}
```

```kotlin
ClassTest0("a", "b")
ClassTest0("a")
```

basic

```
class ClassTest0{
    private val propA:String
    private val propB:String
    constructor(a:String, b:String){
        println("constructor1")
        propA = a
        propB = b
    }
    constructor(a:String):this(a, "b"){
        println("constructor2")
    }
}
```

```
ClassTest0("a", "b")
ClassTest0("a")
```

basic

```
class ClassTest0 constructor(a:String, b:String){
    private val propA:String
    private val propB:String
    init{
        println("constructor1")
        propA = a
        propB = b
    }
    constructor(a:String):this(a, "b"){
        println("constructor2")
    }
}
```

```
ClassTest0("a", "b")
ClassTest0("a")
```

basic

```
class ClassTest0(a:String, b:String){
    private val propA:String
    private val propB:String
    init{
        println("constructor1")
        propA = a
        propB = b
    }
    constructor(a:String):this(a, "b"){
        println("constructor2")
    }
}
```

```
ClassTest0("a", "b")
ClassTest0("a")
```

basic

```
class ClassTest0(a:String, b:String){
    private val propA:String
    private val propB:String
    init{
        println("constructor1")
        propA = a
        propB = b
    }
    constructor(a:String):this(a, "b"){
        println("constructor2")
    }
}
```

```
ClassTest0("a", "b")
ClassTest0("a")
```

basic

```
class ClassTest0(private val propA:String, private val propB:String){
    init{
        println("constructor1")
    }
    constructor(a:String):this(a, "b"){
        println("constructor2")
    }
}
```

basic

```
class ClassTest0(private val propA:String, private val propB:String){
    init{
        println("constructor1")
    }
    constructor(a:String):this(a, "b"){
        println("constructor2")
    }
}
```

basic

```
class ClassTest0(private val propA:String, private val propB:String){
    constructor(a:String):this(a, "b")
}
```

super

```
open class ClassTest0(private val propA:String, private val propB:String){
    constructor(a:String):this(a, "b")
}
```

super

```
open class ClassTest0(private val propA:String, private val propB:String){
    constructor(a:String):this(a, "b")
}
```

```
class ClassTest1:ClassTest0("a"){
    private val propC = "c"
}
```

super

```kotlin
open class ClassTest0(private val propA:String, private val propB:String){
    constructor(a:String):this(a, "b")
}
```

```kotlin
class ClassTest1:ClassTest0("a", "b"){
    private val propC = "c"
}
```

super

```
open class ClassTest0(private val propA:String, private val propB:String){
    constructor(a:String):this(a, "b")
}
```

```
class ClassTest1(a:String, b:String, c:String):ClassTest0(a, b){
    private val propC = c
}
```

super

```
open class ClassTest0(private val propA:String, private val propB:String){
    constructor(a:String):this(a, "b")
}
```

```
class ClassTest1(a:String, b:String, private val propC:String):ClassTest0(a, b)
```

operator overloading

simple map

```kotlin
class Map{
    private val map = mutableMapOf<String, String>()
    operator fun get(key:String) = map[key]
    operator fun set(key:String, value:String){map[key] = value}
}
```

simple map

```kotlin
class Map{
    private val map = mutableMapOf<String, String>()
    operator fun get(key:String) = map[key]
    operator fun set(key:String, value:String){map[key] = value}
}
```

