

Kotlin

Crash course

Павел Галанин



Котлин

A-118

КРОНШТАДТ

A-118

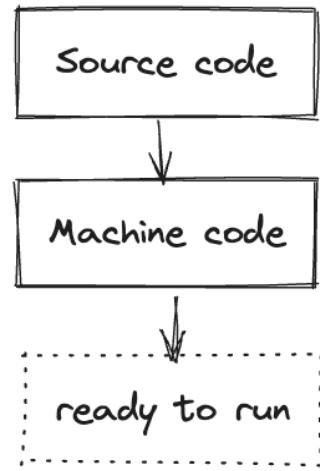
О языке



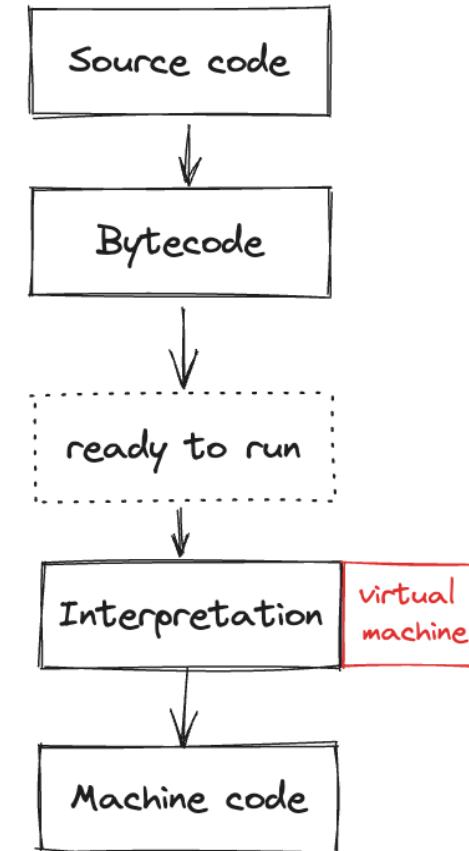
- ① Разрабатывается с 2010 года в компании JetBrains
- ① С 2019 Kotlin-first разработка под Android
- ① Первый релиз – 2016
- ① Актуальная версия – 1.9
- ① Основное применение – backend, Android

Компилируемые и интерпретируемые языки

Компилируемые

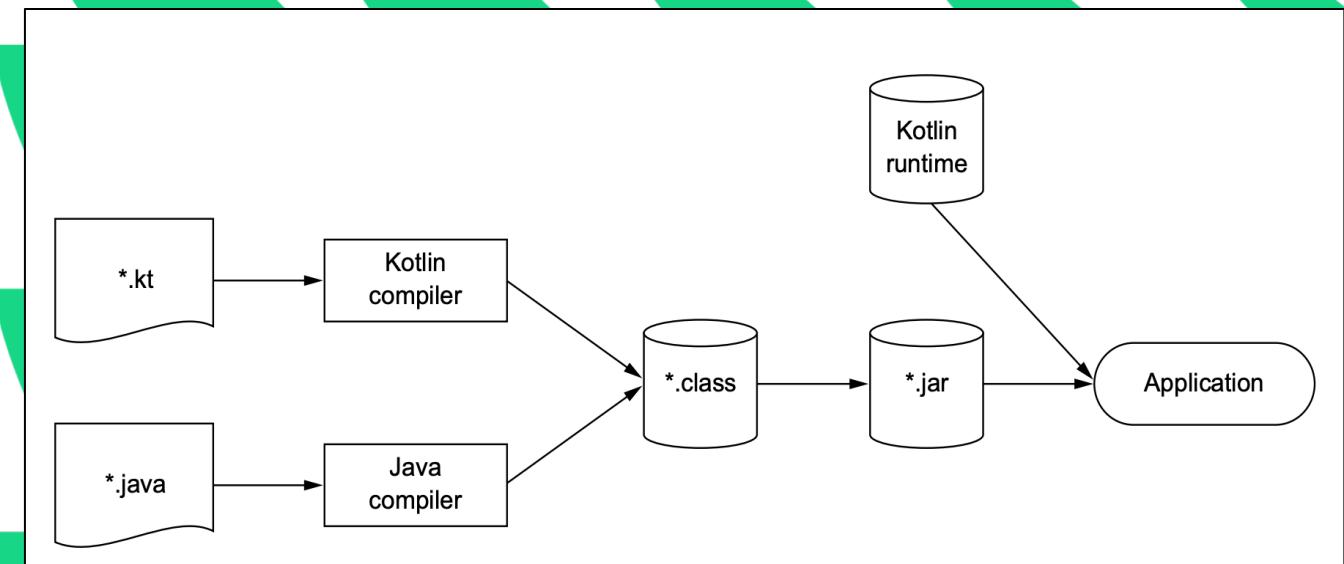


Интерпретируемые



Kotlin и Java

- ❑ Исходники компилируются в файлы .class одного формата
- ❑ Из Java-кода можно вызывать Kotlin-код и наоборот



Сборка и запуск

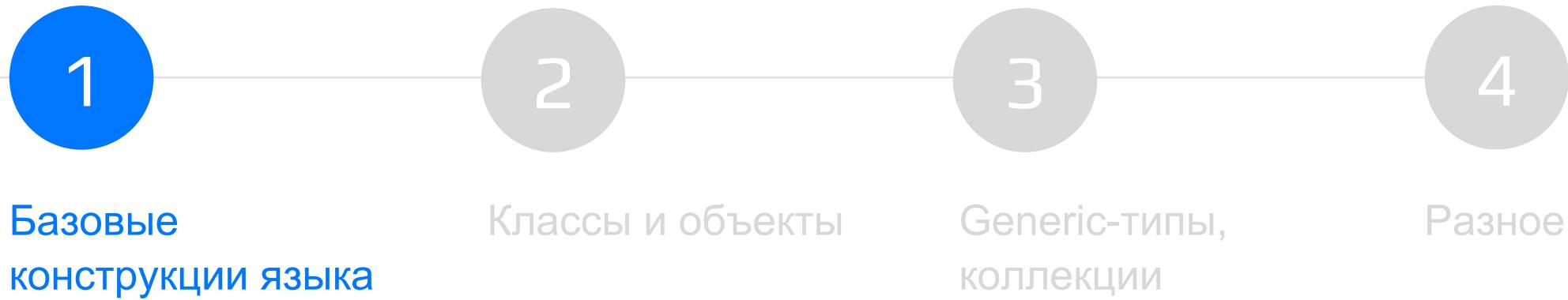
Сборка:

```
kotlinc -include-runtime -d <jar name>
```

