Програмування для мобільних додатків

Мова програмування Kotlin

Тема 01, ЧДБК 2019

План лекції

01 **Базовий синтаксис мови програмування Kotlin**

Управління ходом виконання програми

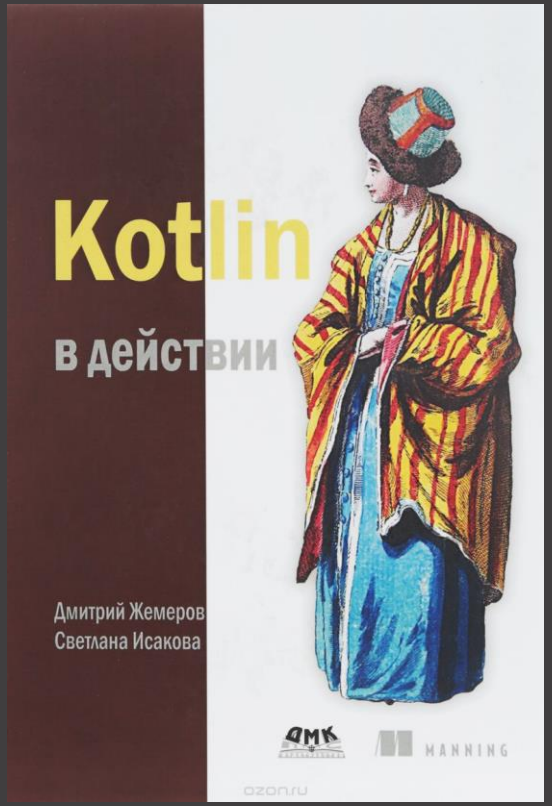
02 **Об’єктно-орієнтоване програмування засобами Kotlin**

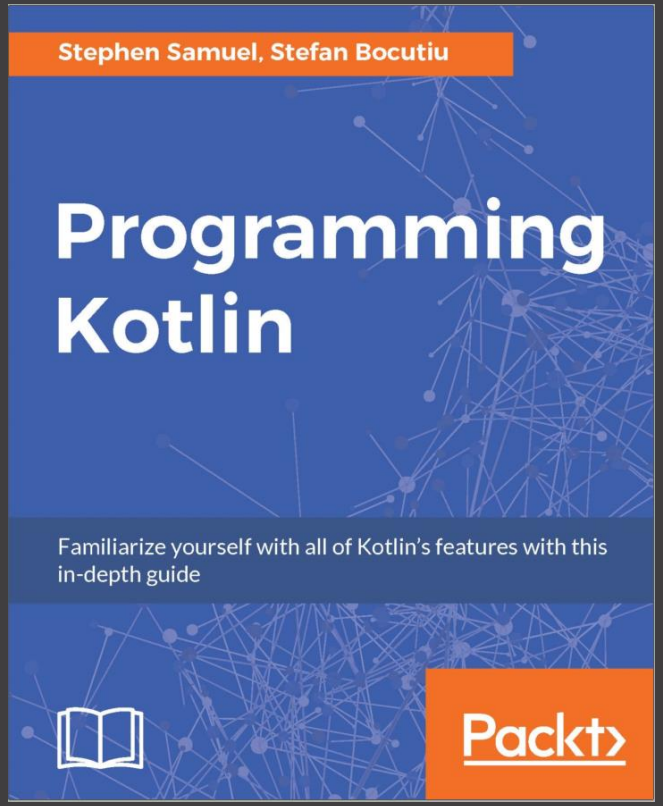
03 **Основи функціонального програмування в Kotlin**

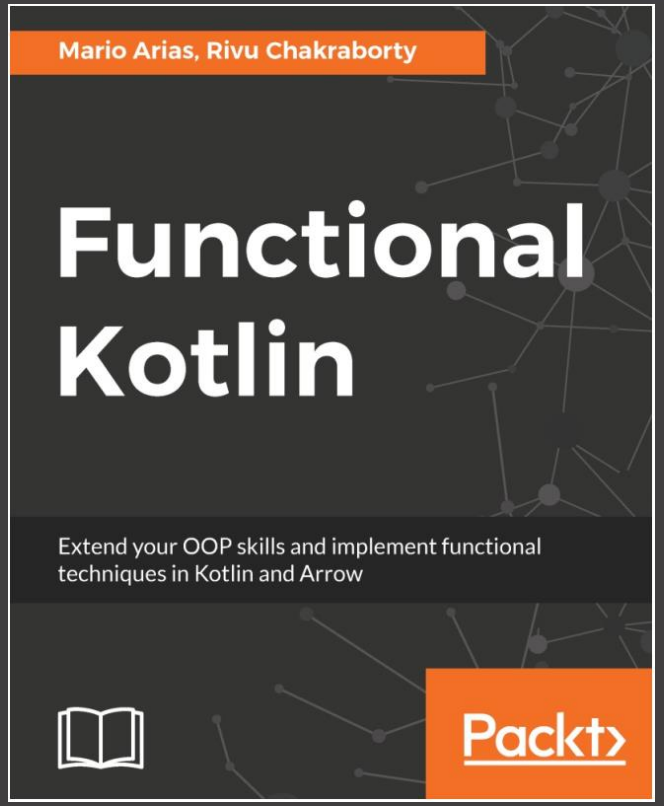
04 **Система типів у мові програмування Kotlin**

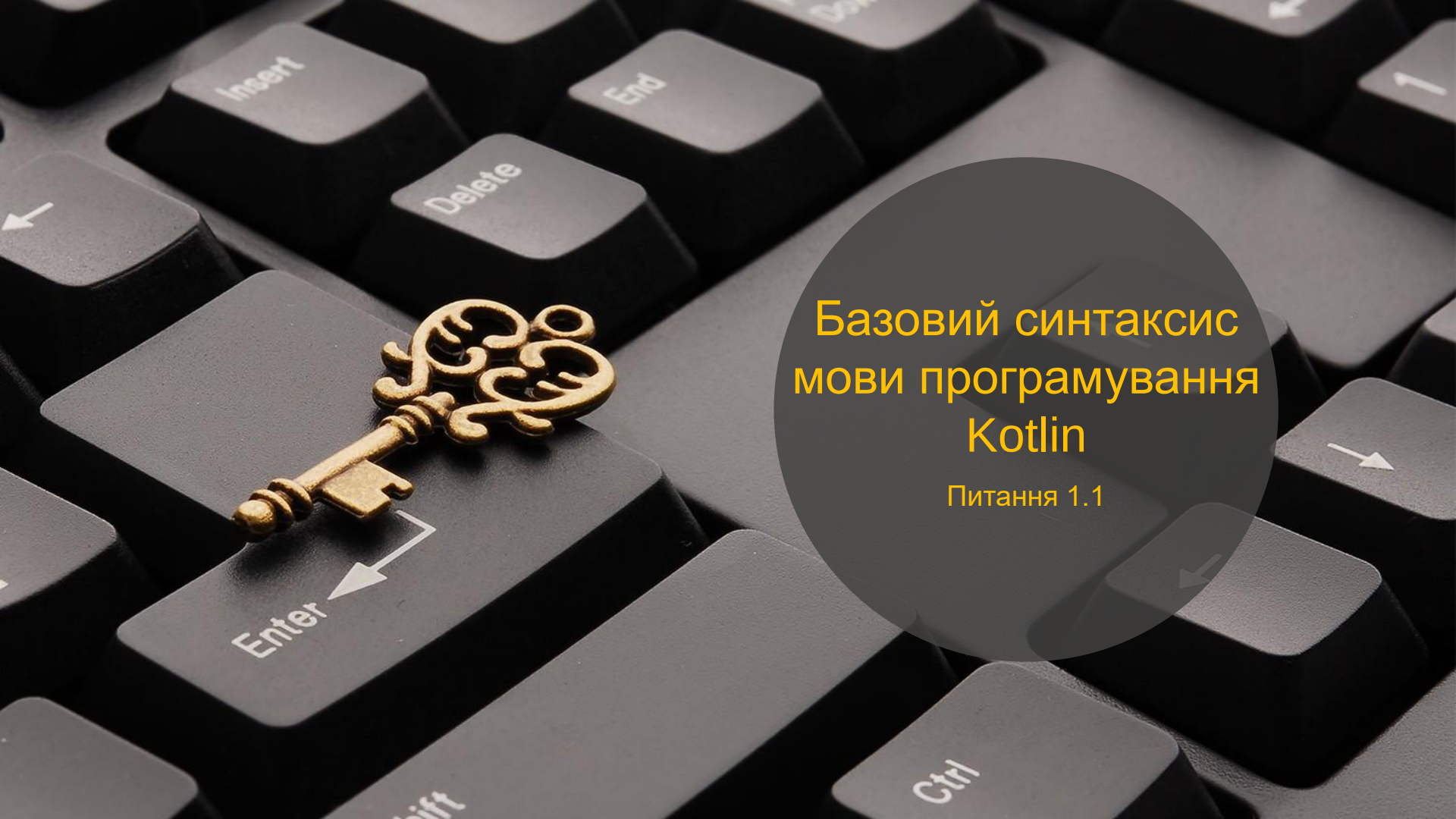
Узагальнене програмування

Література


Kotlin
в действии
Дмитрий Жемеров
Светлана Исакова
ДМК
MANNING
ozon.ru


Stephen Samuel, Stefan Bocutiu
Programming Kotlin
Familiarize yourself with all of Kotlin's features with this in-depth guide
Packt>


Mario Arias, Rivu Chakraborty
Functional Kotlin
Extend your OOP skills and implement functional techniques in Kotlin and Arrow
Packt>



Базовий синтаксис мови програмування Kotlin

Питання 1.1

Мова програмування Kotlin

Мета нової мови програмування – запропонувати більш компактну, продуктивну та без печну альтернативу мові Java.

Типові галузі застосування:

- Розробка коду, який працює на стороні сервера;
- Створення додатків, що працюють як на Android-пристроях, та на iOS (за допомогою Intel Multi-OS Engine).
- Можна писати й настільні додатки (у поєднанні з TornadoFX або JavaFX)

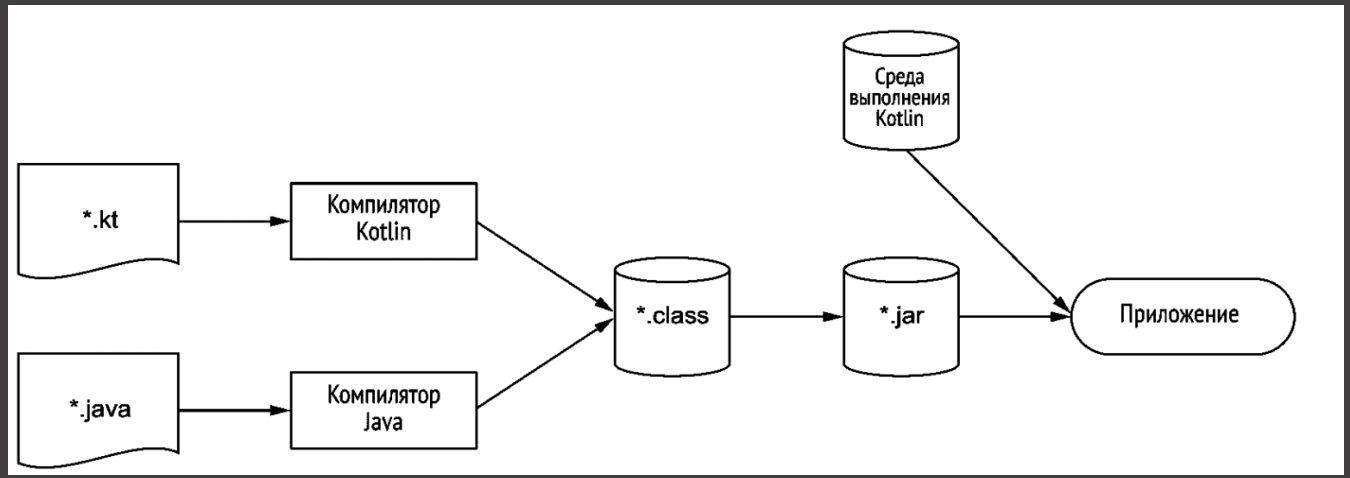
Компілятор мови, бібліотеки та інструменти розробки частіше за все мають відкритий код та розповсюджуються за умовах

Збірка Kotlin-проектів

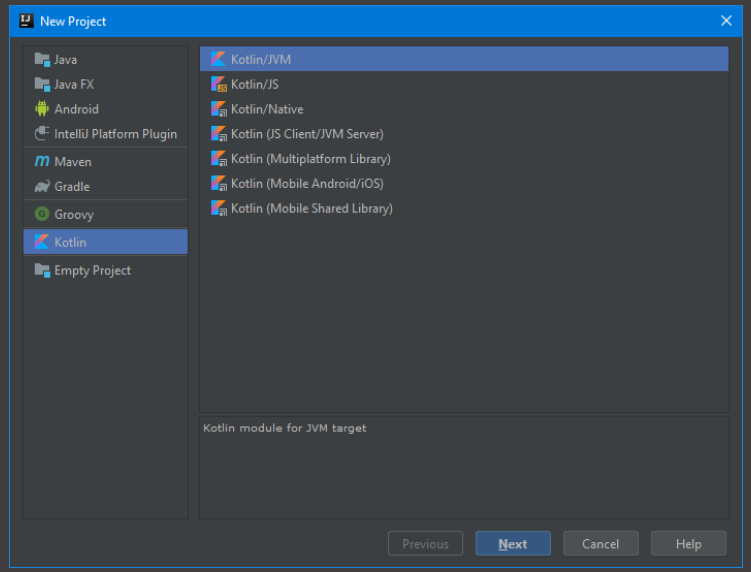
Первинний код на Kotlin зазвичай зберігається в .kt-файлах.