```kotlin
val m = Map()
m["test"] = "123"
println(m["test"])
```

Supported operator

Expression	Translated to
+a	a.unaryPlus()
-a	a.unaryMinus()
!a	a.not()
a++	a.inc() + see below
a--	a.dec() + see below
a + b	a.plus(b)
a - b	a.minus(b)
a * b	a.times(b)
a / b	a.div(b)
a % b	a.rem(b), a.mod(b) (deprecated)
a..b	a.rangeTo(b)
a in b	b.contains(a)
a !in b	!b.contains(a)

Supported operator

Expression	Translated to
a += b	a.plusAssign(b)
a -= b	a.minusAssign(b)
a *= b	a.timesAssign(b)
a /= b	a.divAssign(b)
a == b	a?.equals(b) ?: (b === null)
a != b	!(a?.equals(b) ?: (b === null))
a > b	a.compareTo(b) > 0
a < b	a.compareTo(b) < 0
a >= b	a.compareTo(b) >= 0
a <= b	a.compareTo(b) <= 0

Supported operator

Expression	Translated to
a[i]	a.get(i)
a[i, j]	a.get(i, j)
a[i_1, ..., i_n]	a.get(i_1, ..., i_n)
a[i] = b	a.set(i, b)
a[i, j] = b	a.set(i, j, b)
a[i_1, ..., i_n] = b	a.set(i_1, ..., i_n, b)
a()	a.invoke()
a(i)	a.invoke(i)
a(i, j)	a.invoke(i, j)
a(i_1, ..., i_n)	a.invoke(i_1, ..., i_n)

getter, setter

simple map

```kotlin
class Map{
    private val map = mutableMapOf<String, String>()
    operator fun get(key:String) = map[key]
    operator fun set(key:String, value:String){map[key] = value}
    val name:String? get() = map["name"]
    var job:String? get() = map["job"]
                    set(value){value?.let{map["job"] = it}}
}
```

simple map

```
class Map{
    private val map = mutableMapOf<String, String>()
    operator fun get(key:String) = map[key]
    operator fun set(key:String, value:String){map[key] = value}
    val name:String? get() = map["name"]
    var job:String? get() = map["job"]
                    set(value){value?.let{map["job"] = it}}
}
```

```
val m = Map()
m["name"] = "hika"
println(m.name)
m.job = "developer"
println(m.job)
```

by, by lazy

simple map

```
class Map{
    private val map = mutableMapOf<String, String>()
    operator fun get(key:String) = map[key]
    operator fun set(key:String, value:String){map[key] = value}
    val name by lazy{map["firstName"] + " " +  map["lastName"]}
}
```

lazy

```kotlin
public actual fun <T> lazy(initializer: () -> T): Lazy<T> = UnsafeLazyImpl(initializer)
public actual fun <T> lazy(mode: LazyThreadSafetyMode, initializer: () -> T): Lazy<T> = UnsafeLazyImpl(initializer)
public actual fun <T> lazy(lock: Any?, initializer: () -> T): Lazy<T> = UnsafeLazyImpl(initializer)
```

```kotlin
internal class UnsafeLazyImpl<out T>(initializer: () -> T) : Lazy<T>, Serializable {
    private var initializer: (() -> T)? = initializer
    private var _value: Any? = UNINITIALIZED_VALUE

    override val value: T
        get() {
            if (_value === UNINITIALIZED_VALUE) {
                _value = initializer!!()
                initializer = null
            }
            @Suppress("UNCHECKED_CAST")
            return _value as T
        }
```

lazy

```
public actual fun <T> lazy(initializer: () -> T): Lazy<T> = UnsafeLazyImpl(initializer)
public actual fun <T> lazy(mode: LazyThreadSafetyMode, initializer: () -> T): Lazy<T> = UnsafeLazyImpl(initializer)
public actual fun <T> lazy(lock: Any?, initializer: () -> T): Lazy<T> = UnsafeLazyImpl(initializer)
```

```
private class SynchronizedLazyImpl<out T>(initializer: () -> T, lock: Any? = null) : Lazy<T>, Serializable {
    private var initializer: (() -> T)? = initializer
    @Volatile private var _value: Any? = UNINITIALIZED_VALUE
    private val lock = lock ?: this
    override val value: T get() {
        val _v1 = _value
        if (_v1 !== UNINITIALIZED_VALUE) return _v1 as T
        return synchronized(lock) {
            val _v2 = _value
            if (_v2 !== UNINITIALIZED_VALUE) _v2 as T
            else {
                val typedValue = initializer!!()
                _value = typedValue
                initializer = null
                typedValue
            }
        }
    }
```

simple map