Запуск:

```
java -jar <jar name>
```

Вы находитесь здесь



ФУНКЦИИ И ПЕРЕМЕННЫЕ

- функции-выражения
- именные параметры
- параметры по умолчанию

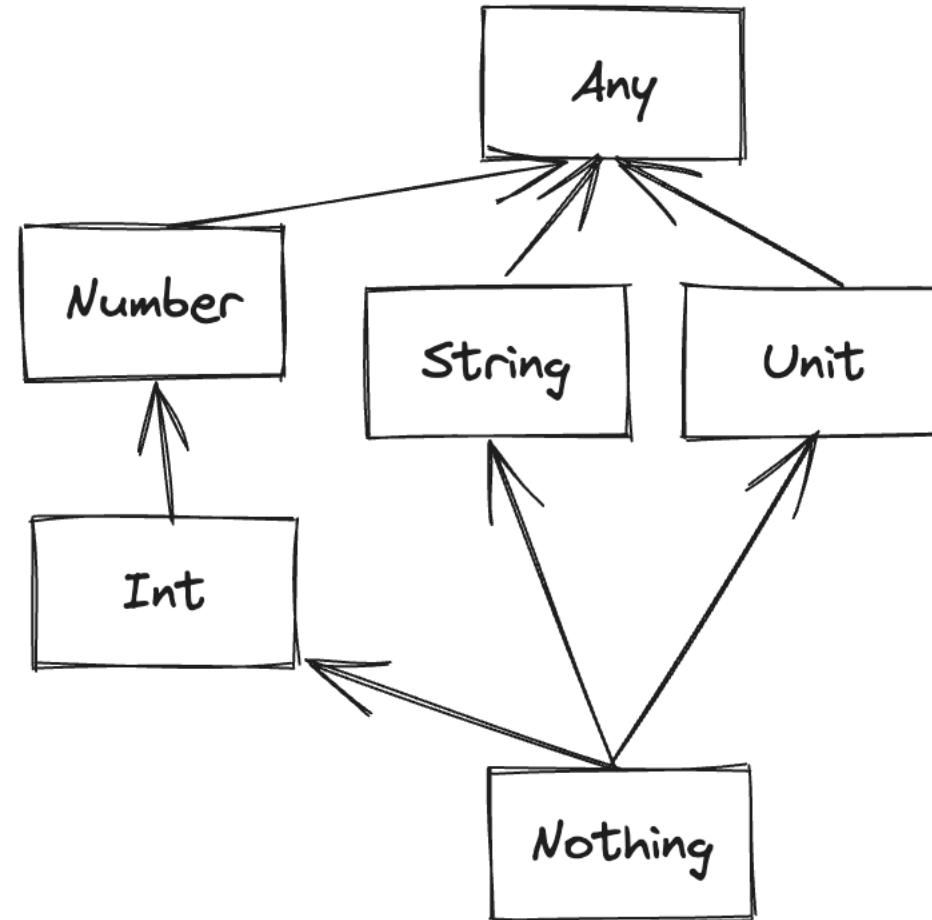
```
fun max(a: Int, b: Int): Int {  
    return if (a > b) a else b  
}  
  
fun max(a: Int, b: Int) = if (a > b) a else b  
  
fun defaultParams(  
    a: String,  
    b: String = "2",  
    c: String = "3",  
) {  
    println("a=$a; b=$b; c=$c")  
}
```

Типы

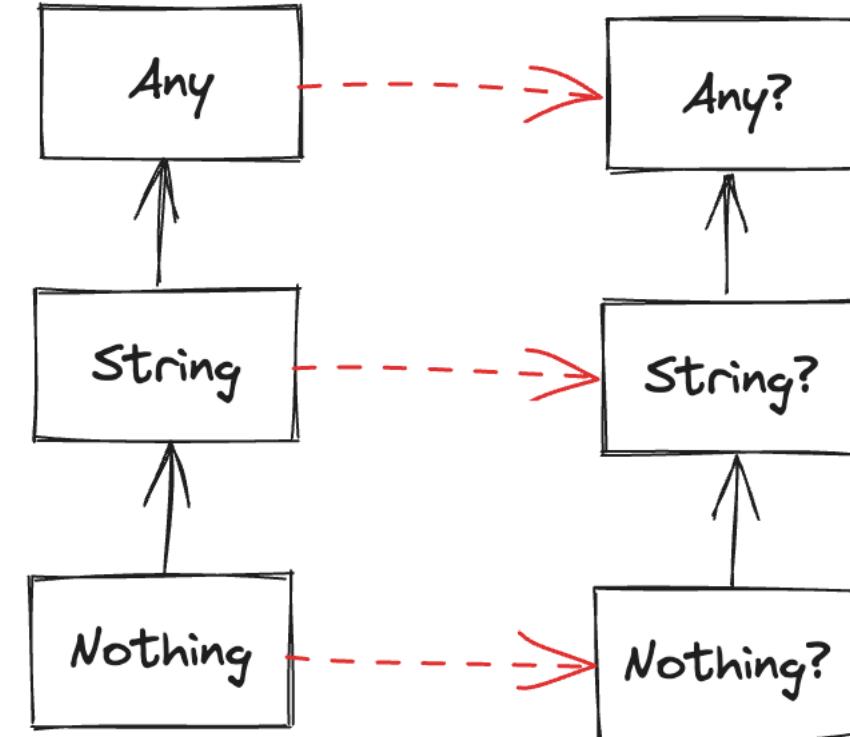
- var/val
- примитивные типы и строки

```
val boolTrue: Boolean = true
val number: Int = 1
val longNumber: Long = 1
val shortNumber: Short = 1
val byteNumber: Byte = 1
val doubleNumber: Double = 1.0
val floatNumber: Float = 1F
val charA = 'a'
```