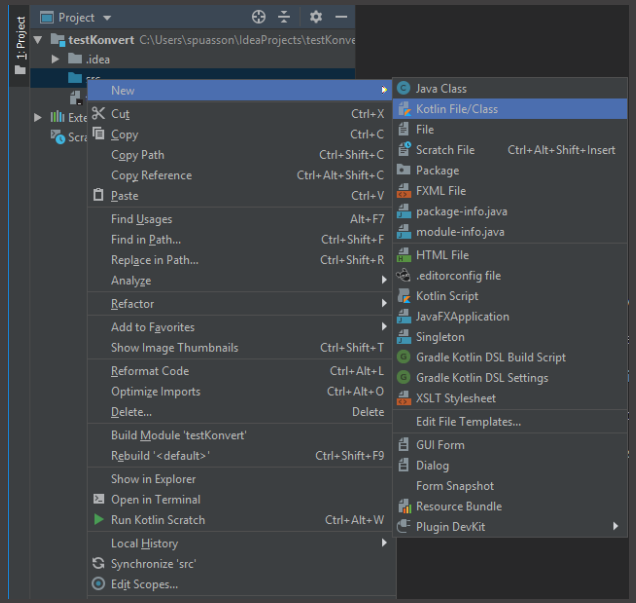
- Компілятор аналізує код та генерує class-файли
- Далі вони упаковуються та виконуються

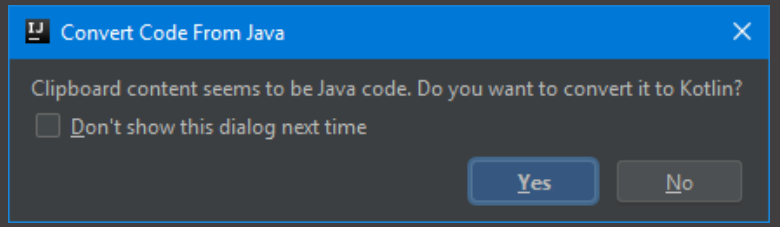
```
kotlinc <первинний файл або каталог> -include-runtime -d <назва jar-файла>
java -jar <назва jar-файла>
```



Конвертація Java-коду в Kotlin







Порівняння коду

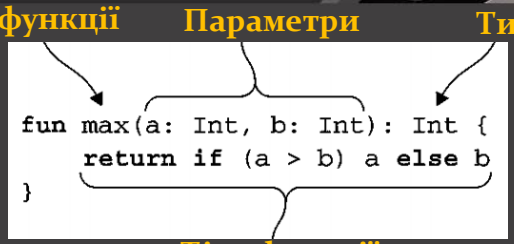
```java
public class JavaCode {
    public String toJSON(Collection<Integer> collection) {
        StringBuilder sb = new StringBuilder();
        sb.append("[");
        Iterator<Integer> iterator = collection.iterator();
        while (iterator.hasNext()) {
            Integer element = iterator.next();
            sb.append(element);
            if (iterator.hasNext()) {
                sb.append(", ");
            }
        }
        sb.append("]");
        return sb.toString();
    }
}
```

```kotlin
class JavaCode {
    fun toJSON(collection: Collection<Int>): String {
        val sb = StringBuilder()
        sb.append("[")
        val iterator = collection.iterator()
        while (iterator.hasNext()) {
            val element = iterator.next()
            sb.append(element)
            if (iterator.hasNext()) {
                sb.append(", ")
            }
        }
        sb.append("]")
        return sb.toString()
    }
}
```

Конвертер також доступний в Eclipse

Hello, world! мовою Kotlin

```
fun main(args: Array<String>) {
    println("Hello, world")
}
```



- Оголошення функцій починаються з ключового слова fun
- Тип параметру вказується після його назви (так і при оголошенні змінних)
- Функцію можна оголосити на верхньому рівні в файлі, поміщати в клас не обов'язково
- Масиви – це просто класи. На відміну від Java, в Kotlin немає спеціального синтаксису для оголошення масивів
- Замість System.out.println() можна писати просто println(). Стандартна бібліотека Kotlin в ключає багато обгорток для функцій зі стандартної бібліотеки Java.
- Крапку з комою можна опускати

if у мові Kotlin є виразом (має значення), а не інструкцією (оператором МП).

- У мові Java всі управляючі структури – інструкції
- У мові Kotlin більшість управляючих структур – вирази (крім циклів for, do, do/while)

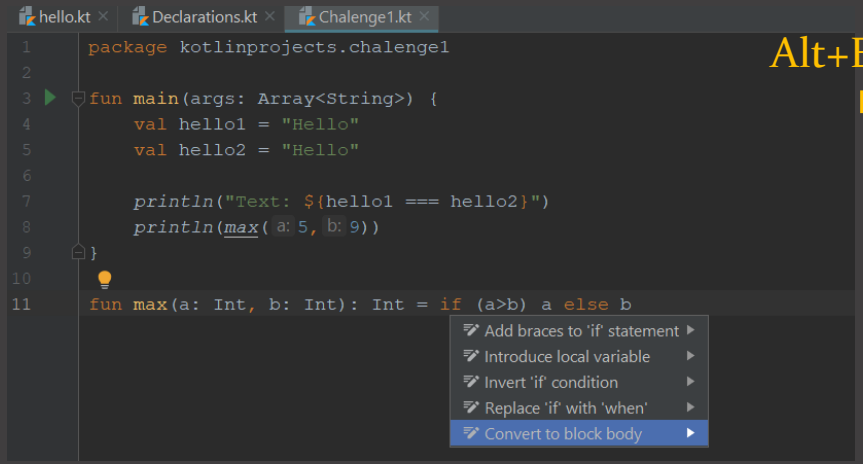
З іншого боку, оператор присвоєння є виразом у Java, але інструкцією в Kotlin

Тіла виразів

Функцію з попереднього слайду можна ще більше спростити

```
fun max(a: Int, b: Int): Int = if (a>b) a else b
```

Якщо тіло функції знаходиться в фігурних дужках, тоді функція має ***тіло-блок*** (block body).
Функція, що повертає вираз напряму, має ***тіло-вираз*** (expression body).



Alt+Enter

```
fun max(a: Int, b: Int): Int {
    return if (a>b) a else b
}
```

Спростимо ще за рахунок виведення типу (type inference) для функцій з тілом-виразом:

```
fun max(a: Int, b: Int) =
if (a>b) a else b
```

Змінні

Оскільки Kotlin дозволяє інколи опускати типи, вони записуються в кінці оголошення змінних

```
val question = "The Ultimate Question of life, the Universe, and Everything"
val answer = 42
```

За бажання тип можна додати:

```
val answer: Int = 42
```

- val – незмінюване посилання. Такій змінній неможливо присвоїти значення після ініціалізації. Відповідає фінальним змінним з Java. Рекомендується для оголошення переважної більшості змінних.
- var – змінюване посилання. Відповідає звичайній змінній з Java

Змінні. Ключові слова val і var

```
val message: String
if (canPerformOperation()) {
  message = "Success"
  // ... выполнить операцию
}
else {
  message = "Failed"
}
```

Оголошена з ключовим словом val змінна повинна ініціалізуватись лише 1 раз підчас виконання блоку, в якому була визначена.

Незважаючи на незмінюваність посилання, об'єкт, на який воно вказує, можна змінити:

```
val languages = arrayListOf("Java")
Languages.add("Kotlin")
```

Хоч ключове слово var дозволяє змінювати значення змінної, її тип фіксується

```
var answer = 42
answer = "no answer"        // помилка – розбіжність типів
```

Слід перетворювати тип вручну

Просте форматування рядків

Шаблони

```
fun main(args: Array<String>) {
  val name = if (args.size > 0) args[0] else "Kotlin"
  println("Hello, $name!")
}
```

Демонструються рядкові шаблони (string templates).
Запис із знаком $ рівносильний конкатенації рядків у Java `("Hello, " + name + "!")`
Щоб включити в текст $, його треба екранувати: `println("\$x")`
Для складних виразів використовують фігурні дужки:

```
fun main(args: Array<String>) {
  if (args.size > 0) {
    println("Hello, ${args[0]}!")
  }
}
```

Класи

Простий Java-клас Person

```java
/* Java */
public class Person {
  private final String name;

  public Person(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
}
```

```
class Person(val name: String)
```

Класи такого виду (лише дані) часто називають об'єктами-значеннями (value objects).

- Багато мов пропонують короткий синтаксис.

Після конвертації зник модифікатор public.

- Область видимості public у Kotlin за замовчуванням

Властивості

У мові Kotlin заміняють поля та методи доступу.
Оголошуються також за допомогою val (тільки для читання) і var.

```
class Person(
  val name: String,        // незмінювана властивість (поле+метод зчитування)
  var isMarried: Boolean   // змінювана властивість
)
```

```
/* Java */
>>> Person person = new Person("Bob", true);
>>> System.out.println(person.getName());
Bob
>>> System.out.println(person.isMarried());
true
```

```
>>> val person = Person("Bob", true)
>>> println(person.name)
Bob
>>> println(person.isMarried)
true
```

Конструктор викликається без new

Власні методи доступу

Нехай визначається клас прямокутників, який може повідомити, чи є така фігура квад ратом.

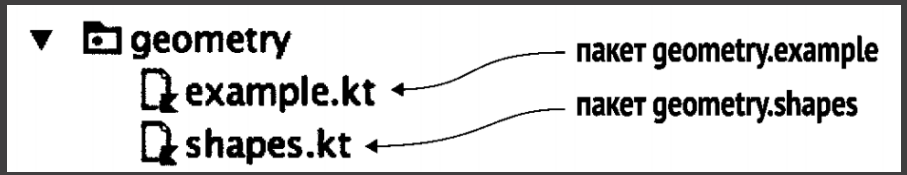
- Окреме поле не потрібно, оскільки рівність ширини та довжини можна перевірити динамічно
- Достатньо методу зчитування з особливою реалізацією
- Можна писати й коротше: get() = height == width

```
class Rectangle(val height: Int, val width: Int) {
  val isSquare: Boolean
    get() {                          <— Объявление метода чтения для свойства
      return height == width
  }
}
```

```
>>> val rectangle = Rectangle(41, 43)
>>> println(rectangle.isSquare)
false
```

Пакети і каталоги в Kotlin

```
package geometry.shapes      <— Объявление пакета

import java.util.Random      <— Импорт класса из стандартной библиотеки Java

class Rectangle(val height: Int, val width: Int) {
  val isSquare: Boolean
    get() = height == width
}

fun createRandomRectangle(): Rectangle {
  val random = Random()
  return Rectangle(random.nextInt(), random.nextInt())
}
```



Kotlin не розрізняє імпорт класів та функцій
Функції вищого рівня можна імпортувати за іменем

Імпорт із зіркою робить видимими не лише класи, а й властивості та функції вищого рів ня

```
package geometry.example

import geometry.shapes.createRandomRectangle

fun main(args: Array<String>) {
  println(createRandomRectangle().isSquare)
}
```

Представлення та обробка вибору

Представлення та обробка вибору

Перелічення

Kotlin **`enum`** – м'яке ключове слово (soft keyword).

- Має особливе значення тільки перед ключовим словом **`class`**:

```
enum class Color {
    RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET
}
```

Як і в Java, перелічення допускають оголошення властивостей та методів всередині

```
enum class Color(
  val r: Int, val g: Int, val b: Int
) {
  RED(255, 0, 0), ORANGE(255, 165, 0),
  YELLOW(255, 255, 0), GREEN(0, 255, 0), BLUE(0, 0, 255),
  INDIGO(75, 0, 130), VIOLET(238, 130, 238);

  fun rgb() = (r * 256 + g) * 256 + b

}
>>> println(Color.BLUE.rgb())
255
```

Значения свойств определяются для каждой константы

Объявление свойств констант перечисления

Точка с запятой здесь обязательна

Определение метода класса перечисления

Представлення та обробка вибору

Використання оператору when з класами перелічень

Оператор when – аналог Java-інструкції switch:

- При наявності збігу виконається тільки відповідна вітка

```
fun getMnemonic(color: Color) =
        when (color) {
            Color.RED -> "Каждый"
            Color.ORANGE -> "Охотник"
            Color.YELLOW -> "Желает"
            Color.GREEN -> "Знать"
            Color.BLUE -> "Где"
            Color.INDIGO -> "Сидит"
            Color.VIOLET -> "Фазан"
        }
>>> println(getMnemonic(Color.BLUE))
Где
```

Об'єднання варіантів в одну вітку when:

```
fun getWarmth(color: Color) = when(color) {
  Color.RED, Color.ORANGE, Color.YELLOW -> "теплый"
  Color.GREEN -> "нейтральный"
  Color.BLUE, Color.INDIGO, Color.VIOLET -> "холодный"
}

>>> println(getWarmth(Color.ORANGE))
теплый
```

Можна спростити код, імпортувавши значення констант:

```
import ch02.colors.Color
import ch02.colors.Color.*
fun getWarmth(color: Color) = when(color) {
  RED, ORANGE, YELLOW -> "теплый"
       GREEN -> "нейтральный"
       BLUE, INDIGO, VIOLET -> "холодный"
}
```

Представлення та обробка вибору

Використання оператору when з довільними об’єктами

На відміну від switch у мові Java, оператор when дозволяє використовувати довільні об’єкти у визначеннях варіантів (не тільки константи):

```
fun mix(c1: Color, c2: Color) =
  when (setOf(c1, c2)) {
    setOf(RED, YELLOW) -> ORANGE
    setOf(YELLOW, BLUE) -> GREEN
    setOf(BLUE, VIOLET) -> INDIGO
    else -> throw Exception("Грязный цвет")
  }

>>> println(mix(BLUE, YELLOW))
GREEN
```

Перечисление пар цветов, пригодных для смешивания

Аргументом выражения «when» может быть любой объект. Он проверяется условными выражениями ветвей

Выполняется, если не соответствует ни одной из ветвей

Стандартна бібліотека Kotlin включає функцію setoff(), яка створює множину Set з об’єктами, переданими в аргументах.
Оскільки порядок у множині неважливий, у першому порівнянні будуть умови, щоб c1==RED & c2==YELLOW або навпаки.