```kotlin
class Map{
    private val map = mutableMapOf<String, String>()
    operator fun get(key:String) = map[key]
    operator fun set(key:String, value:String){map[key] = value}
    val name by lazy{map["firstName"] + " " +  map["lastName"]}
}
```

```kotlin
val m = Map()
m["firstName"] = "hika"
m["lastName"] = "Maeng"
println(m.name)
```

object & companion object

static

```
class Parent{
  static void action(){}
}

Parent.action();
```

companion object

```
class Parent{
  static void action(){}
}

Parent.action();
```

```
class Parent{
    companion object{
        fun action(){}
    }
}
```

anonymous class

```
abstract class Parent{
}

Parent child1 = new Parent(){}
```

anonymous class

```
abstract class Parent{
}

Parent child1 = new Parent(){}
```

```
const Parent = class{}

const instance = new (class extends Parent{})()
```

anonymous class

```
abstract class Parent{
}

Parent child1 = new Parent(){}
```

```
const Parent = class{}

const instance = new (class extends Parent{})()
```

```
abstract class Parent
class ClassTest2{
    val Child1 = object:Parent(){}
}
```

anonymous class

```
abstract class Parent{
}

Parent child1 = new Parent(){}
```

```
const Parent = class{}

const instance = new (class extends Parent{})()
```

```
abstract class Parent
class ClassTest2{
    val Child1 = object:Parent(){}
}
object Child1:Parent(){}
```

singleton

```
class SingleTon{
    companion object{
        val INSTANCE = SingleTon()
    }
}
```

singleton

```
class SingleTon{
    companion object{
        val INSTANCE by lazy{SingleTon()}
    }
}
```

singleton

```
class SingleTon{
    companion object{
        val INSTANCE by lazy{SingleTon()}
    }
}
```

```
object Child1:Parent(){}
```

singleton

```
class SingleTon{
    companion object{
        val INSTANCE by lazy{SingleTon()}
    }
}
```

```
object Child1:Parent(){}
```

```
class Child1:Parent(){
    companion object{
        val INSTANCE by lazy{Child1()}
    }
}
```

sealed class & enum

enum

```
enum class Color(val code:String){
    Red("#f00"), Blue("#00f"), Green("#0f0")
}
```

enum

```
enum class Color(val code:String){
    Red("#f00"), Blue("#00f"), Green("#0f0")
}
```

```
abstract class Color(val code:String){
    object Red:Color("#f00")
    object Blue:Color("#00f")
    object Green:Color("#0f0")
}
```

enum

```
enum class Color(val code:String){
    Red("#f00"), Blue("#00f"), Green("#0f0")
}
```

```
abstract class Color(val code:String){
    object Red:Color("#f00")
    object Blue:Color("#00f")
    object Green:Color("#0f0")
}
```

```
object Yellow:Color("#ff0")
```

sealed class

```kotlin
enum class Color(val code:String){
    Red("#f00"), Blue("#00f"), Green("#0f0")
}
```

```kotlin
sealed class Color(val code:String){
    object Red:Color("#f00")
    object Blue:Color("#00f")
    object Green:Color("#0f0")
}
```

~~`object Yellow:Color("#ff0")`~~

sealed class

```kotlin
enum class Color(val code:String){
    Red("#f00"), Blue("#00f"), Green("#0f0")
}
```

```kotlin
sealed class Color(val code:String){
    object Red:Color("#f00")
    object Blue:Color("#00f")
    object Green:Color("#0f0")
    class Custom(code:String):Color(code)
}
```

```kotlin
val brown = Color.Custom("#cc865c")
```

html builder

El

```
abstract class El(val tagName:String){
    protected val el = when(tagName){
        "body" -> document.body ?: throw Throwable("no body")
        else -> document.createElement(tagName) as HTMLElement
    }
```

El

```
abstract class El(val tagName:String){
    protected val el = when(tagName){
        "body" -> document.body ?: throw Throwable("no body")
        else -> document.createElement(tagName) as HTMLElement
    }
    var html:String get() = el.innerHTML
                    set(value){el.innerHTML = value}
```

El