Встроенные типы Kotlin



Nullable-типы



Условные операторы

- if/else
- when

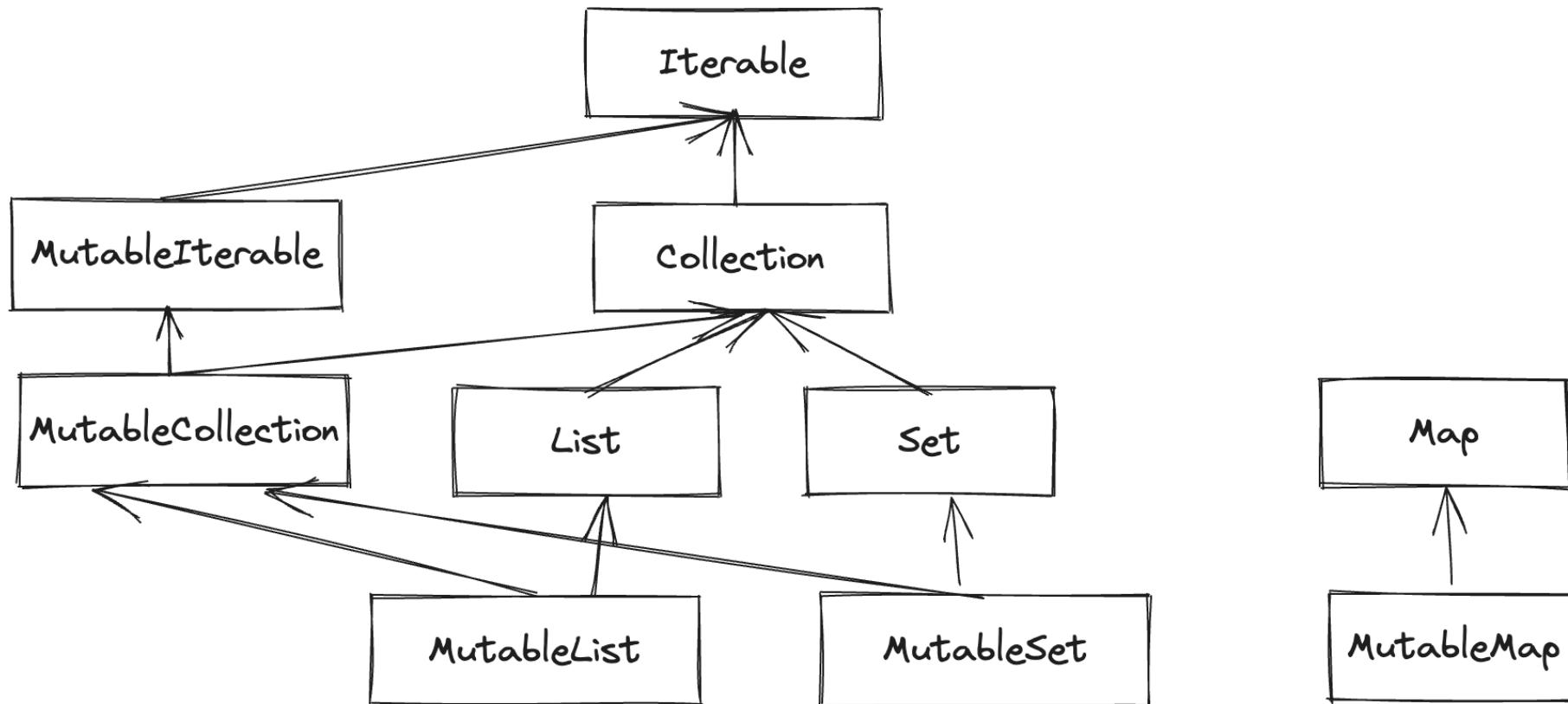
```
fun recognize(c: Char) = when (c) {  
    in '0' .. ≤ '9' -> "It's a digit!"  
    in 'a' .. ≤ 'z', in 'A' .. ≤ 'Z' -> "It's a letter!"  
    else -> "I don't know..."  
}
```

Циклы

- while, do-while
- for
- ranges

```
fun forLoopChar() {  
    for (c in 'A' .. 'z') {  
        println(c)  
    }  
}
```

Базовые коллекции, итератор



Лямбда-выражения

1

Отложенное
исполнение
операций

```
button.setOnClickListener {  
    /* do on click */  
}
```

2

Переиспользование
операций

```
val largestNumber = listOf(10, 40, 5).maxBy { it }  
val lastWord = listOf("Computer", "Code").maxBy { it }
```

Синтаксис лямбда-выражений

Parameters Body

```
{ x: Int, y: Int -> x + y }
```

```
val oldest = people.maxBy { human: Human ->
    human.age
}
```

```
val oldest = people.maxBy { human ->
    human.age
}
```

```
val oldest = people.maxBy { it.age }
```

Лямбда-выражения при работе с коллекциями

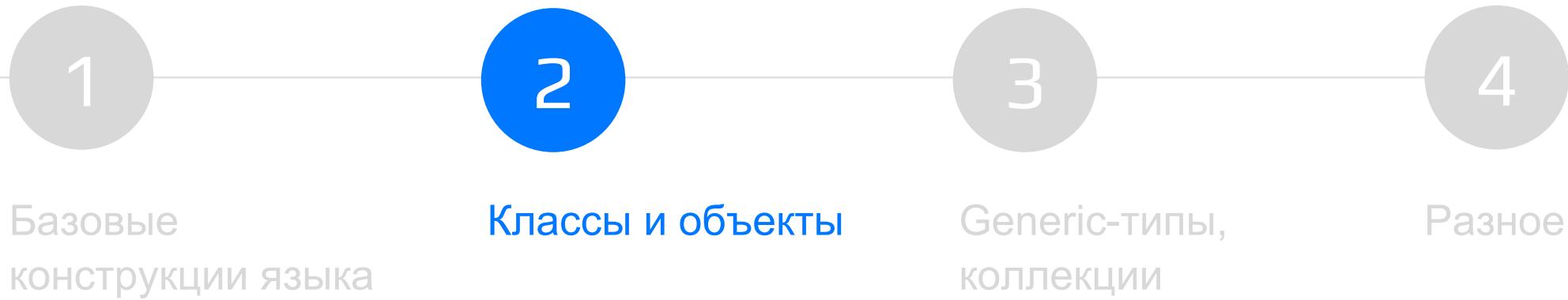
- filter
- map
- all/any
- find
- count
- maxBy
- joinToString
- flatMap

Удобные функции с лямбда-выражениями

- `with`
- `apply`

```
override fun saveState(): Bundle {
    return Bundle().apply { this: Bundle
        // Ensure that even if ViewModels aren't recreated after process deal
        // that we keep their state until they are recreated
        if (restoredState != null) {
            putAll(restoredState)
        }
        // But if we do have ViewModels, prefer their state over what we may
        // have restored
        viewModel.handles.forEach { (key, handle) ->
            val savedState = handle.savedStateProvider().saveState()
            if (savedState != Bundle.EMPTY) {
                putBundle(key, savedState)
            }
        }
    }.also { it: Bundle
        // After we've saved the state, allow restoring a second time
        restored = false
    }
}
```

Вы находитесь здесь



Классы и объекты

- Класс определяет «формат» для однотипных сущностей
- В общем случае класс имеет какие-то данные (поля) и определяет какое-то поведение (методы)
- Объекты – конкретные «представители» классов, «инстансы»

```
1 class Human {  
2     private val name: String  
3     private val weight: Int  
4  
5     constructor(name: String, weight: Int) {  
6         this.name = name  
7         this.weight = weight  
8     }  
9  
10    fun sayYourName() {  
11        println("My name is $name")  
12    }  
13}  
14  
15 val pavel = Human(name: "Pavel", weight: 80)  
16 pavel.sayYourName() // "My name is Pavel"  
17
```