Представлення та обробка вибору

Вираз when без аргументів

Код стає менш читабельним, але продуктивнішим:

```
fun mixOptimized(c1: Color, c2: Color) =
    when {                                    <— Выражение «when» без аргумента
        (c1 == RED && c2 == YELLOW) ||
        (c1 == YELLOW && c2 == RED) ->
            ORANGE
        (c1 == YELLOW && c2 == BLUE) ||
        (c1 == BLUE && c2 == YELLOW) ->
            GREEN
        (c1 == BLUE && c2 == VIOLET) ||
        (c1 == VIOLET && c2 == BLUE) ->
            INDIGO
        else -> throw Exception("Dirty color")
    }
>>> println(mixOptimized(BLUE, YELLOW))
GREEN
```

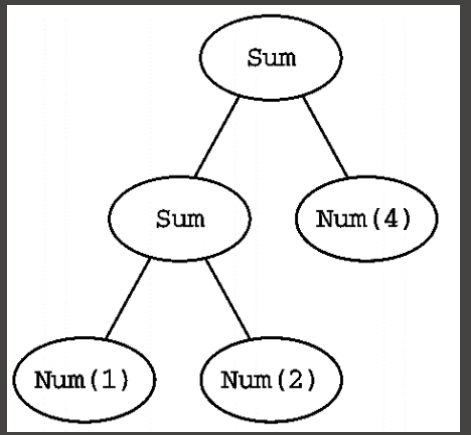
У виразах when без аргументу умовою вибору вітки може ста ти будь-який логічний вираз.

Представлення та обробка вибору

Автоматичне зведення типів

Демонстраційна задача: написати функцію, яка обчислює прості арифметичні вирази (тип операцій–додавання, наприклад (1+2)+4)

```
interface Expr
class Num(val value: Int): Expr
class Sum(val left:Expr, val right:Expr): Expr
```



Результатом має стати число 7:

```
>>> println (eval(Sum(Sum(Num(1), Num(2)), Num (4))))
7
```

Інтерфейс Expr має 2 реалізації:

- Якщо вираз є числом, повертається його значення
- Якщо це операція додавання, після обчислення лівої та правої частини виразу необхідно повернути їх суму

Представлення та обробка вибору

Автоматичне зведення типів

Спочатку напишемо Kotlin-метод у Java-стилі (послідовність операторів if):

```
fun eval(e: Expr): Int {
  if (e is Num) {
    val n = e as Num      <— Явное приведение к типу Num здесь излишне
    return n.value
  }
  if (e is Sum) {
    return eval(e.right) + eval(e.left)  <— Переменная e уже приведена к нужному типу!
  }
  throw IllegalArgumentException("Unknown expression")
}

>>> println(eval(Sum(Sum(Num(1), Num(2)), Num(4))))
7
```

Приналежність змінної певному типу перевіряється за допомогою оператора is (подібно до instansof у Java). Проте компілятор Kotlin також виконує ***автоматичне зведення типу*** (smart cast): після перевірки приналежності типу змінна відразу зводиться до нього.

Явне зведення до конкретного типу виражається ключовим словом as: **`val n = e as Num`**

Представлення та обробка вибору

Автоматичне зведення типів

Зведемо код до стилю Kotlin:

1) Використаємо переваги if як виразу (не потрібно return):

```
fun eval(e: Expr): Int =
  if (e is Num) {
    e.value
  } else if (e is Sum) {
    eval(e.right) + eval(e.left)
  } else {
    throw IllegalArgumentException("Unknown expression")
  }

>>> println(eval(Sum(Num(1), Num(2))))
3
```

2) Замінимо if на when

```
fun eval(e: Expr): Int =
  when (e) {
    is Num ->                    <— Ветка «when» проверяет тип аргумента
      e.value                    <— Используется автоматическое приведение типов
    is Sum ->                                  <— Ветка «when» проверяет тип аргумента
      eval(e.right) + eval(e.left)             <— Используется автоматическое приведение типов
    else ->
      throw IllegalArgumentException("Unknown expression")
  }
```

Представлення та обробка вибору

Блоки у виразах if та when

Результатом вітки стає результат останнього виразу в блоці:

```
fun evalWithLogging(e: Expr): Int =
  when (e) {
    is Num -> {
      println("num: ${e.value}")
      e.value
    }
    is Sum -> {
      val left = evalWithLogging(e.left)
      val right = evalWithLogging(e.right)
      println("sum: $left + $right")
      left + right
    }
    else -> throw IllegalArgumentException("Unknown expression")
  }
```

Это последнее выражение в блоке, функция вернет его значение, если e имеет тип Num

Функция вернет значение этого выражения, если e имеет тип Sum

```
>>> println(evalWithLogging(Sum(Sum(Num(1), Num(2)), Num(4))))
num: 1
num: 2
sum: 1 + 2
num: 4
sum: 3 + 4
7
```

Ітерації: цикли while і for

Цикл while

Синтаксис циклів while та do-while у мові Kotlin той же, що і в мові Java

```
while(condition) {
    /*…*/
}

do {
    /*…*/
} while(condition)
```

Ітерування по послідовності

Діапазони та прогресії

Стандартного для Java циклу for у Kotlin немає:

- for у Kotlin – аналог for-each Java
- ***Діапазони*** визначаються за допомогою оператору **..**: `val OneToTen = 1..10`
- Діапазони закриті та включні (для напівзакритих – оператор until: `x in 0 until size`)
- ***Прогресія*** – це діапазон, обхід якого можна повністю здійснити

Гра Fizz-Buzz: гравці домовляються про діапазон чисел, а потім виконують ходи, починаючи з 1.

- Вони заміняють числа, що кратні 3, на слово fizz, а числа, що кратні 5 – на buzz.
- Якщо число кратне і 3, і 5, - заміна на fizz buzz
- Візьмемо діапазон від 1 до 100

Реалізація гри Fizz-Buzz

```
fun fizzBuzz(i: Int) = when {
  i % 15 == 0 -> "FizzBuzz "
  i % 3 == 0 -> "Fizz "
  i % 5 == 0 -> "Buzz "
  else -> "$i "
}

>>> for (i in 1..100) {
...     print(fizzBuzz(i))
...   }

1 2 Fizz 4 Buzz Fizz 7 …
```

Ускладнимо правила гри: рахунок у зворотному порядку

```
>>> for (i in 100 downTo 1 step 2) {
  ... print(fizzBuzz(i))
  ... }
Buzz 98 Fizz 94 92 FizzBuzz 88 …
```

Ітерації по елементах словників

```
val binaryReps = TreeMap<Char, String>()
for (c in 'A'..'F') {
  val binary = Integer.toBinaryString(c.toInt())
  binaryReps[c] = binary
}

for ((letter, binary) in binaryReps) {
  println("$letter = $binary")
}
```

Словарь TreeMap хранит ключи в порядке сортировки

Обход диапазона символов от A до F

Преобразует ASCII-код в двоичное представление

Сохраняет в словаре значение с ключом в c

Обход элементов словаря; ключ и значение присваиваются двум переменным

Цикл for дозволяє розпаковувати елементи колекції.

Код

```
binaryReps[c] = binary
```

еквівалентний

```
binaryReps.put(c, binary)
```

Той же синтаксис дозволяє зберігати індекс поточного елементу колекції:

```
val list = arrayListOf("10", "11", "1001")
for ((index, element) in list.withIndex()) {
  println("$index: $element")
}
```

```
0: 10
1: 11
2: 1001
```

Використання in для перевірки входження в діапазон/колекцію

```
fun isLetter(c: Char) = c in 'a'..'z' || c in 'A'..'Z'
fun isNotDigit(c: Char) = c !in '0'..'9'

>>> println(isLetter('q'))
true
>>> println(isNotDigit('x'))
true
```

Оператори in та !in можуть працювати разом із 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…"
}

>>> println(recognize('8'))
It's a digit!
```

Діапазони не обмежуються символами, якщо клас підтримує порівняння екземплярів (реалізує java.lang.Comparable)

```
>>> println("Kotlin" in "Java".."Scala")
true
```

То же, что и "Java" <= "Kotlin" && "Kotlin" <= "Scala"

Така перевірка працює і з колекціями

```
>>> println("Kotlin" in setOf("Java", "Scala"))
false
```

Обробка виключень

Виконується аналогічно до Java

Збудження виключення (у Kotlin не використовується new):

```
if (percentage !in 0..100) {
  throw IllegalArgumentException(
    "A percentage value must be between 0 and 100: $percentage")
}
```

Конструкція throw у Kotlin теж є виразом:

```
val percentage =
  if (number in 0..100)
    number
  else
   throw IllegalArgumentException(
     "A percentage value must be between 0 and 100: $number")
```

Конструкція try-catch-finally

```
fun readNumber(reader: BufferedReader): Int? {
  try {
    val line = reader.readLine()
    return Integer.parseInt(line)
  }
  catch (e: NumberFormatException) {
    return null
  }
  finally {
    reader.close()
  }
}

>>> val reader = BufferedReader(StringReader("239"))
>>> println(readNumber(reader))
239
```

Найбільша відмінність від Java полягає у відсутності конструкції throws у сигнатурі функції.
Виключення IOException є контрольованим, тому в Java повинно обов'язково оброблятись

Спеціального синтаксису для конструкції try-with-resources не передбачено
Доступна реалізація у вигляді бібліотечної функції.

try як вираз

```
fun readNumber(reader: BufferedReader) {
    val number = try {
        Integer.parseInt(reader.readLine())
    } catch (e: NumberFormatException) {
        return
    }
    println(number)
}
```

Ключове слово try також є виразом.
На відміну від if, тіло виразу завжди повинно бути в фігурних дужках

```
>>> val reader = BufferedReader(StringReader("not a number"))
>>> readNumber(reader)
```

Ничего не выведет

```
fun readNumber(reader: BufferedReader) {
  val number = try {
    Integer.parseInt(reader.readLine())
  } catch (e: NumberFormatException) {
    null
  }
  println(number)
}
```

Якщо потрібно, щоб функція продовжила роботу після виходу з блоку catch, він має повернути значення

```
>>> val reader = BufferedReader(StringReader("not a number"))
>>> readNumber(reader)
null
```

Возбудит исключение, поэтому функция выведет «null»

Визначення та виклик функцій

Колекції в мові Kotlin

Представлені тими ж класами, що в Java, але мають ширші можливості
Створення колекцій:

множини
списки
словники

```
val set  = hashSetOf(1, 7, 53)
val list = arrayListOf(1, 7, 53)
val map  = hashMapOf(1 to “one”,
                     7 to “seven”,
                     53 to “fifty=three”)
```

```
>>> println(set.javaClass)
class java.util.HashSet

>>> println(list.javaClass)
class java.util.ArrayList

>>> println(map.javaClass)
class java.util.HashMap
```

Також можна знайти останній елемент у списку або максимальне значення в колекції чисел:

```
>>> val strings = listOf("first", "second", "fourteenth")

>>> println(strings.last())
fourteenth

>>> val numbers = setOf(1, 14, 2)

>>> println(numbers.max())
14
```

Спрощення виклику функцій

У Java-колекціях є вбудована реалізація методу toString(), яка не завжди зручна:

```
>>> val list = listOf(1, 2, 3)
>>> println(list)
[1, 2, 3]
```

```
fun <T> joinToString(
  collection: Collection<T>,
  separator: String,
  prefix: String,
  postfix: String
): String {

  val result = StringBuilder(prefix)

  for ((index, element) in collection.withIndex()) {
    if (index > 0) result.append(separator)
    result.append(element)
  }

  result.append(postfix)
  return result.toString()
}
```

Функція joinToString додає елементи з колекції в об'єкт StringBuilder, залишаючи між ними роздільник (префікс на початок та суфікс в кінець)

```
>>> val list = listOf(1, 2, 3)
>>> println(joinToString(list, "; ", "(", ")"))
(1; 2; 3)
```

Функції Kotlin підтримують іменовані аргументи, проте лише для Java 8+:

```
joinToString(collection, separator = " ",
                         prefix = " ", postfix = ".")
```

Значення параметрів за замовчуванням

Розповсюджена проблема Java-коду – надмірна кількість перевантажених методів у деяких класах
Імена параметрів та типи повторюються, а відсутність деяких параметрів у перевантажених версіях може вносити непорозуміння та неточності.
Перевантаження можна уникнути за рахунок вказівки значень параметрів за замовчуванням.

```
fun <T> joinToString(
  collection: Collection<T>,
  separator: String = ", ",
  prefix: String = "",
  postfix: String = ""
): String
```

```
>>> joinToString(list, ", ", "", "")
1, 2, 3
>>> joinToString(list)
1, 2, 3
>>> joinToString(list, "; ")
1; 2; 3
```

При використанні іменованих аргументів їх порядок може бути нестрогим.