```kotlin
abstract class El(val tagName:String){
    protected val el = when(tagName){
        "body" -> document.body ?: throw Throwable("no body")
        else -> document.createElement(tagName) as HTMLElement
    }
    var html:String get() = el.innerHTML
                    set(value){el.innerHTML = value}
    operator fun get(key:String) = el.getAttribute(key) ?: ""
    operator fun set(key:String, value: Any) = el.setAttribute(key, "$value")
    operator fun invoke() = el
```

El

```
abstract class El(val tagName:String){
    protected val el = when(tagName){
        "body" -> document.body ?: throw Throwable("no body")
        else -> document.createElement(tagName) as HTMLElement
    }
    var html:String get() = el.innerHTML
                    set(value){el.innerHTML = value}
    operator fun get(key:String) = el.getAttribute(key) ?: ""
    operator fun set(key:String, value: Any) = el.setAttribute(key, "$value")
    operator fun invoke() = el
    operator fun plusAssign(child:El){el.appendChild(child.el)}
    operator fun minusAssign(child:El){el.removeChild(child.el)}
    val style:CSSStyleDeclaration get() = el.style
}
```

El

```kotlin
abstract class El(val tagName:String){
    protected val el = when(tagName){
        "body" -> document.body ?: throw Throwable("no body")
        else -> document.createElement(tagName) as HTMLElement
    }
    var html:String get() = el.innerHTML
                    set(value){el.innerHTML = value}
    operator fun get(key:String) = el.getAttribute(key) ?: ""
    operator fun set(key:String, value: Any) = el.setAttribute(key, "$value")
    operator fun invoke() = el
    operator fun plusAssign(child:El){el.appendChild(child())}
    operator fun minusAssign(child:El){el.removeChild(child())}
    val style:CSSStyleDeclaration get() = el.style
}
```

El

```
abstract class El(val tagName:String){
    protected val el = when(tagName){
        "body" -> document.body ?: throw Throwable("no body")
        else -> document.createElement(tagName) as HTMLElement
    }
    var html:String get() = el.innerHTML
                    set(value){el.innerHTML = value}
    operator fun get(key:String) = el.getAttribute(key) ?: ""
    operator fun set(key:String, value: Any) = el.setAttribute(key, "$value")
    operator fun invoke() = el
    operator fun plusAssign(child:El){el.appendChild(child.el)}
    operator fun minusAssign(child:El){el.removeChild(child.el)}
    val style:CSSStyleDeclaration get() = el.style
}
```

```
object Body:El("body")
class Div:El("div")
class Canvas:El("canvas"){
  val context:CanvasRenderingContext2D? get() =
  (el as? HTMLCanvasElement)?.getContext("2d") as? CanvasRenderingContext2D
}
```

El

```
abstract class El(val tagName:Strir
    protected val el = when(tagName
        "body" -> document.body ?:
        else -> document.createElen
    }
    var html:String get() = el.inne
                    set(value){el.i
    operator fun get(key:String) =
    operator fun set(key:String, va
    operator fun invoke() = el
    operator fun plusAssign(child:E
    operator fun minusAssign(child:
    val style:CSSStyleDeclaration g
}
```

```
object Body:El("body
class Div:El("div")
class Canvas:El("can
  val context:Canvas
  (el as? HTMLCanvas
}
```

```
fun htmlBuilder(){
    (0..5).map{Div().apply{html = "div-$it"}}.forEach {Body += it}
    Body += Canvas().apply {
        this["width"] = 500
        this["height"] = 500
        context?.run {
            lineWidth = 10.0
            strokeRect(75.0, 140.0, 150.0, 110.0)
            fillRect(130.0, 190.0, 40.0, 60.0)
            moveTo(50.0, 140.0)
            lineTo(150.0, 60.0)
            lineTo(250.0, 140.0)
            closePath()
            stroke()
        }
    }
}
```