Создание классов

- ☑ Первичный (primary) конструктор – если есть, должен быть обязательно вызван
- ☑ Вторичные (secondary) конструкторы – «альтернативные» способы создания; может быть 0+ вторичных конструкторов

```
class User1 {  
    val _name: String  
    constructor(name: String) {  
        _name = name  
    }  
  
    class User2 constructor(name: String) {  
        val _name: String = name  
    }  
  
    class User3(name: String) {  
        val _name: String = name  
    }  
  
    class User4(name: String) {  
        val _name: String = name  
        constructor(nameSurname: Pair<String, String>) : this(nameSurname.first)  
    }  
  
    class User5 {  
        val _name: String  
        val _surname: String  
  
        constructor(name: String, surname: String) {  
            _name = name  
            _surname = surname  
        }  
        constructor(name: String) : this(name, surname: "-")  
        constructor(nameSurname: Pair<String, String>) {  
            _name = nameSurname.first  
            _surname = nameSurname.second  
        }  
    }  
}
```

БЛОКИ ИНИЦИАЛИЗАЦИИ



Можно воспринимать как дополнение к первичному конструктору

```
class Shape(rawColor: String) {  
    val color: String  
  
    init {  
        println("Create color $rawColor")  
        color = rawColor  
    }  
}
```

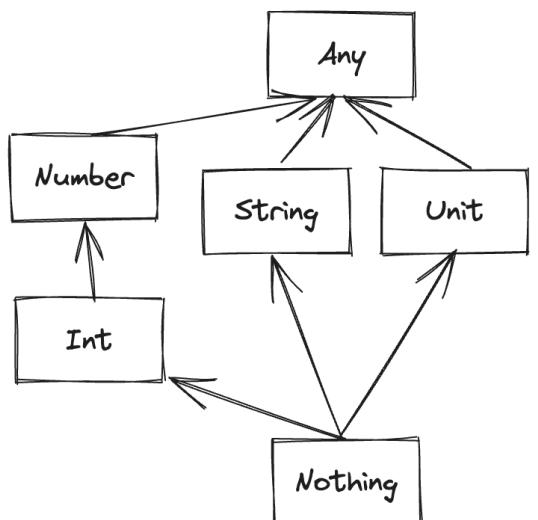
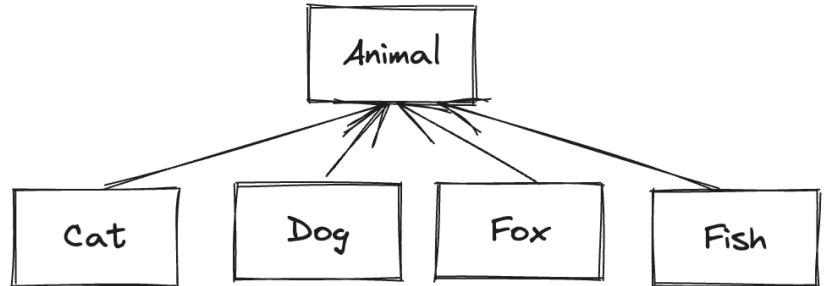
Поля



в общем случае хранят состояние объекта

```
class User(val name: String) {  
    var address: String = "unspecified"  
    get(): String {  
        println("Address was requested")  
        return field  
    }  
    set(value: String) {  
        println("Address was changed for $name: field -> $value")  
        field = value  
    }  
}
```

Наследование



```
1 abstract class Animal(val weight: String) {
2     abstract fun whatDoYouSay()
3 }
4
5 class Cat(weight: String) : Animal(weight) {
6     override fun whatDoYouSay() {
7         println("Meow, meow")
8     }
9 }
10
11 class Dog(weight: String) : Animal(weight) {
12     override fun whatDoYouSay() {
13         println("Woof, woof")
14     }
15 }
16 class Fox(weight: String) : Animal(weight) {
17     override fun whatDoYouSay() {
18         println("https://www.youtube.com/watch?v=jofNR_WkoCE") ⏪
19     }
20 }
21
22 class Fish(weight: String) : Animal(weight) {
23     override fun whatDoYouSay() {
24         throw NotImplementedError()
25     }
26 }
```

Все классы наследуются от Any



Это значит, что у всех классов есть методы:

- equals
- hashCode
- toString

package kotlin

The root of the Kotlin class hierarchy. Every Kotlin class has Any as a superclass.

public open class Any {

Indicates whether some other object is "equal to" this one. Implementations must fulfil the following requirements:

- Reflexive: for any non-null value x , $x.equals(x)$ should return true.
- Symmetric: for any non-null values x and y , $x.equals(y)$ should return true if and only if $y.equals(x)$ returns true.
- Transitive: for any non-null values x , y , and z , if $x.equals(y)$ returns true and $y.equals(z)$ returns true, then $x.equals(z)$ should return true.
- Consistent: for any non-null values x and y , multiple invocations of $x.equals(y)$ consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
- Never equal to null: for any non-null value x , $x.equals(null)$ should return false.

Read more about [equality ↗](#) in Kotlin.



public open operator fun equals(other: Any?): Boolean

Returns a hash code value for the object. The general contract of hashCode is:

- Whenever it is invoked on the same object more than once, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified.
- If two objects are equal according to the equals() method, then calling the hashCode method on each of the two objects must produce the same integer result.



public open fun hashCode(): Int



Returns a string representation of the object.



public open fun toString(): String



Переопределение методов



Метод можно переопределить с помощью модификатора `override`



Можно переопределить:

- абстрактный метод
- метод интерфейса
- метод с модификатором `open`



В Java по умолчанию можно переопределять, в том числе из Kotlin

```
1 abstract class Animal(val weight: String) {  
2     abstract fun whatDoYouSay()  
3 }  
4  
5 class Cat(weight: String) : Animal(weight) {  
6     override fun whatDoYouSay() {  
7         println("Meow, meow")  
8     }  
9 }  
10  
11 class Dog(weight: String) : Animal(weight) {  
12     override fun whatDoYouSay() {  
13         println("Woof, woof")  
14     }  
15 }  
16 class Fox(weight: String) : Animal(weight) {  
17     override fun whatDoYouSay() {  
18         println("https://www.youtube.com/watch?v=jofNR_WkoCE")  
19     }  
20 }  
21  
22 class Fish(weight: String) : Animal(weight) {  
23     override fun whatDoYouSay() {  
24         throw NotImplementedError()  
25     }  
26 }
```

Сравнение объектов

1

Сравнение ссылок
(идентичности)