```
>>> joinToString(list, suffix = ";", prefix = "# ")
# 1, 2, 3;
```

Значення параметрів за замовчуванням визначаються у викликаючій функції, а не в місці виклику.
Оскільки значень за замовчуванням у Java немає, при виклику Kotlin-функцій в Java-коді слід перед авати ці значення в якості параметрів.

Значення за замовчуванням і Java

Анотація @JvmOverloads спрощує виклик та вимагає від компілятора створити перевантажені Java-методи, опускаючи по одному параметру, починаючи з кінця.
Приклад: анотування функції joinToString() дозволить отримати наступні перевантажені версії:

```
/* Java */
String joinToString(Collection<T> collection, String separator,
  String prefix, String postfix);

String joinToString(Collection<T> collection, String separator,
  String prefix);

String joinToString(Collection<T> collection, String separator);

String joinToString(Collection<T> collection);
```

Усунення допоміжних статичних класів

Функції вищого рівня

Іноді операція працює з об’єктами двох різних рівноважливих класів.
Іноді є один основний об’єкт, проте ускладнювати його інтерфейс новими операціями небажано

З’являються багато класів, що не мають ні стану, ні методів екземплярів, а виступають контейнерами для великої кількості статичних методів.
Приклад – клас Collections у JDK та багато Util-класів.

У Kotlin такі класи непотрібні: використовують функції вищого рівня.
Вони залишаються членами пакету та їх все е потрібно імпортувати в інших пакетах для використання.

Усунення допоміжних статичних класів

Оголошення joinString() як функції вищого рівня

```
package strings
fun joinToString(...): String { ... }
```

Оскільки JVM працює тільки з кодом всередині класів, відповідний Java-код матиме вигляд:

```
/* Java */
package strings;
public class JoinKt {
  public static String joinToString(...) { ... }
}
```

Соответствует имени файла join.kt из листинга 3.3

Усі функції вищого рівня компілюються в статичні методи класу. Для зміни назви класу з функціям и вищого рівня використовують анотацію @JvmName

```
@file:JvmName("StringFunctions")

package strings

fun joinToString(...): String { ... }
```

Тепер функцію можна викликати так:

```
/* Java */
import strings.StringFunctions;
StringFunctions.joinToString(list, ", ", "", "");
```

Усунення допоміжних статичних класів

Властивості вищого рівня

```
var opCount = 0

fun performOperation() {
    opCount++
    // ...
}
fun reportOperationCount() {
    println("Operation performed $opCount times")
}
```

Зберігання окремих фрагментів даних за межами класу потрібно не часто, проте буває корисним.
Наприклад, var-властивість для підрахування виконаних операцій.

Значення такої властивості зберігатиметься в статичному полі

Також властивості вищого рівня дозволяють визначати в коді константи

```
val UNIX_LINE_SEPARATOR = "\n"
```

Якщо потрібно зробити константу доступною з Java-коду як поле **public static final**, достатньо поставити модифікатор const (для простих типів та String):

```
const val UNIX_LINE_SEPARATOR = "\n"
```

Еквівалентний Java-код:

```
public static final String UNIX_LINE_SEPARATOR = "\n";
```

Допоміжні функції як розширення

```
fun <T> Collection<T>.joinToString(
    separator: String = ", ",
    prefix: String = "",
    postfix: String = ""
): String {
  val result = StringBuilder(prefix)

  for ((index, element) in this.withIndex())
    if (index > 0) result.append(separator)
    result.append(element)
  }

  result.append(postfix)
  return result.toString()
}
```

Значения по умолча
для параметров

```
>>> val list = listOf(1, 2, 3)
>>> println(list.joinToString(separator = "; ",
... prefix = "(", postfix = ")"))
(1; 2; 3)
```

Створено розширення для колекції елементів та визначено значення за замовчуванням для всіх аргументів.

Тепер joinToString() можна викликати як звичайний член класу:

```
>>> val list = arrayListOf(1, 2, 3)
>>> println(list.joinToString(" "))
1 2 3
```

Функції-розширення є синтаксичним цукром, тому для них потрібний конкретніший тип – тут колекція рядків:

```
>>> println(listOf("one", "two", "eight").join(" "))
one two eight
>>> listOf(1, 2, 8).join()
Error: Type mismatch: inferred type is List<Int> but Collection<String> was expected.
```

```
fun Collection<String>.join(
  separator: String = ", ",
  prefix: String = "",
  postfix: String = ""
) = joinToString(separator, prefix, postfix)
```

Функції-розширення не переозначаються

У Kotlin допускається переозначати функції-члени, проте неможливо переозначити функцію-розширення.

Нехай маємо 2 класи: View та його підклас Button (переозначає функцію click() суперкласу)

```
open class View {
  open fun click() = println("View clicked")
}

class Button: View() {
  override fun click() = println("Button clicked")
}
```

```
>>> val view: View = Button()
>>> view.click()
Button clicked
```

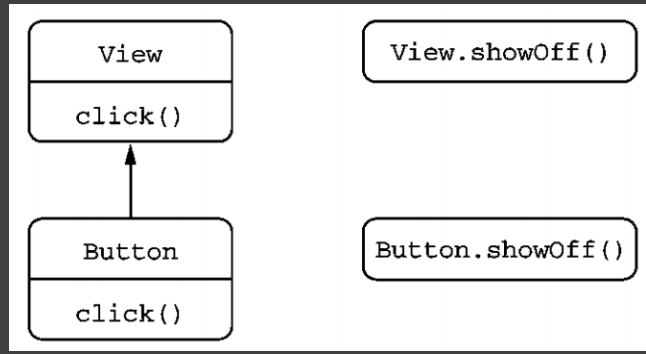
Для функцій-розширень це не працює

Функції-розширення оголошуються за межами класу та не є його частиною.

Приклад демонструє 2 функції-розширення showOff(), оголошені для класів View та Button

```
fun View.showOff() = println("I'm a view!")
fun Button.showOff() = println("I'm a button!")
```

```
>>> val view: View = Button()
>>> view.showOff()
I'm a view!
```



Тип змінної – View, тому showOff() – відповідний.

Якщо клас має метод з такою ж сигнатурою, як у функції-розширення, перевага віддається методу.
При розширенні API класів додавання такого методу змінить семантику коду.

Властивості-розширення

Дозволяють додавати в класи функції, до яких можна звернутись, як до властивостей

Властивості-розширення НЕ мають стану: неможливо додати додаткові поля в існуючі екземпляри об'єктів Java.

```
val String.lastChar: Char
    get() = get(length-1)
```

Аналогічно заборонені ініціалізатори.
Не маючи стану, повинен завжди оголошуватись метод для зчитування.

Для класу StringBuilder властивість-розширення буде var.
Для звернення до властивості-розширення з Java потрібно явно викликати її метод зчитування:
`StringUtilKt.getLastChar("Java")`

```
var StringBuilder.lastChar: Char
  get() = get(length - 1)
  set(value: Char) {
    this.setCharAt(length - 1, value)
  }
```

```
>>> println("Kotlin".lastChar)
n
>>> val sb = StringBuilder("Kotlin?")
>>> sb.lastChar = '!'
>>> println(sb)
Kotlin!
```

Робота з колекціями

Розширення API колекцій з Java

Колекції в Kotlin підтримують більше операцій, хоча є просто екземплярами Java-класів, за рахунок функцій-розширень.

Приклади (спрощені):

```
fun <T> List<T>.last(): T {/*повертає останній елемент*/}
fun Collection<Int>.max(): Int {/*шукає максимальне значення в колекції*/}
```

Функції, що приймають довільну кількість аргументів

Викликаючи функцію створення списку, їй можна передати довільну кількість аргументів:

```
val list = listOf(2, 3, 5, 7, 11)
```

Сигнатура бібліотечної функції:

```
fun listOf<T>(vararg values: T): List<T> {…}
```

Синтаксис виклику функцій у Kotlin та Java також відрізняється способом передачі упакова них в масив аргументів:

- Java: масив передається безпосередньо
- Kotlin: вимагає явно розпакувати масив, щоб кожен елемент став окремим аргументом функції, що викликається. Використовується оператор розпаковки (spread operator) *

```
fun main(args: Array<String>) {
  val list = listOf("args: ", *args)
  println(list)
}
```

Робота з парами: інфіксні виклики та мультидекларації

Для створення словника застосовується функція mapOf:

```
val map = mapOf(1 to “one”, 7 to “seven”, 53 to “fifty-three”)
```

Ключове слово to – спеціальна форма (інфіксного) виклику методу.
Еквівалентні виклики: `1.to(“one”)` та `1 to “one”`

```
fun listOf<T>(vararg values: T): List<T> {…}
```

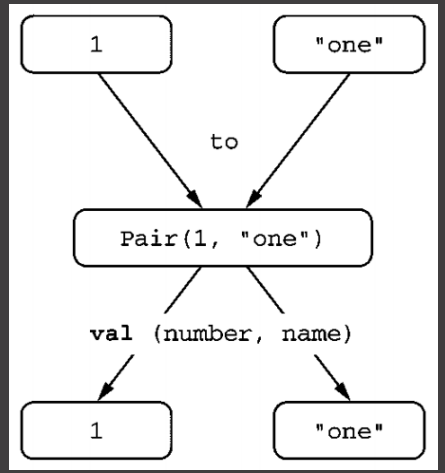
Для застосування інфіксної форми виклику до сигнатури дописують модифікатор infix. Спрощений приклад:

```
infix fun Any.to(other: Any) = Pair(this, other)
```

Значеннями об’єкта Pair можна ініціалізувати відразу 2 змінні:

```
val (number, name) = 1 to “one”
```

Це приклад ***мультидекларації*** (destructing declaration)



Функція to є функцією-розширенням, функція розширяє тип-приймач:

```
1 to “one”, “one” to 1, list to list.size()
```

і т. д.

Робота з рядками

Рядки в Kotlin – це ті ж об’єкти, що і в Java.
Проте з ними приємніше працювати.

Додаються багато корисних функцій-розширень.
Також приховуються деякі методи, що викликали плутанину в Java.

Приклад 1. Розбиття рядків.

- Java: метод split(). Працює з регулярними виразами, тому сприймає деякі символи (‘.’ , ‘?’) не як сим воли пунктуації.
- Kotlin: пропонує кілька перевантажених реалізацій (для регулярних виразів аргумент типу Regex, а не String), які однозначно інтерпретують рядок.

Явна передача регулярного виразу (синтаксис регулярних виразів аналогічний Java):

```
>>> println("12.345-6.A".split("\\.|-".toRegex()))
[12, 345, 6, A]
```

Інша версія split() приймає довільну кількість роздільників у вигляді рядків:

```
>>> println("12.345-6.A".split(".", "-"))
[12, 345, 6, A]
```

Регулярні вирази та рядки в потрій них лапках

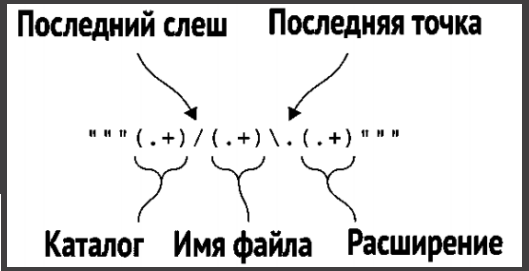
Задача: розбити повний шлях до файлу на компоненти: каталог, назва файлу, розширення.

Стандартна бібліотека Kotlin має функції для отримання підряд ка до першої (чи після останньої) появи заданого роздільника.


```
fun parsePath(path: String) {
  val directory = path.substringBeforeLast("/")
  val fullName = path.substringAfterLast("/")

  val fileName = fullName.substringBeforeLast(".")
  val extension = fullName.substringAfterLast(".")

  println("Dir: $directory, name: $fileName, ext: $extension")
}
>>> parsePath("/Users/yole/kotlin-book/chapter.adoc")
Dir: /Users/yole/kotlin-book, name: chapter, ext: adoc
```

Регулярні вирази та рядки в потрій них лапках

Розв'язок попередньої задачі за допомогою регулярних виразів.
При потрійних лапках не потрібно екранувати символи.
Також вони дозволяють створювати багаторядкові літерали.




```
fun parsePath(path: String) {
  val regex = """(.+)/(.+)\.(.+)""".toRegex()
  val matchResult = regex.matchEntire(path)
  if (matchResult != null) {
    val (directory, filename, extension) = matchResult.destructured
    println("Dir: $directory, name: $filename, ext: $extension")
  }
}
```

Для кращого форматування коду можна використовувати відступи, проте в парі з методом trimMargin().

```
val kotlinLogo = """| //
                    .|//
                    .|/ \"""
>>> println(kotlinLogo.trimMargin("."))
| //
|//
|/ \
```

Чистимо код

У багатьох випадках крупні методи розбиваються на менші фрагменти (операція Extract Method) з метою повторного використання.
Проте ускладнюється розуміння: багато незв'язаних між собою методів.
Можна згрупувати отримані методи у внутрішньому класі, проте буде потрібн о багато шаблонного коду.

Вихід Kotlin: виокремлені функції можна зробити вкладеними.
Функція в лістингу зберігає інформацію в БД та перевіряє, що об'єкт, який предс тавляє користувача, містить допустимі дані.

```
class User(val id: Int, val name: String, val address: String)

fun saveUser(user: User) {
  if (user.name.isEmpty()) {
    throw IllegalArgumentException(
      "Can't save user ${user.id}: empty Name")
  }

  if (user.address.isEmpty()) {
    throw IllegalArgumentException(
      "Can't save user ${user.id}: empty Address")
  }

  // Сохранение информации о пользователе в базе данных
}

>>> saveUser(User(1, "", ""))
java.lang.IllegalArgumentException: Can't save user 1: empty Name
```

Дублируется проверка

Виділення локальної функції для усунення дублювання коду

```
class User(val id: Int, val name: String, val address: String)

fun saveUser(user: User) {
  fun validate(user: User,
               value: String,
               fieldName: String) {
    if (value.isEmpty()) {
      throw IllegalArgumentException(
        "Can't save user ${user.id}: empty $fieldName")
    }
  }

  validate(user, user.name, "Name")
  validate(user, user.address, "Address")

  // Сохранение информации о пользователе в базе данных
}
```

Объявление локальной функции для проверки произвольного поля

Вызов функции для проверки конкретных полей

Логіка перевірки не дублюється.

Оскільки функція локальна, передавати змінну user не обов'язково.

```
class User(val id: Int, val name: String, val address: String)

fun saveUser(user: User) {
  fun validate(value: String, fieldName: String) {
    if (value.isEmpty()) {
      throw IllegalArgumentException(
        "Can't save user ${user.id}: " +
        "empty $fieldName")
    }
  }

  validate(user.name, "Name")
  validate(user.address, "Address")

  // Сохранение информации о пользователе в базе данных
}
```

Теперь не нужно дублировать параметра user в функции saveUser

Можно напрямую обращаться к параметрам внешней функции

Подальше покращення

Перенесемо логіку перевірки у функцію-розширення класу User.

```
class User(val id: Int, val name: String, val address: String)

fun User.validateBeforeSave() {
  fun validate(value: String, fieldName: String) {
    if (value.isEmpty()) {
      throw IllegalArgumentException(
        "Can't save user $id: empty $fieldName")
    }
  }
  validate(name, "Name")
  validate(address, "Address")
}

fun saveUser(user: User) {
  user.validateBeforeSave()

  // Сохранение пользователя в базу данных
}
```

К свойствам класса User можно обращаться напрямую

Вызов функции-расширения

Об’єктно-орієнтоване програмування засобами Kotlin

Класи, об’єкти, інтерфейси

Класи та інтерфейси в Kotlin дещо відрізняються від Java.

Kotlin-інтерфейси можуть містити оголошення властивостей.
За замовчуванням оголошення отримують модифікатори public final.
Вкладені класи за замовчуванням не стають внутрішніми, оскільки не містять явного посилання на зовнішній клас.

Компілятор Kotlin сам може генерувати корисні методи.

Якщо клас оголошується ***класом даних (data class)***, компілятор додасть у нього кілька стандартних методів.
Також немає потреби делегувати методи вручну: шаблон делегування підтримується на рівні мови.

Створення ієрархій класів

Інтерфейси в Kotlin

Інтерфейси в Kotlin нагадують Java 8: можуть містити визначення абстрактних та реалізації конкретних методів, проте не містять стани.