```
abstract class El(val tagName:Strin
    protected val el = when(tagName
        "body" -> document.body ?:
        else -> document.createElem
    }
    var html:String get() = el.inne
                    set(value){el.i
    operator fun get(key:String) =
    operator fun set(key:String, va
    operator fun invoke() = el
    operator fun plusAssign(child:E
    operator fun minusAssign(child:
    val style:CSSStyleDeclaration g
}
```

```
object Body:El("body
class Div:El("div")
class Canvas:El("can
  val context:Canvas
  (el as? HTMLCanvas
}
```

```
fun
                                   it"}}.forEach {Body += it}

                                   0, 110.0)
                                    60.0)
}
```

```
div-0
div-1
div-2
div-3
div-4
div-5
```

fetch builder

Param

```
class FetchParam{
    val queries = mutableMapOf<String, Any>()
    val headers = mutableMapOf<String, String>()
    var method = "GET"
}
```

Param

```
class FetchParam{
    val queries = mutableMapOf<String, Any>()
    val headers = mutableMapOf<String, String>()
    var method = "GET"
}
fun fetch(url:String, block:FetchParam.()->Unit)= FetchParam().apply{block()}
```

Param

```
class FetchParam{
    val queries = mutableMapOf<String, Any>()
    val headers = mutableMapOf<String, String>()
    var method = "GET"
}
fun fetch(url:String, block:FetchParam.()->Unit)= FetchParam().apply{block()}.let{
    window.fetch(Request(url, RequestInit()))
}
```

Param

```
class FetchParam{
    val queries = mutableMapOf<String, Any>()
    val headers = mutableMapOf<String, String>()
    var method = "GET"
}
fun fetch(url:String, block:FetchParam.()->Unit)= FetchParam().apply{block()}.let{
    window.fetch(Request(url, RequestInit(
            method = it.method
    )))
}
```

Param

```
class FetchParam{
    val queries = mutableMapOf<String, Any>()
    val headers = mutableMapOf<String, String>()
    var method = "GET"
}
fun fetch(url:String, block:FetchParam.()->Unit)= FetchParam().apply{block()}.let{
    window.fetch(Request(url, RequestInit(
            method = it.method,
            headers = run{
                val obj = js("{}")
                it.headers.forEach {(k, v)->obj[k] = v}
                obj
            }
    )))
}
```

Param

```
class FetchParam{
    val queries = mutableMapOf<String, Any>()
    val headers = mutableMapOf<String, String>()
    var method = "GET"
}
fun fetch(url:String, block:FetchParam.()->Unit)= FetchParam().apply{block()}.let{
    window.fetch(Request(url, RequestInit(
            method = it.method,
            headers = run{
                val obj = js("{}")
                it.headers.forEach {(k, v)->obj[k] = v}
                obj
            },
            body = if(it.method != "GET") it.queries.toList().joinToString("&"){
                    (k, v)->"$k=$v"
                }
                else null
    )))
}
```

Param

```
class FetchParam{
    val que
    val hea
    var met
}

fun fetch                                 .()
     window.fetch(Request(url, RequestInit
               method = it.method,
               headers = run{
                    val obj = js("{}")
                    it.headers.forEach {(k, v
                    obj
               },
               body = if(it.method != "GET") it.queries.toList().joinToString("&"){
                         (k, v)->"$k=$v"
                    }
                    else null
     )))
}
```

```
{
   "test": "testJSON"
}
```

```
fun testFetch(){
     fetch("test.json"){}.then {
          it.text()
     }.then {
          println(it)
     }
}
```

Param

```
class FetchParam{
    val que
    val hea
    var met
}

fun fetch                                      .()
    window.fetch(Request(url, RequestInit
            method = it.method,
            headers = run{
                val obj = js("{}")
                it.headers.forEach {(k, v
                obj
            },
            body = if(it.method != "GET") it
                        (k, v)->"$k=$v"
                    }
                    else null
    )))
}
```

```
{
   "test": "testJSON"
}
```

```
fun testFetch(){
    fetch("test.json"){}.then {
        it.text()
    }.then {
        println(it)
    }
}
```

```
{
  "test": "testJSON"
}
>
```