`==`

`a === b`

2

Проверка (структурного)
равенства

`==`

`a == b`

2

Проверка (структурного) через equals
равенства

`==, equals`

`a == b`
`a.equals(b)` · `b.equals(a)`

Контракт equals и hashCode

- ☐ Если $x.equals(y)$, то $x.hashCode() == y.hashCode()$
- ☐ Если $x.hashCode() == y.hashCode()$, то $x.equals(y) == ?$

Полиморфизм



Полиморфизм – это способность программы идентично использовать объекты с одинаковым интерфейсом без информации о конкретном типе этого объекта.

```
1 open class PriceTag(protected val originPrice: Int) {
2     open fun printPrice() {
3         println("Price: $originPrice")
4     }
5 }
6
7 class YellowPriceTag(originPrice: Int) : PriceTag(originPrice) {
8     override fun printPrice() {
9         println("DISCOUNT! Price: ${originPrice * 0.5}")
10    }
11 }
12
13 fun main() {
14     val priceTag: PriceTag = YellowPriceTag(originPrice: 100)
15     val yellowPriceTag: YellowPriceTag = YellowPriceTag(originPrice: 100)
16     priceTag.printPrice()
17     yellowPriceTag.printPrice()
18
19     fun priceTagPrinter(priceTag: PriceTag) {
20         priceTag.printPrice()
21     }
22     priceTagPrinter(priceTag)
23     priceTagPrinter(yellowPriceTag)
24 }
25
```

Абстрактные классы

- ☐ Абстрактный класс содержит хотя бы один абстрактный метод
- ☐ Абстрактный метод не имеет реализации – её должны определять наследники
- ☐ Нельзя создать объект абстрактного класса

```
1 abstract class Animal(val weight: String) {  
2     abstract fun whatDoYouSay()  
3 }  
4  
5 class Cat(weight: String) : Animal(weight) {  
6     override fun whatDoYouSay() {  
7         println("Meow, meow")  
8     }  
9 }  
10  
11 class Dog(weight: String) : Animal(weight) {  
12     override fun whatDoYouSay() {  
13         println("Woof, woof")  
14     }  
15 }  
16 class Fox(weight: String) : Animal(weight) {  
17     override fun whatDoYouSay() {  
18         println("https://www.youtube.com/watch?v=jofNR_WkoCE")  
19     }  
20 }  
21  
22 class Fish(weight: String) : Animal(weight) {  
23     override fun whatDoYouSay() {  
24         throw NotImplementedError()  
25     }  
26 }
```

Интерфейсы

- ↗ Позволяют выделять из разных объектов общую функциональность и за счёт этого единообразно использовать эти объекты
- ↗ Класс может реализовывать несколько интерфейсов
- ↗ Методы интерфейсов могут иметь реализацию по умолчанию
- ↗ Интерфейсы могут наследоваться от других интерфейсов

```
interface Clickable {  
    fun click()  
}  
  
class A : Clickable {  
    override fun click() {  
        println("Click A")  
    }  
}  
  
class B : Clickable {  
    override fun click() {  
        println("Click B")  
    }  
}  
  
fun clicker(clickable: Clickable) {  
    clickable.click()  
}
```

Виды классов: data class

checkbox используется для классов, основная функция которых – «обёртка» над данными

checkbox обязательно должен содержать поля, определённые в конструкторе

checkbox при компиляции генерируется несколько методов, среди которых equals, hashCode, toString, copy

checkbox нельзя унаследоваться от data class

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

Виды классов: enum

- ☑ При определении enum-классов создаются все возможные экземпляры класса
- ☑ Больше экземпляры enum-классов не могут ниоткуда взяться

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

Виды классов: sealed

- sealed – англ. «запечатанный»
- расширение концепции enum-классов – зафиксированы все возможные наследники класса
- наследники могут располагаться только в том же пакете
- можно определять sealed-интерфейсы

```
sealed class Result

data class Loaded(val data: String) : Result()
data class Error(val description: String) : Result()
object NotModified : Result()

fun parseResult(result: Result) {
    when (result) {
        is Loaded -> println(result.data)
        is Error -> println(result.description)
        is NotModified -> println(result)
    }
}
```

Виды классов: object

- singleton из коробки
- к «обычному» классу можно добавить companion object – сымитировать (Java-) статические члены
- в объектах поля могут быть помечены как const

Внутренние и вложенные классы

	Вложенный (nested)	Внутренний (inner)
Объявляются внутри другого класса	Да	Да
Модификатор объявления класса	(по умолчанию)	inner
Имеют доступ к private-членам класса	Да (статическим)	Да
Требуют для создания экземпляр внешнего класса	Нет	Да
Имеют ссылку на экземпляр внешнего класса	Нет	Да

Пакеты

- пакеты (packages) позволяют группировать файлы исходного кода
- пакет файла может не совпадать с путём к файлу
- можно определять функции и поля на уровне пакета
- с помощью `import kotlin.coroutines.cancellation.*` можно импортировать все классы пакета

```
package kotlin.coroutines.cancellation

Thrown by cancellable suspending functions if the coroutine is cancelled while it is suspended.  
It indicates normal cancellation of a coroutine.

@SinceKotlin( version: "1.4")
public expect open class CancellationException : IllegalStateException {
    public constructor()
    public constructor(message: String?)
}

▼ □ Gradle: org.jetbrains.kotlin:kotlin-stdlib-common:1.8.10
  ▼ □ kotlin-stdlib-common-1.8.10.jar library root
    ▼ □ kotlin
      ▷ □ collections
      ▷ □ comparisons
      ▷ □ contracts
      ▼ □ coroutines
        ▼ □ cancellation
          □ CancellationException
          □ CancellationExceptionHkt.kotlin_metadata
        ▷ □ intrinsics
        □ AbstractCoroutineContextElement
```

Модификаторы доступа

	Члены класса	Объявления на уровне пакета
public (по умолчанию)	Доступны везде	Доступны везде
internal	Доступны внутри модуля	Доступны внутри модуля
protected	Доступны в подклассах	-
private	Доступны из того же класса	Доступны из того же файла

Extension-функции

- ↗ Позволяют добавлять функциональность в классы и интерфейсы, не модифицируя их
- ↗ Компилируются в обычные функции с добавленным параметром – объектом расширяемого класса

```
public interface CoroutineScope {
```

The context of this scope. Context is encapsulated by the scope and used for implementation of coroutine builders that are extensions on the scope. Accessing this property in general code is not recommended for any purposes except accessing the `Job` instance for advanced usages.