```
interface Clickable {
    fun click()
}

Class Button: Clickable {
    override fun click() = println("I was clicked")
}
```

```
>>> Button().click()
I was clicked
```

На відміну від Java, застосування модифікатора `override` в Kotlin ***обов'язкове***.
Уникається випадкове переозначення методів.

Для методів за замовчуванням у Kotlin не передбачено ключових слів.

```
interface Clickable {
    fun click()
    fun showOff() = println("I'm clickable!")
}
```

Інтерфейси в Kotlin

Нехай інший інтерфейс оголошує метод showOff() зі своєю реалізацією.

```
interface Focusable {
    fun setFocus(b: Boolean) =
        println("I ${if (b) "got" else "lost"} focus.")
    fun showOff() = println("I'm focusable!")
}
```

Яка реалізація буде обрана?

Жодна з перелічених: компілятор змусить писати власну.

```
The class 'Button' must
override public open fun showOff() because it inherits
many implementations of it.
```

```
class Button: Clickable, Focusable {
    override fun click() = println("I was clicked")

    override fun showOff() {
        super<Clickable>.showOff()
        super<Focusable>.showOff()
    }
}
```

```
fun main(args: Array<String>) {
    val button = Button()
    button.showOff()          <- I'm clickable!
                                 I'm focusable!
    button.setFocus(true)     <- I got focus.
    button.click()            <- I was clicked.
}
```

Модифікатори open, final, abstract

Проблема крихкого базового класу: вплив змін базового класу на підкласи.
Якщо не прописані точні правила наслідування (які методи та як переозначаються), може виявитись не передбачена автором базового класу поведінка.

Джошуа Блох рекомендує «проектувати і документувати наслідування або забороняти його»
Тобто всі класи та методи, що не передбачені для переозначення, повинні бути final.
Для дозволу на переозначення в Kotlin використовують модифікатор open.

```
open class RichButton: Clickable {
    fun disable() {}          // закрита функція: переозначення неможливе
    open fun animate() {}     // відкрита функція: доступне переозначення
    override fun click() {}   // переозначення відкритої функції є відкритим
}
```

Переозначена версія методу завжди відкрита.
Для заборони подальшого переозначення в підкласах додають модифікатор final.

```
open class RichButton : Clickable {
    final override fun click() {}
}
```

Відкриті класи та розумне зведення типів

Одна з основних переваг закритих за замовчуванням класів – автоматичне зведення типів у багатьох сценаріях.

Властивості класу мають бути final, інакше підклас зможе переозначити властивість та власний метод доступу, порушуючи основну вимогу автоматичного зведення типів.

Клас можна оголосити абстрактним.

Абстрактні методи завжди відкриті.

```
abstract class Animated {
    abstract fun animate()
    open fun stopAnimating() {}    // конкретні функції в абстрактних класах
    fun animateTwice() {}          // за замовчуванням закриті
}
```

Узагальнена таблиця модифікаторів

Модифікатор	Відповідний член	Коментарі
final	Не може переозначатись	Застосовується до членів класу за замовчуванням
open	Може переозначатись	Повинен вказуватись явно
abstract	Повинен переозначатись	Використовується лише в абстрактних класах; абстрактні методи не можуть мати реалізацію
override	Переозначає метод супер класу або інтерфейсу	За замовчуванням переозначений метод відкритий, якщо не оголошений як final

Модифікатори видимості: за замовчу ванням public

Контролюють доступні об’єктів у коді.

Обмеження видимості деталей реалізації може гарантувати можливість їх зміни без ризику поломки коду, що залежить від класу.

Використовуються ключові слова public, protected, private.

Видимість за замовчуванням: public.
Модифікатор private доступний для оголошень вищого рівня – такі оголошення видно лише в межах файлу, де вони визначені.

Видимість на рівні пакету в Kotlin відсутня.

Альтернатива – видимість на рівні модуля (набору файлів, що разом компілюються).
Модифікатор internal дозволяє справжню інкапсуляцію.
В Java автор стороннього коду отримає доступ до оголошень з області видимості пакету, якщо визначить класи в тому ж пакеті

Модификатор	Член класса	Объявление верхнего уровня
public (по умолчанию)	Доступен повсюду	Доступно повсюду
internal	Доступен только в модуле	Доступно в модуле
protected	Доступен в подклассах	–
private	Доступен в классе	Доступно в файле

Приклад

Кожен рядок у функції giveSpeech() при виконанні порушувала б правила видимості.
Компілюється з помилкою.

```
internal open class TalkativeButton : Focusable {
    private fun yell() = println("Hey!")
    protected fun whisper() = println("Let's talk!")
}
fun TalkativeButton.giveSpeech() {
    yell()
    whisper()
}
```

Ошибка: «публичный» член класса раскрывает «внутренний» тип-приемник «TalkativeButton»

Ошибка: функция «yell» недоступна; в классе «TalkativeButton» она объявлена с модификатором «private»

Ошибка: функция «whisper» недоступна; в классе «TalkativeButton» она объявлена с модификатором «protected»

Kotlin забороняє посилатись із публічної функції giveSpeech() на тип TalkativeButton з вужчою областю видимості (тут – internal).

Це частинний випадок правила: всі класи в списку базових та параметризованих типів класу або в сигнатурах методів повинні мати таку ж або ширшу область видимості, ніж клас або метод.
Для виправлення помилки треба оголосити функцію як internal або зробити клас публічним.

Відмінності модифікаторів у Java і Kotlin

Java: член класу доступний у всьому пакеті.
Kotlin: член класу доступний тільки в класі та його підкласах.
Функції-розширення класу не можуть звертатись до його членів з модифікаторами protected або private.

При компіляції в байт-код Java модифікатори public, protected, private зберігаються.
Виключення: клас з модифікатором private компілюється з областю видимості пакету.
Модифікатор internal у байт-коді перетворюється в public.

Внутрішні та вкладені класи

Вкладені класи не мають доступу до екземпляру зовнішнього класу, якщо явно не ви конати запит.

Нехай потрібно визначити видимий елемент (View), стан якого може бути серіалізовано.
Часто це складно, проте можна скопіювати необхідні дані в допоміжний клас.

```
interface State: Serializable

interface View {
    fun getCurrentState(): State
    fun restoreState(state: State) {}
}
```

Проблемний код

Зручно визначити клас, який зберігатиме стан кнопки в класі Button.

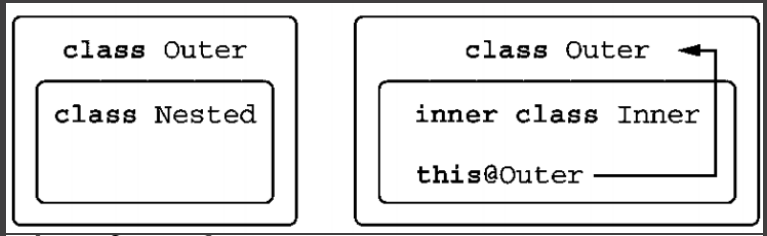
```
/* Java */
public class Button implements View {
    @Override
    public State getCurrentState() {
      return new ButtonState();
    }

    @Override
    public void restoreState(State state) { /*...*/ }

    public class ButtonState implements State { /*...*/ }
}
```

При спробі серіалізації стану кнопки виникає виключення java.io.NotSerializableException:Button.
У мові Java при оголошенні одного класу всередині іншого він автоматично стає внутрішнім – тут неявно міститиме посилання на зовнішній клас Button.
Клас Button не серіалізується, а посилання на нього в ButtonState не дають серіалізуватись і йому.

Виправлення коду

У Java достатньо зробити клас ButtonState статичним (неявне посилання видалиться).
У Kotlin аналогом є вкладений клас без модифікаторів.
Для перетворення вкладеного класу у внутрішній використовують модифікатор inner.

```
class Button : View {
    override fun getCurrentState(): State = ButtonState()

    override fun restoreState(state: State) { /*...*/ }

    class ButtonState : State { /*...*/ }
}
```




```
class Outer {
    inner class Inner {
      fun getOuterReference(): Outer = this@Outer
    }
}
```

Клас A, оголошений всередині іншого класу B	У Java	У Kotlin
Вкладений клас	static class A	class A
Внутрішній клас	class A	inner class A

Запечатані класи

Визначення жорстко заданих ієрархій

Згадаємо задачу про знаходження значення виразів

```
interface Expr
class Num(val value: Int): Expr
class Sum(val left:Expr, val right:Expr): Expr

fun eval(e: Expr): Int =
    when(e) {
        is Num -> e.value
        is Sum -> eval(e.right) + eval(e.left)
        else -> throw IllegalArgumentException("Unknown expression")
    }
```

При обчисленні виразу за допомогою when компілятор змушує додати вітку, що виконується за замовчуванням.

Тут – генерується виключення.

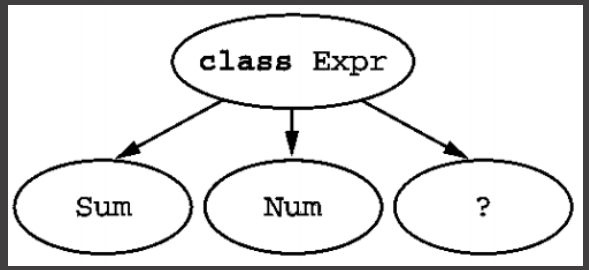
Якщо забути додати нову вітку в код, будуть генеруватись складні для налагодження помилки.

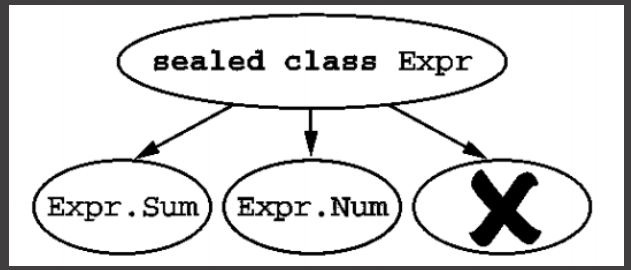
Запечатані класи

Проблема вирішується за допомогою ***запечатаних*** (sealed) класів: усі їх прямі підкласи повинні бути вкладеними в суперклас.

Запечатаний клас автоматично є відкритим.

```
sealed class Expr {
    class Num(val value: Int): Expr
    class Sum(val left:Expr, val right:Expr): Expr
}
fun eval(e: Expr): Int =
    when(e) {
        is Num -> e.value
        is Sum -> eval(e.right) + eval(e.left)
    }
```





Тут вираз when охоплює всі можливі варіанти.

При додаванні нового підкласу вираз when не скомпілюється, оскільки клас запечатаний.

Всередині клас Expr отримає приватний конструктор, який можна викликати лише всередині класу.

Запечатаний інтерфейс – неможливий, оскільки його можна було б реалізувати в Java-коді.

Класи з нетривіальними конструкторами або властивостями

Оголошення стандартного класу: `class User(val nickname: String)`
Блок коду у круглих дужках – це основний конструктор (primary constructor).
Інший запис коду:

```
class User constructor(_nickname: String) { // основний конструктор
    val nickname: String
    init {                                  // блок ініціалізації
        nickname = _nickname
    }
}
```

Блок ініціалізації містить код, що виконується при створенні кожного екземпляру класу.
Призначені для використання разом з первинними конструкторами.
Можна оголошувати кілька блоків ініціалізації.
Не маючи анотацій та модифікаторів видимості, код можна спростити до вигляду:

```
class User(_nickname: String) { // основний конструктор
    val nickname = _nickname
}
```

Класи з нетривіальними конструктор ами або властивостями

Параметрам конструктора та функцій можна присвоювати значення за замовчуванням:

```
class User(val nickname: String, val isSubscribed: Boolean = true)
```

Створення екземплярів класу:

```
>>> val alice = User("Alice")
>>> println(alice.isSubscribed)
true
>>> val bob = User("Bob", false)
>>> println(bob.isSubscribed)
false
>>> val carol = User("Carol", isSubscribed = false)
>>> println(carol.isSubscribed)
false
```

Конструктор дочірнього класу

Основний конструктор дочірнього класу також ініціалізує успадковані властивості:

```
open class User(val nickname: String) {…}
class TwitterUser(nickname: String): User(nickname) {…}
```

Якщо взагалі не оголосити конструктор, компілятор додасть конструктор за замовчу ванням: `open class Button`

Для його успадкування треба явно викликати конструктор суперкласу:

```
class RadioButton: Button()
```

Інтерфейси не мають конструктора, тому не вимагають круглих дужок при реалізації

Якщо хочете, щоб ніякий інший код не міг створювати екземпляри класу, зробіть конструктор приватним:

```
class Secretive private constructor() {}
```

Вторинні конструктори

Різні способи ініціалізації суперкласу

Часто потрібно кілька конструкторів для ініціалізації класу різними способами.

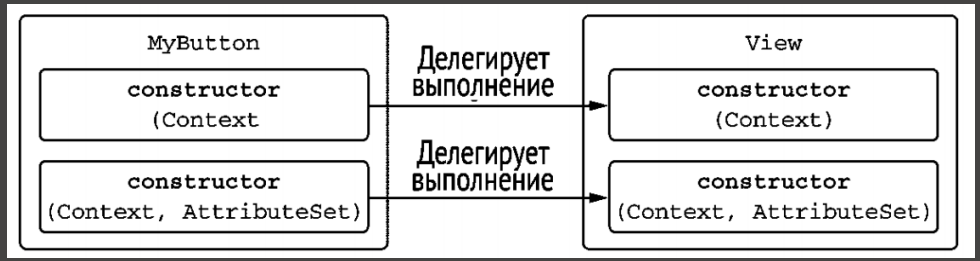
Нехай є Java-клас View з 2 конструкторами. У Kotlin аналогічне оголошення:

```
open class View {
    constructor(ctx: Context) {/*певний код*/}
    constructor(ctx: Context, attr: AttributeSet) {/*певний код*/}
}
```

Основного конструктора немає (відсутні дужки), проте є 2 вторинних.

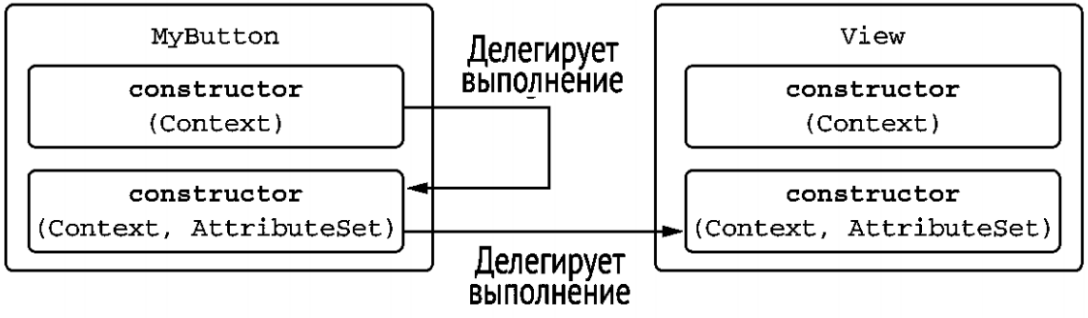
Для розширення класу оголошуються ті ж конструктори:

```
class MyButton: View {
    constructor(ctx: Context):super(ctx) {/*певний код*/}
    constructor(ctx: Context, attr: AttributeSet): super(ctx, attr) {
        /*певний код*/
    }
}
```



Ключове слово this()

```
class MyButton : View {
    constructor(ctx: Context): this(ctx, MY_STYLE) {
      // ...
    }

    constructor(ctx: Context, attr: AttributeSet): super(ctx, attr) {
      // ...
    }
}
```



Реалізація оголошених в інтерфейсах властивостей

Інтерфейси в Kotlin можуть містити оголошення абстрактних властивостей:

Оскільки інтерфейс не має стану, зберігати значення зможуть лише класи, які його реалізують.

```
interface User {
    val nickname: String
}
```

```
class PrivateUser(override val nickname: String) : User

class SubscribingUser(val email: String) : User {
    override val nickname: String
      get() = email.substringBefore('@') )
    }

class FacebookUser(val accountId: Int) : User {
    override val nickname = getFacebookName(accountId)
}
```

Свойство основного конструктора

Собственный метод чтения

Инициализация свойства

```
>>> println(PrivateUser("test@kotlinlang.org").nickname)
test@kotlinlang.org
>>> println(SubscribingUser("test@kotlinlang.org").nickname)
test
```

Різні реалізації властивості nickname

SubscribingUser: визначено власний метод зчитування, що повертає substringBefore
FacebookUser: для властивості передбачено поле, що зберігає обчислене підчас ініціа лізації класу значення.

Також інтерфейс може містити властивості з методами зчитування/запису за умови відсутності звертань до поля в пам'яті:

```
interface User {
    val email: String                // абстрактна властивість
    val nickname: String             // властивість з методом доступу
        get() = email.substringBefore('@')
}
```

Звернення до поля з методів доступу

Нехай потрібно організувати журналювання будь-яких змін даних, що зберігаються у властивості.

Оголосимо змінюване поле та додатковий код при кожному зверненні до нього.

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

Изменение значения поля

```
>>> val user = User("Alice")
>>> user.address = "Elsenheimerstrasse 47, 80687 Muenchen"
Address was changed for Alice:
"unspecified" -> "Elsenheimerstrasse 47, 80687 Muenchen".
```

Зверніть увагу: для змінюваної властивості можна переозначити тільки 1 з методів доступу.

Зміна видимості методів доступу

Можна додавати модифікатор доступу перед ключовими словами get і set.
Оголосимо змінюване поле та додатковий код при кожному зверненні до нього.

Клас обчислює загальну довжину слів, що в нього додаються

```
>>> val lengthCounter = LengthCounter()
>>> lengthCounter.addWord("Hi!")
>>> println(lengthCounter.counter)
3
```

```
class LengthCounter {
    var counter: Int = 0
      private set

    fun addWord(word: String) {
      counter += word.length
    }
}
```

Методи, згенеровані компілятором

Класи даних та делегування

Як і в Java, всі класи в Kotlin мають обов'язкові для переозначення методи equals(), hashCode(), toString().

Приклад: клас Client, який зберігає ім'я клієнта та поштовий індекс.

```
class Client(val name: String, val postalCode: Int)
```

За замовчуванням рядкове представлення неінформативне – переозначується метод toString().

```
class Client(val name: String, val postalCode: Int) {
    override fun toString() = "Client(name=$name, postalCode=$postalCode)"
}
```

```
>>> val client1 = Client("Alice", 342562)
>>> println(client1)
Client(name=Alice, postalCode=342562)
```

Метод equals()

Клас Client просто зберігає дані, проте може мати вимоги до поведінки.
Приклад: об’єкти вважаються рівними, якщо містять однакові дані.

```
>>> val client1 = Client("Alice", 342562)
>>> val client2 = Client("Alice", 342562)
>>> println(client1 == client2)
false
```

Java: оператор == порівнює прості типи за значенням, а посилкові типи – за посила нням. Поширена практика – використовувати метод equals(). Проблема – про нього постійно забувають.

Kotlin: оператор == викликає за кулісами метод equals(). Для порівняння посилань використовується оператор ===.

Реалізація методу equals() у класі Client

```
class Client(val name: String, val postalCode: Int) {
    override fun equals(other: Any?): Boolean {
      if (other == null || other !is Client)
        return false
      return name == other.name &&
        postalCode == other.postalCode
    }
    override fun toString() = "Client(name=$name, postalCode=$postalCode)"
}
```

«Any» – это аналог java.lang.Object: суперкласс всех классов в Kotlin. Знак вопроса в «Any?» означает, что аргумент «other» может иметь значение null

Убедиться, что «other» имеет тип Client

Вернуть результат сравнения свойств

Після переозначення equals() можна очікувати, що екземпляри з однаковими значеннями будуть рівними.

Проте для більш складних операцій також потрібно переозначити метод hashCode().

```
>>> val processed = hashSetOf(Client("Alice", 342562))
>>> println(processed.contains(Client("Alice", 342562)))
false
```

Причина – відсутність методу hashCode() у класі Client

Реалізація методу hashCode() у класі Client

Без hashCode() порушується контракт: рівні об'єкти повинні мати однакові хеш-коди.
При порівнянні значень у HashSet спочатку порівнюються хеш-коди, а потім – фактичні значення.

```
class Client(val name: String, val postalCode: Int) {
    ...
    override fun hashCode(): Int = name.hashCode() * 31 + postalCode
}
```

Зважаючи на велику кількість коду загалом, компілятор Kotlin створює такі методи автоматично.
Відповідні класи називають ***класами даних***.

```
data class Client(val name: String, val postalCode: Int)
```

Рекомендується працювати саме з незмінюваними властивостями.
У випадку HashMap ключі не можуть змінюватись.
Спрощується для розуміння багатопоточний код.
Для класів даних Kotlin також генерує метод для копіювання екземплярів.

Копіювання екземплярів у класах даних

Створення копії – хороша альтернатива модифікації екземпляру на місці: копія не впливає на код, що посилається на початковий екземпляр.

Вигляд та використання методу для копіювання:

```
class Client(val name: String, val postalCode: Int) {
    ...
    fun copy(name: String = this.name,
             postalCode: Int = this.postalCode) =
        Client(name, postalCode)
}
```

```
>>> val bob = Client("Bob", 973293)
>>> println(bob.copy(postalCode = 382555))
Client(name=Bob, postalCode=382555)
```

Делегування в класах. Ключове слово by

Поширена проблема в ОО проектуванні – нестабільність, що пов'язана з наслідуванням реалізації.

Код стає залежним від деталей реалізації базового класу.

У дизайні мови Kotlin всі класи за замовчуванням є фінальними, що гарантує наслідування лише тих класів, для яких така можливість передбачена.

Може бути потрібно додати поведінку в інший клас, що не призначений для наслідування.

Застосовується шаблон Декоратор.

Створюється новий клас з тим же інтерфейсом та екземпляром оригінального класу всередині.

Методи з незмінюваною поведінкою просто передають виклики оригінальному екземпляру класу.

Недолік – багато шаблонного коду.

Шаблонний код інтерфейсу Collection

```
class DelegatingCollection<T> : Collection<T> {
    private val innerList = arrayListOf<T>()

    override val size: Int get() = innerList.size
    override fun isEmpty(): Boolean = innerList.isEmpty()
    override fun contains(element: T): Boolean = innerList.contains(element)
    override fun iterator(): Iterator<T> = innerList.iterator()
    override fun containsAll(elements: Collection<T>): Boolean =
        innerList.containsAll(elements)
}
```

У Kotlin писати стільки коду не потрібно.

Можна ***делегувати*** реалізацію іншому об'єкту за допомогою ключового слова by.

```
class DelegatingCollection<T>(
    innerList: Collection<T> = ArrayList<T>()
) : Collection<T> by innerList {}
```

Використання делегування

Реалізуємо колекцію, яка підраховує кількість спроб додавання елементів у неї

```
class CountingSet<T>(
    val innerSet: MutableCollection<T> = HashSet<T>()
) : MutableCollection<T> by innerSet {

    var objectsAdded = 0

    override fun add(element: T): Boolean {
      objectsAdded++
      return innerSet.add(element)
    }

    override fun addAll(c: Collection<T>): Boolean {
      objectsAdded += c.size
      return innerSet.addAll(c)
    }
}
```

Делегирование реализации MutableCollection объекту в поле innerSet

Собственная реализация вместо делегирования

```
>>> val cset = CountingSet<Int>()
>>> cset.addAll(listOf(1, 1, 2))
>>> println("${cset.objectsAdded} objects were added, ${cset.size} remain")
3 objects were added, 2 remain
```

Наприклад, при усуванні дублікатів колекція дозволить виміряти, наскільки ефективно це робиться.

Прийом не створює залежності від особливостей реалізації основної колекції.

Ключове слово object

Спільне оголошення класу та його екземпляру

Використовується в різних випадках з однією метою: одночасно оголосити клас та створити його екземпляр. Типові ситуації:

- ***Оголошення об'єкту*** як спосіб реалізації шаблону «Одиночка»;
- Реалізація ***об'єкта-компаньйона***, що містить лише фабричні методи, а також пов'язані з класом методи, які не вимагаються звернень до його екземпляру.
- Запис ***об'єкту-виразу***, який використовується замість анонімного внутрішнього класу Java.

Шаблон «Одиночка».

Приклад: зарплатний фонд – можливий єдиний екземпляр.

Java: оголошується клас з приватним конструктором та статичним полем, яке містить єдиний екземпляр класу.

Kotlin: ключове слово object. Всередині все аналогічно класу, проте не допускаються конструктори!
Оголошення об'єктів створюються безпосередньо в точці їх визначення

```
object Payroll {
    val allEmployees = arrayListOf<Person>()

    fun calculateSalary() {
      for (person in allEmployees) {
        ...
      }
    }
}
```

Оголошення об’єктів

Як і звичайні змінні, дозволяють викликати методи та звертатись до властивостей:

```
Payroll.allEmployees.add(Person(…))
Payroll.calculateSalary()
```

Можуть успадковувати класи та інтерфейси.

Корисно, коли фреймворк потребує реалізації інтерфейсу, проте в вашій реалізації немає потрібного стану.

Наприклад, Java-інтерфейс java.util.Comparator, екземпляр якого приймає 2 об’єкта та повертає ціле число, що вказує на більший з цих об’єктів.

Такий компаратор не зберігає даних, тому для конкретного способу порівняння об’єктів достатньо одного компаратора.

Приклад: компаратор

Порівнює шляхи до файлів без урахування регістру

```
object CaseInsensitiveFileComparator : Comparator<File> {
    override fun compare(file1: File, file2: File): Int {
      return file1.path.compareTo(file2.path,
        ignoreCase = true)
    }
}
```

```
>>> println(CaseInsensitiveFileComparator.compare(
... File("/User"), File("/user")))
0
```

```
>>> val files = listOf(File("/Z"), File("/a"))
>>> println(files.sortedWith(CaseInsensitiveFileComparator))
[/a, /Z]
```

Оголошення об'єктів, як і шаблон «Одиночка», не завжди доречні у великих програмних системах.

Вони підходять для невеликих модулів з малою кількістю залежностей.

Основна причина – відсутність контролю над створенням об'єктів та неможливість управляти параметрами конструкторів.

Об’єкти можна оголошувати в класах

Такі об’єкти створюються в єдиному числі.
Наприклад, в ньому логічно розмістити компаратор, що порівнює об’єкти певного класу:

```
data class Person(val name: String) {
    object NameComparator : Comparator<Person> {
      override fun compare(p1: Person, p2: Person): Int =
        p1.name.compareTo(p2.name)
    }
}
>>> val persons = listOf(Person("Bob"), Person("Alice"))
>>> println(persons.sortedWith(Person.NameComparator))
[Person(name=Alice), Person(name=Bob)]
```

Оголошення об’єкта в Kotlin компілюється в клас зі статичним полем, що зберігає його єдиний екземпляр з назвою INSTANCE.
Для використання з Java-коду потрібно звертатись до статичного поля екземпляру:

```
/* Java */
CaseInsensitiveFileComparator.INSTANCE.compare(file1, file2);
```

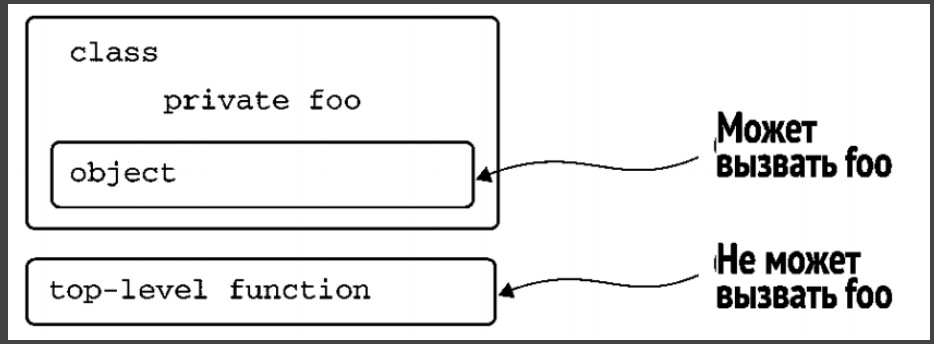
Об’єкти-компаньйони

Місце для фабричних методів та статичних членів класу

Функції вищого рівня не мають доступу до приватних членів класу:

Щоб написати функцію, яку можна викликати без екземпляру класу, проте з доступом до його внутрішнього устрою, можна зробити її членом оголошення об’єкту всередині класу.

Приклад – фабричний метод.



Один з об’єктів у класі можна відмітити ключовим словом companion.

Після цього можна звертатись до методів та вла стивостей об’єкта безпосередньо за ім’ям класу, який його містить.