By convention, should contain an instance of a `Job` to enforce structured concurrency.

```
    public val coroutineContext: CoroutineContext
```

```
}
```

```
    public fun CoroutineScope.ensureActive(): Unit = coroutineContext.ensureActive()
```

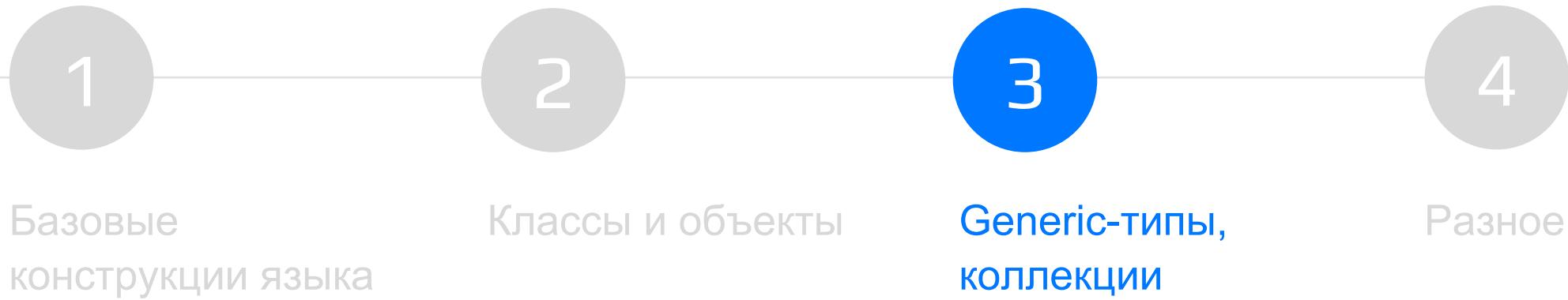
```
@Suppress(...names: "FunctionName")
```

```
    public fun CoroutineScope(context: CoroutineContext): CoroutineScope =  
        ContextScope(if (context[Job] != null) context else context + Job())
```

```
    public val CoroutineScope.isActive: Boolean
```

```
        get() = coroutineContext[Job]?.isActive ?: true
```

Вы находитесь здесь



Generic-типы

- позволяют использовать единую логику для разных типов объектов
- могут определяться для всего класса, для конкретной функции или поля
- стирание типов** (type erasure) – информация о generic-типе не хранится в скомпилиированном коде: данные хранятся в виде Any (Object), в местах использования – приведение к нужному типу
 - исключение: inline-функции с reified-параметром
- <*> - может быть использован любой тип (тип неизвестен или неважен)

Коллекции

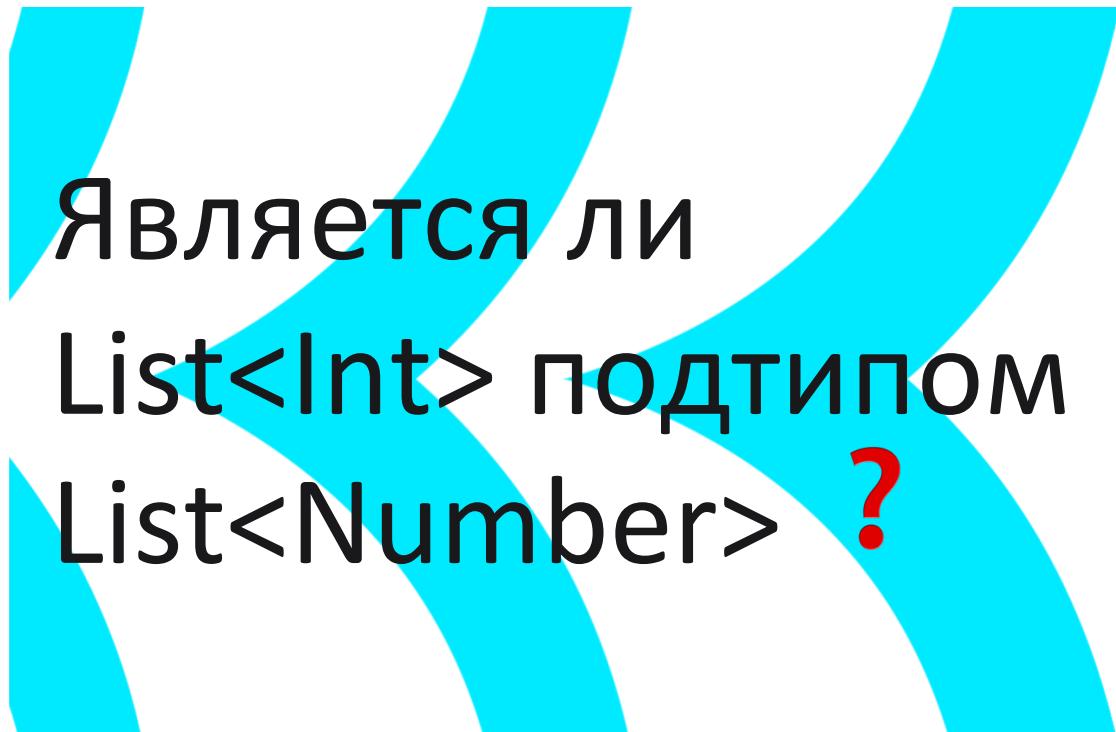
java.util.List<T>

- ArrayList<T> - массив (arrayListOf, listOf, mutableListOf)
- LinkedList<T> - связный список

java.util.Set<T>

- HashSet<T> - сет
- LinkedHashSet<T> - сет, отслеживающий порядок добавления (setOf, mutableSetOf)

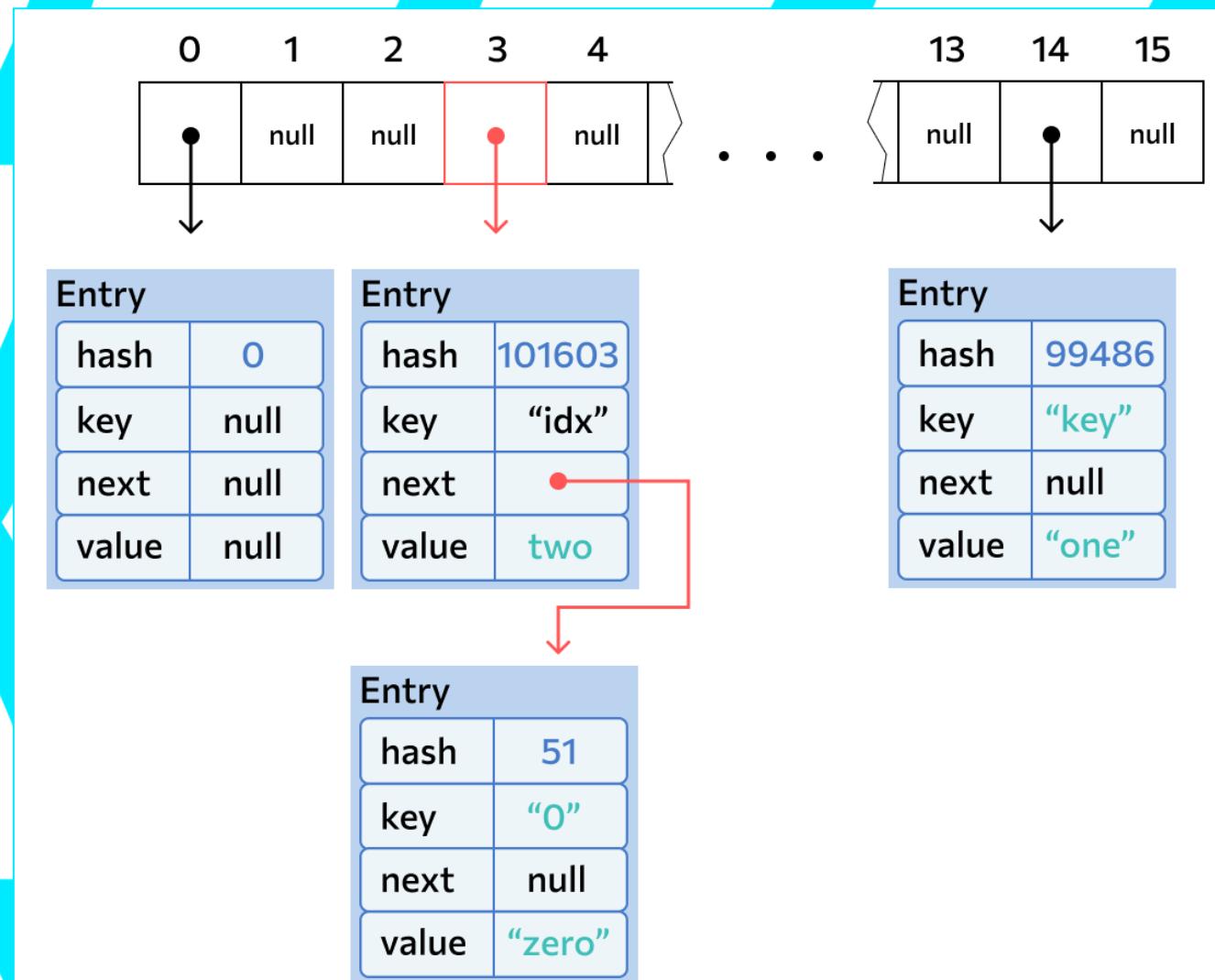
Generic-типы: вариация



- тип Generic - **инвариантный**, если
В подтип A $\not\rightarrow$ Generic подтип Generic<A>
- тип Generic - **ковариантный**, если
В подтип A \rightarrow Generic подтип Generic<A>
- тип Generic - **контравариантный**, если
В подтип A \rightarrow Generic<A> подтип Generic
- Generic<T> - инвариантный
Generic<**out** T> - ковариантный
Generic<**in** T> - контравариантный

HashMap

- ☐ позволяет «быстро» получать значения по ключу
- ☐ hashCode ключа определяет [бакет](#), где хранится значение
- ☐ при возникновении [коллизии](#) в бакете формируется связный список из значений
- ☐ для поиска нужного значения в бакете применяется equals



Вы находитесь здесь

1

Базовые
конструкции языка

2

Классы и объекты

3

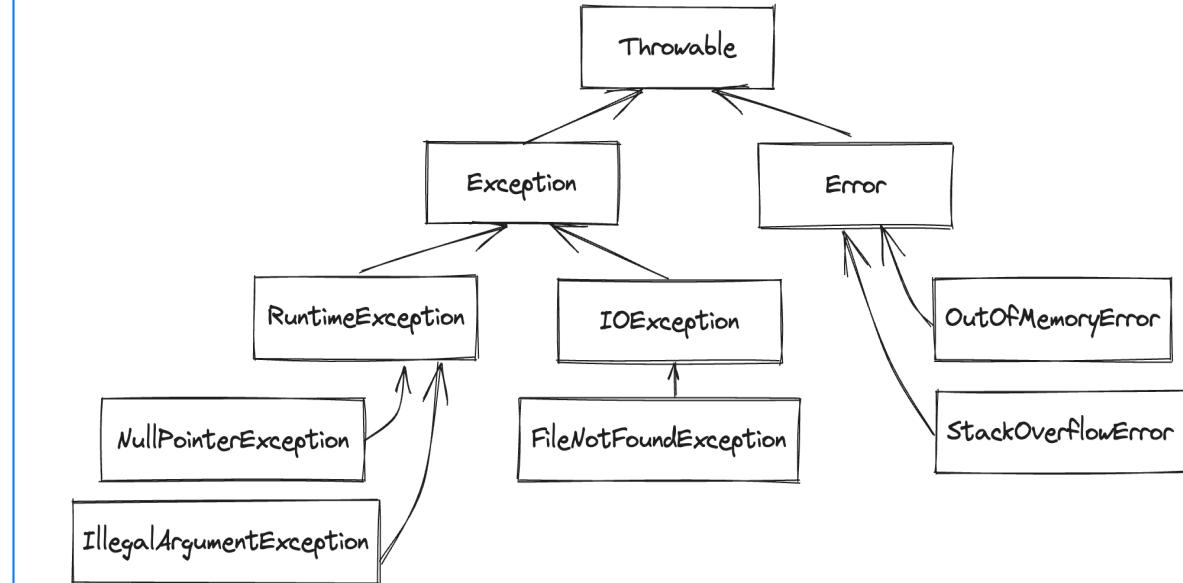
Generic-типы,
коллекции

4

Разное

Исключения

- Указывают на появление нештатной ситуации
- Прерывают исполнение программы, но можно перехватить, обработать и продолжить выполнение – с помощью try-catch
- «Выбрасываются» с помощью оператора throw



```
private inline fun <T> screenFloatValue(str: String, parse: (String) -> T): T? {  
    return try {  
        if (ScreenFloatValueRegEx.value.matches(str))  
            parse(str)  
        else  
            null  
    } catch (e: NumberFormatException) { // overflow  
        null  
    }  
}  
  
public override fun nextByte(): Byte {  
    prepareNext()  
    if (finished)  
        throw NoSuchElementException("Input stream is over.")  
    val res = nextByte.toByte()  
    nextPrepared = false  
    return res  
}
```

finally

- Позволяет выполнить действие независимо от исключений
- Чаще всего используется для очистки ресурсов

Executes the given `action` under this lock.