```
class A {
    companion object {
      fun bar() {
        println("Companion object called")
      }
    }
}

>>> A.bar()
Companion object called
```

Об’єкти-компаньйони

Згадаємо приклад з класами FacebookUser та SubscribingUser.

Раніше ці класи реалізовували один інтерфейс – User.
Об’єднаємо їх в одному класі.

```
class User {
    val nickname: String

    constructor(email: String) {
      nickname = email.substringBefore('@')
    }

    constructor(facebookAccountId: Int) {
      nickname = getFacebookName(facebookAccountId)
    }
}
```

Альтернатива: використовувати фабричні методи для створення екземплярів класу.

Екземпляр User створюється викликом фабричного методу, а не за допомогою конструкторів.

Заміщення вторинних конструкторів фабричними методами

```
class User private constructor(val nickname: String) {
    companion object {
      fun newSubscribingUser(email: String) =
        User(email.substringBefore('@'))

      fun newFacebookUser(accountId: Int) =
        User(getFacebookName(accountId))
    }
}
```

Основной конструктор объявлен приватным

Объявление объекта-компаньона

Фабричный метод создает нового пользователя на основе идентификатора в Facebook

Можна викликати об’єкт-компаньйон через назву класу:

```
>>> val subscribingUser = User.newSubscribingUser("bob@gmail.com")
>>> val facebookUser = User.newFacebookUser(4)
>>> println(subscribingUser.nickname)
bob
```

Об’єкти-компаньйони як звичайні об’єкти

Об’єкт-компаньйон – це звичайний об’єкт, оголошений в класі.
Може мати ім’я, реалізовувати інтерфейс, мати функції-розширення та властивості-розширення.
Приклад: веб-служба для розрахунку заробітної плати підприємства.
Потрібно серіалізувати та десеріалізувати об’єкти в форматі JSON.
Можна помістити логіку серіалізації в об’єкт-компаньйон.

```
class Person(val name: String) {
    companion object Loader {
      fun fromJSON(jsonText: String): Person = ...
    }
}
```

```
>>> person = Person.Loader.fromJSON("{name: 'Dmitry'}")
>>> person.name
Dmitry
>>> person2 = Person.fromJSON("{name: 'Brent'}")
>>> person2.name
Brent
```

У більшості випадків можна посилатись на об’єкт-компаньйон через назву класу, що його містить.
Якщо назви об’єкта-компаньйона не вказано, за замовчуванням обирається Companion.

Об’єкти-компаньйони та статичні члени в Kotlin

Об’єкт-компаньйон класу компілюється як і звичайний об’єкт: статичне поле класу посилається на власний екземпляр.

Якщо об’єкт не має назви, до нього можна звернутись із Java-коду через посилання Companion:

```
/* Java */
Person.Companion.fromJSON("...");
```

Проте може знадобитись працювати з Java-кодом, який вимагає, щоб методи вашого класу були статичними.

Тоді додається анотація @JvmStatic перед відповідним методом.

Для статичного поля – анотація @JvmField перед властивістю верхнього рівня або оголошеною в об’єкті.

Реалізація інтерфейсів в об’єктах-компаньйонах

Нехай в системі багато типів об’єктів, зокрема Person.
Бажання: забезпечити єдиний спосіб створення об’єктів усіх типів.
Нехай є інтерфейс JSONFactory для об’єктів, які можна десеріалізувати з формату JSON.

```
interface JSONFactory<T> {
    fun fromJSON(jsonText: String): T
}

class Person(val name: String) {
    companion object : JSONFactory<Person> {
      override fun fromJSON(jsonText: String): Person = ...
    }
}
```

Объект-компаньон, реализующий интерфейс

Якщо є функція, що використовує абстрактну фабрику для завантаження сутностей, їй передається об’єкт Person

```
fun loadFromJSON<T>(factory: JSONFactory<T>): T {
    ...
}

loadFromJSON(Person)
```

Передача объекта-компаньона в функцию

Розширення об’єктів-компаньйонів

Причина: потрібно створити функцію, яку можна викликати для самого класу так же, як методи об’єктів-компаньйонів або статичні методи в Java.

Якщо клас має об’єкт-компаньйон, можна визначити для нього функцію-розширення.

Приклад: розділення обов’язків у класі Person.

Клас буде частиною модуля основної бізнес-логіки, проте зв’язувати його з конкретним форматом даних буде небажано.

```
// модуль реализации бизнес-логики
class Person(val firstName: String, val lastName: String) {
    companion object {                    <- Объявление пустого объекта-компаньона
    }
}

// модуль реализации взаимодействий между клиентом и сервером
fun Person.Companion.fromJSON(json: String): Person {    <- Объявление функции-расширения
    ...
}

val p = Person.fromJSON(json)
```

Об’єкти-вирази

Інший спосіб реалізації анонімних внутрішніх класів

Ключове слово object можна використовувати і для створення ***анонімних об’єктів***.
Вони заміняють анонімні внутрішні класи з Java.
Стандартний приклад – реалізація обробника подій:

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

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

Объявление анонимного объекта, наследующего MouseAdapter

Переопределение методов MouseAdapter

Об’єкт-вираз оголошує клас та створює його екземпляр, проте не присвоює їм назви.
Зазвичай у цьому немає потреби, оскільки об’єкт використовується як параметр виклику функції.
Якщо об’єкту потрібно дати назву, його можна зберегти у змінній:

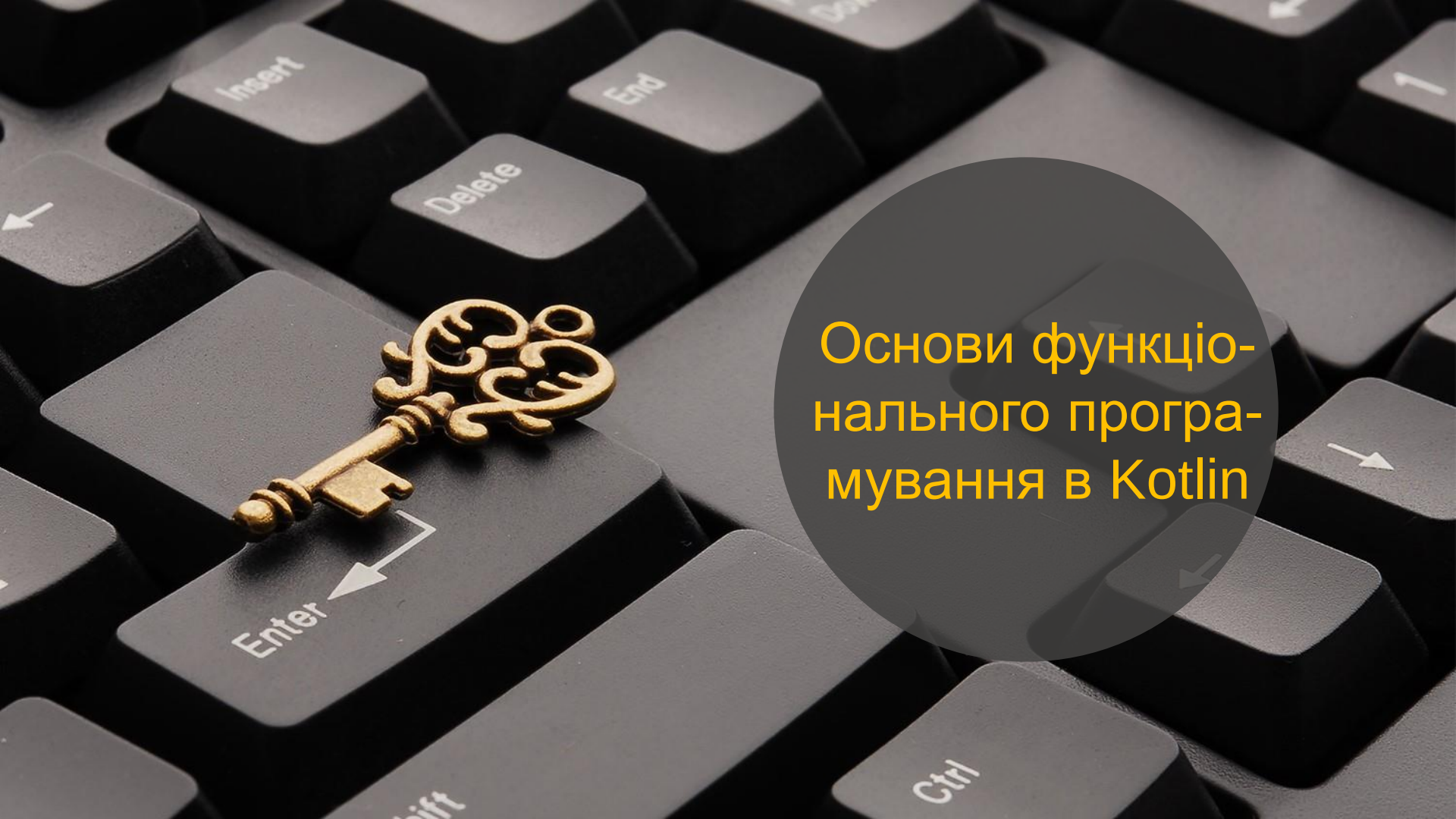
```
val listener = object : MouseAdapter() {
  override fun mouseClicked(e: MouseEvent) { ... }
  override fun mouseEntered(e: MouseEvent) { ... }
}
```

Об’єкти-вирази

Анонімний об’єкт Kotlin може реалізовувати кілька інтерфейсів.
Анонімні об’єкти не є «одиночками», на відміну від оголошення об’єкту.
Як анонімні Java-класи, код об’єкту-виразу може звертатись до змінних у функціях, в яких його було створено.
Проте в Kotlin немає обмеження на фінальність змінних.
Приклад: підрахунок кількості кліків у вікні за допомогою обробника подій:

```
fun countClicks(window: Window) {
    var clickCount = 0                          <- Объявление локальной переменной

    window.addMouseListener(object : MouseAdapter() {
      override fun mouseClicked(e: MouseEvent) {
        clickCount++                            <- Изменение значения переменной
      }
    })
    // ...
}
```

Об’єкти-вираз корисні, коли в анонімному об’єкті потрібно переозначити кілька методів.



Основи функціонального програмування в Kotlin

Лямбда-вирази

Приклад: визначення реакції на натиснення кнопки

Додамо обробник кліку, що реалізує інтерфейс OnClickListener з єдиним методом onClick().

```
/* Java */
button.setOnClickListener(new OnClickListener() {
  @Override
  public void onClick(View view) {
    /* действия по щелчку */
  }
});
```

Код характеризується надмірністю.
У Java 8 та Kotlin можна використовувати лямбда-вирази.

```
button.setOnClcikListener {/* дії по кліку */}
```

Лямбда-вирази та колекції

Приклад з використанням класу Person (зберігає ім'я та вік людини)

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

Нехай потрібно знайти найстарішу людину в списку людей.

Підхід без лямбда-виразів

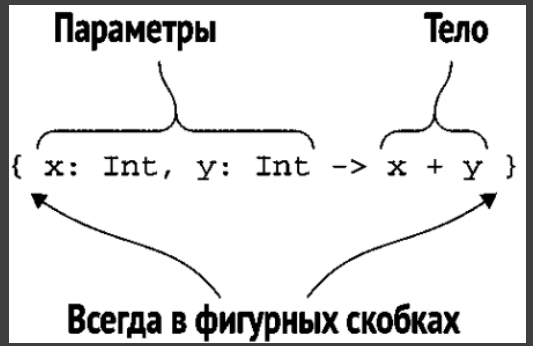
```
fun findTheOldest(people: List<Person>) {
  var maxAge = 0                          <— Хранит максимальный возраст
  var theOldest: Person? = null           <— Хранит самого старого человека
  for (person in people) {
    if (person.age > maxAge) {            <— Если следующий старше предыдущего,
      maxAge = person.age                    максимальный возраст изменится
      theOldest = person
    }
  }
  println(theOldest)
}
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> findTheOldest(people)
Person(name=Bob, age=31)
```

Підхід на основі лямбда-виразів та функцій колекції

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.maxBy { it.age })      <— Найдет элемент коллекции с максимальным
Person(name=Bob, age=31)                     значением свойства age
```

- Функцію maxBy() можна викликати для будь-якої колекції.
- У фігурних дужках – лямбда-вираз, аргумент якого є елементом колекції.
- Лямбда-вираз повертає значення для порівняння.

Синтаксис лямбда-виразів



Часто лямбда-вираз оголошується безпосередньо при передачі в функцію

У Kotlin лямбда-вирази завжди оточені фігурними дужками, а круглі дужки для переліку параметрів відсутні.