Returns: the return value of the action.

```
@kotlin.internal.InlineOnly
public inline fun <T> Lock.withLock(action: () -> T): T {
    contract { callsInPlace(action, InvocationKind.EXACTLY_ONCE) }
    lock()
    try {
        return action()
    } finally {
        unlock()
    }
}
```

Аннотации



Добавляют «мета»-информацию к коду



Информация может использоваться на уровне исходного кода, при компиляции или во время исполнения

```
@Deprecated(message: "Will be removed soon")
class Rectangle {
    var width: Int = 0
    var height: Int = 0

@RunWith(AndroidJUnit4::class)
class ExampleInstrumentedTest {
    @Test
    fun useApplicationContext() {
        // Context of the app under test.
        val applicationContext = InstrumentationRegistry.getInstrumentation().targetContext
        assertEquals(expected: "com.pavel.kotlin", applicationContext.packageName)
    }
}

@Suppress(...names: "EXTENSION_SHADOWED_BY_MEMBER")
public val CoroutineScope.isActive: Boolean
    get() = coroutineContext[Job]?.isActive ?: true

@Composable
fun Greeting(name: String) {
    Text("Hello $name")
}
```

Делегаты

- ☑ Делегаты классов – позволяют делегировать часть реализации другим объектам
- ☑ Делегаты свойств – позволяют делегировать логику записи и/или чтения свойства другому классу

```
class ListWithGetCounter(  
    private val innerList: List<String>  
) : List<String> by innerList {  
    var counter = 0  
    private set  
  
    override fun get(index: Int): String {  
        counter++  
        return innerList[index]  
    }  
  
}  
  
class WithLazy {  
    val className: String by lazy {  
        println("lazy called")  
        javaClass.simpleName ^lazy  
    }  
}
```

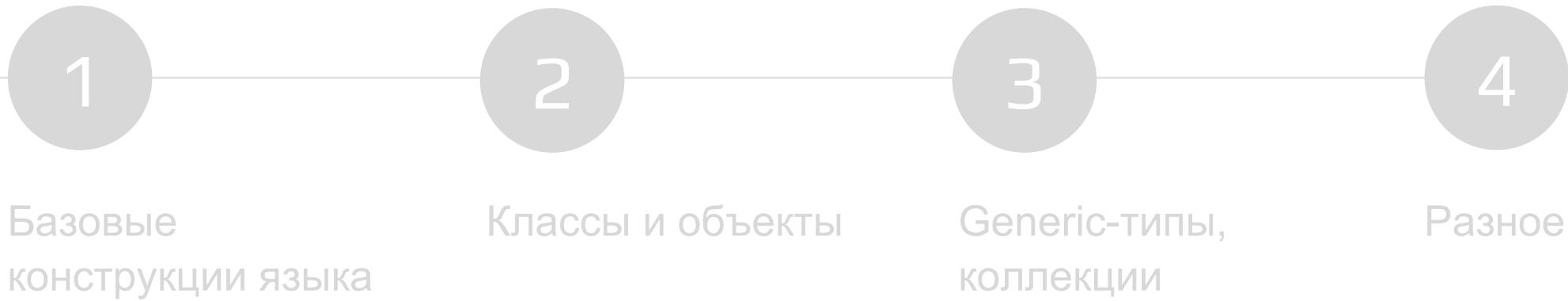
Рефлексия



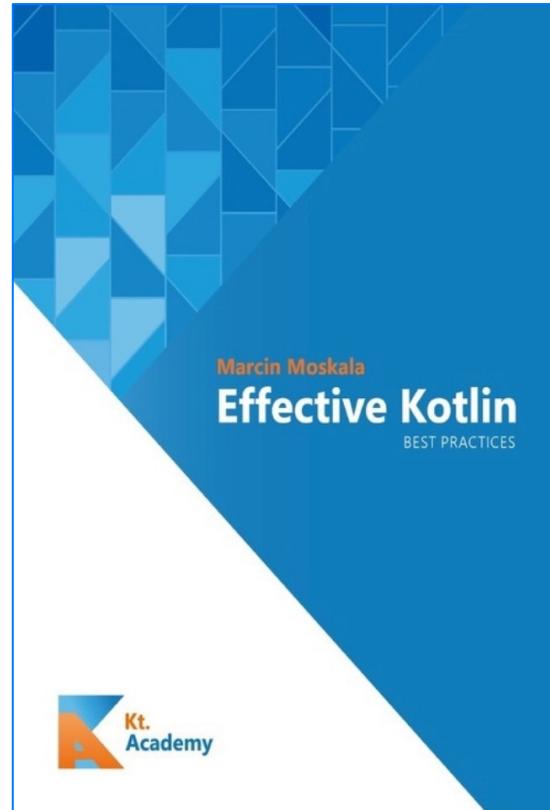
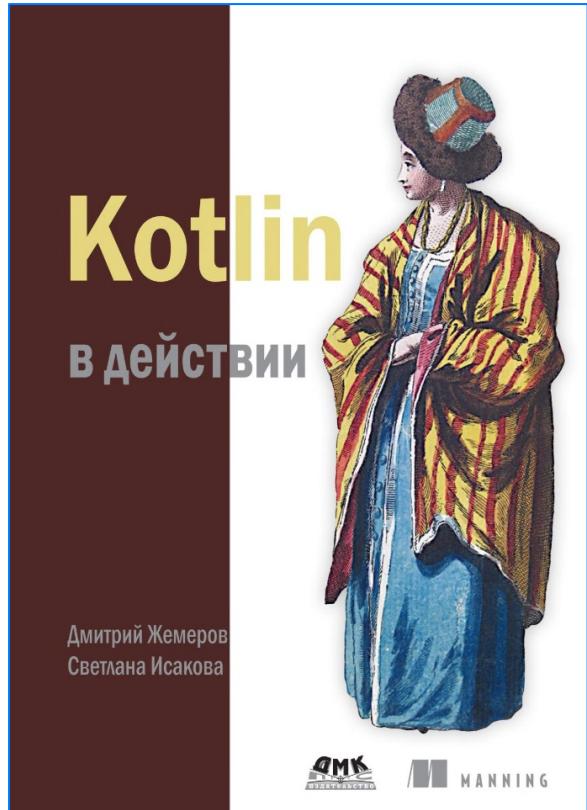
Позволяет **динамически** во время исполнения получить доступ к свойствам, методам и т.д.

```
class ClassToReflect {  
    val timeCreated = System.currentTimeMillis()  
  
    fun callMe(): String {  
        return "Thank you for calling"  
    }  
}  
  
fun main() {  
    val obj = ClassToReflect()  
    val kClass = obj.javaClass.kotlin  
    println(kClass.members.joinToString { it.name })  
  
    val prop = kClass.memberProperties.find { it.name == "timeCreated" }  
    val method = kClass.memberFunctions.find { it.name == "callMe" }  
    println("Reflected: timeCreated=${prop?.get(obj)} callMe=${method?.call(obj)}")  
}
```

Вы находитесь здесь



Рекомендую



Спасибо
за внимание!

Павел Галанин