Лямбда-вираз можна зберегти разом із змінною або викликати напряму.

```
>>> val sum = { x: Int, y: Int -> x + y }
>>> println(sum(1, 2))
3
```

Вызов лямбда-выражения, хранящегося в переменной

```
>>> { println(42) }()
42
```

Зручніше використовувати бібліотечну функцію run:

```
>>> run { println(42) }
42
```

Код пошуку найстарішої людини в списку

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.maxBy { it.age })
Person(name=Bob, age=31)
```

Без синтаксичних скорочень пошук виглядав би так:

```
people.maxBy({p: Person -> p.age})
```

Код надмірний: тип параметру можна вивести з контексту, багато пунктуаційних знаків.

- Kotlin дозволяє винести лямбда-вираз за дужки, якщо він буде останнім аргументом функції:

```
people.maxBy() {p: Person -> p.age}
```

- Якщо лямбда-вираз – єдиний аргумент, можна прибрати круглі дужки:

```
people.maxBy {p: Person -> p.age}
```

- Спростимо код: тип параметру буде виводитись з контексту:

```
people.maxBy {p -> p.age}
```

- Останнє спрощення – використання імені параметру за замовчуванням:

```
people.maxBy {it.age}
```

Більш складний виклик

Бібліотечна функція joinToString() – альтернатива toString(), яка приймає функцію як до датковий параметр (іменований аргумент):

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> val names = people.joinToString(separator = " ",
...         transform = { p: Person -> p.name })
>>> println(names)
Alice Bob
```

За дужками код виглядатиме так:

```
people.joinToString(" ") {p: Person -> p.name}
```

Якщо лямбда-вираз зберігається у змінній, компілятор не має контексту для виводу типу параметру. Його слід вказувати явно:

```
>>> val getAge = { p: Person -> p.age }
>>> people.maxBy(getAge)
```

Лямбда-вирази можуть мати багато рядків

```
>>> val sum = { x: Int, y: Int ->
...     println("Computing the sum of $x and $y...")
...     x + y
...   }
>>> println(sum(1, 2))
Computing the sum of 1 and 2...
3
```

Доступ до змінних із контексту

По аналогії до анонімного внутрішнього класу, визначений у функції лямбда-вираз може звертатись до оголошених нею перед тим параметрів та локальних змінних:

```
fun printMessagesWithPrefix(messages: Collection<String>, prefix: String) {
    messages.forEach {
      println("$prefix $it")
    }
}

>>> val errors = listOf("403 Forbidden", "404 Not Found")
>>> printMessagesWithPrefix(errors, "Error:")
Error: 403 Forbidden
Error: 404 Not Found
```

Принимает в качестве аргумента лямбда-выражение, определяющее, что делать с каждым элементом

Обращение к параметру «prefix» из лямбда-выражения

На відміну від Java, Kotlin дозволяє змінювати такі змінні всередині лямбда-виразів.

Зміна локальних змінних всередині лямбда-виразу

```
fun printProblemCounts(responses: Collection<String>) {
    var clientErrors = 0
    var serverErrors = 0
    responses.forEach {
      if (it.startsWith("4")) {
        clientErrors++
      } else if (it.startsWith("5")) {
        serverErrors++
      }
    }
    println("$clientErrors client errors, $serverErrors server errors")
}

>>> val responses = listOf("200 OK", "418 I'm a teapot",
...         "500 Internal Server Error")
>>> printProblemCounts(responses)
1 client errors, 1 server errors
```

Объявление переменных, к которым будет обращаться лямбда-выражение

Изменение переменных внутри лямбда-выражения

Говорять, що зовнішні змінні ***захоплюються*** лямбда-виразом

Лямбда-вираз може зберігатися в змінну, що може продовжити життя захоплених локальних змінних.

Асинхронні лямбда-вирази

Якщо лямбда-вираз є обробником подій або працює асинхронно, модифікація захоплених змінних відбудеться тільки за умови його виконання.

Приклад: неправильний підрахунок натиснень на кнопку:

```
fun tryToCountButtonClicks(button: Button): Int {
    var clicks = 0
    button.onClick { clicks++ }
    return clicks
}
```

Завжди повертатиме 0, оскільки обробник onClick() викликається після виходу з функції.

Для правильної роботи кількість натиснень слід зберігати в локальну змінну, а за межами функції – на приклад, у властивості класу.

Посилання на члени класу

Лямбда-вирази дозволяють передати блок коду в якості параметру функції.
Якщо цей блок уже є функцією, за допомогою оператору :: він перетворюється на ***посилання на член класу (member reference).***


```
val getAge = Person::age
```

Це скорочений запис виразу:

```
val getAge = {person: Person -> person.age}
```

Незалежно від того, чи вказує посилання на функцію або властивість, круглі дужки після назви члена класу при створенні посилання не ставляться.

Посилання на член класу має той же тип, що і лямбда-вираз, тому вони взаємозамінні:

```
people.maxBy(Person::age)
```

```
fun salute() = println("Salute!")
>>> run(::salute)
Salute!
```

Посилання на функцію замість лямбда-виразу

```
val action = { person: Person, message: String ->
    sendEmail(person, message)
}
val nextAction = ::sendEmail
```

Это лямбда-выражение делегирует работу функции sendEmail

Вместо него можно использовать ссылку на функцию

Можна зберегти або відкласти операцію створення екземпляру класу за допомогою поси лання на конструктор:

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

>>> val createPerson = ::Person
>>> val p = createPerson("Alice", 29)
>>> println(p)
Person(name=Alice, age=29)
```

Операция создания экземпляра Person сохраняется в переменную

Посилання можна отримати і на функцію-розширення:

```
fun Person.isAdult() = age >= 21
val predicate = Person::isAdult
```

Зв’язані посилання

У Kotlin 1.0 завжди потрібно передавати екземпляр класу при зверненні до його методу або властивості за посиланням.

У Kotlin 1.1 додано підтримку зв’язаних посилань – спеціального синтаксису для захоплення посилання на метод конкретного екземпляру класу:

```
>>> val p = Person(“Dmitry”, 34)
>>> val personsAgeFunction = Person::age
>>> println(personsAgeFunction(p))
34
>>> val dmitrysAgeFunction = p::age
>>> println(dmitrysAgeFunction())
34
```

До Kotlin 1.1 замість використання зв’язаного посилання на метод p::age потрібно було писати лямбда-ви раз {p.age}

Функціональний стиль при роботі з колекціями

Функціональний стиль при роботі з колекціями

Основа роботи – функції filter() та map().

- filter() – виконує обхід колекції, відбираючи елементи, для яких лямбда-вираз поверне true

```
>>> val list = listOf(1, 2, 3, 4)
>>> println(list.filter { it % 2 == 0 })
[2, 4]
```

Можна знайти у списку всіх людей, кому за 30 років

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.filter { it.age > 30 })
[Person(name=Bob, age=31)]
```

Функція зможе видалити елементи, проте не зможе змінити їх.
Для цього використовується функція map().

Основа роботи – функції filter() та map().

- map() – застосовує задану функцію до кожного елементу колекції, поєднуючи результати в нову колекцію:

```
>>> val list = listOf(1, 2, 3, 4)
>>> println(list.map { it * it })
[1, 4, 9, 16]
```

Для виводу списку імен можна перетворити початковий список:

```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31))
>>> println(people.map { it.name })
[Alice, Bob]
```

Код можна елегантно переписати: **`people.map(Person::name)`**
Функції можна поєднувати та комбінувати:

```
>>> people.filter { it.age > 30 }.map(Person::name)
[Bob]
```

Приклади застосування функцій map() і filter()

Нехай потрібні імена найдорослiших людей у групі.

- Знаходимо максимальний вік у групі та повертаємо всіх людей з цим віком:

```
people.filter {it.age == people.maxBy (Person::age).age}
```

- Тут пошук максимального віку відбувається для кожної людини, а бажано шукати 1 раз:

```
val maxAge = people.maxBy (Person::age).age
people.filter {it.age == maxAge}
```

До словників також можна застосувати функції відбору та перетворення:

```
>>> val numbers = mapOf(0 to "zero", 1 to "one")
>>> println(numbers.mapValues { it.value.toUpperCase() })
{0=ZERO, 1=ONE}
```

- Для обробки ключів та значень існують окремі функції: filterKeys(), mapKeys(), filterValues() та mapValues().

Застосування предикатів до колекцій

Поширена задача – перевірка всіх елементів колекції на відповідній деякій умові.

- Вирішують функції all() та any().
- Функція count() перевіряє, скільки елементів задовольняє предикат.
- Функція find() повертає перший шуканий елемент

Перевіримо на предикаті `val canBeInClub27 = {p: Person -> p.age <= 27}`

- Всі елементи, які задовольняють предикат:

```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.all(canBeInClub27))
false
```

- Для знаходження хоча б одного доречного елементу – протилежна умова:

```
>>> println(people.any(canBeInClub27))
true
```

Застосування предикатів до колекцій

Перевіримо на предикаті `val canBeInClub27 = {p: Person -> p.age <= 27}`

- Для перевірки кількості елементів, які задовольняють предикат:

```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.count(canBeInClub27))
1
```

- Можна також знайти розмір відфільтрованої колекції, проте тут буде створена проміжна колекція:

```
>>> println(people.filter(canBeInClub27).size)
1
```

- Для знаходження елементу, що задовольняє предикат, потрібно використати find():

```
>>> val people = listOf(Person("Alice", 27), Person("Bob", 31))
>>> println(people.find(canBeInClub27))
Person(name=Alice, age=27)
```

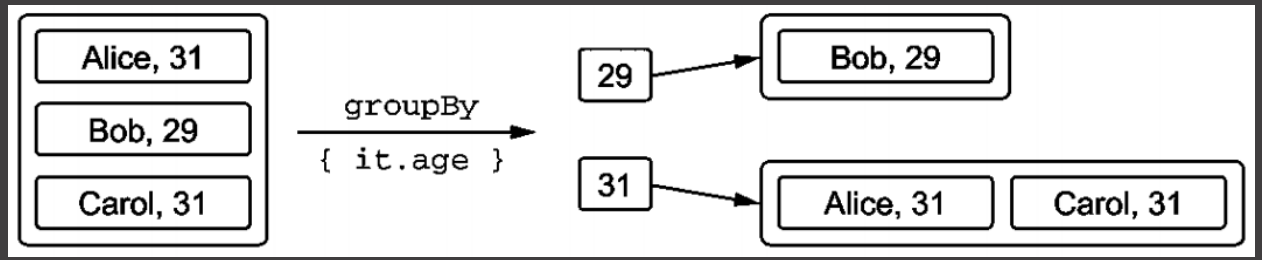
Групування значень у списку з функцією groupBy()

Перевіримо на предикаті `val canBeInClub27 = {p: Person -> p.age <= 27}`

- Потрібно погрупувати елементи колекції відповідно до критерію:

```
>>> val people = listOf(Person("Alice", 31),
...     Person("Bob", 29), Person("Carol", 31))
>>> println(people.groupBy { it.age })
```

- Схема роботи



- Результат застосування:

```
{29=[Person(name=Bob, age=29)],
31=[Person(name=Alice, age=31), Person(name=Carol, age=31)]}
```

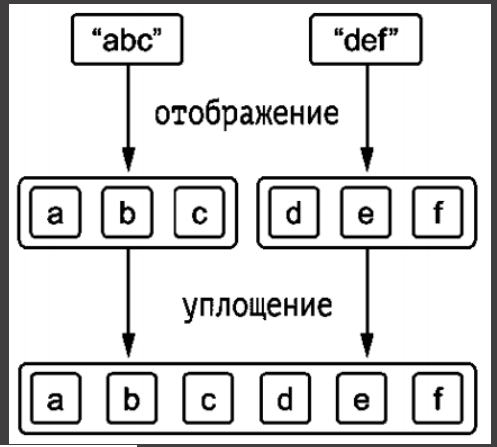
Обробка елементів вкладених колекцій

Нехай є сховище книг, представлених класом Book

```
class Book(val title: String, val authors: List<String>)
```

- Знайти множину всіх авторів у бібліотеці:

```
books.flatMap {it.authors}.toSet()
```



Функція flatMap():
1) перетворює (map) кожен елемент в колекцію відповідно до функції, що передана в аргументі
2)збирає (flatten) кілька списків в один

```
>>> val books = listOf(Book("Thursday Next", listOf("Jasper Fforde")),
...                    Book("Mort", listOf("Terry Pratchett")),
...                    Book("Good Omens", listOf("Terry Pratchett",
...                                              "Neil Gaiman")))
>>> println(books.flatMap { it.authors }.toSet())
[Jasper Fforde, Terry Pratchett, Neil Gaiman]
```

- Якщо потрібна просто плоска колекція, без перетворень, досить використати функцію flatten():

```
listOfLists.flatten()
```

Відкладені операції над колекціями

Послідовності

Відкладені операції над колекціями

Послідовності

Функції filter() та map() ***негайно*** створюють проміжні колекції, зберігаючи результат, що отримано на кожному проміжному кроці.
Альтернатива: послідовності (sequences) – уникають створення тимчасових об'єктів.

Приклад. Знайти множину всіх авторів у бібліотеці:

```
people.map(Person::name).filter {it.startsWith("A")}
```

- Створюються 2 списки: для зберігання результатів filter() та map()

- Ефективніша реалізація за допомогою послідовностей:

```
people.asSequence()
      .map(Person::name)
      .filter {it.startsWith("A")}
      .toList()
```

Відкладені операції над колекціями

Послідовності та виконання операцій над ними

Точка входу для виконання відкладених операцій – інтерфейс Sequence.

- Визначає лише 1 метод - iterator() – для отримання значень послідовності.
- Елементи послідовності обчислюються «ліниво».
- Будь-яка колекція перетворюється в послідовність за допомогою функції-розширення asSequence().
- Зворотне перетворення – toList().

Послідовність ефективніша за колекцію, проте постачає менше можливостей.

- Відсутній доступ до елементу за індексом, потрібно перетворювати у список.
- Послідовність застосовується для виконання ланцюга операцій над ***великою*** колекцією.

Операції над послідовностями діляться на 2 категорії:

- ***Проміжна*** операція повертає іншу послідовність, яка знає, як перетворювати елементи початкової послідовності.
- ***Завершальна*** операція повертає результат (колекцію, елемент, число або інший об'єкт), отриманий протягом перетворень початкової колекції.

Відкладені операції над колекціями

Послідовності та виконання операцій над ними

Виконання проміжних операцій завжди відкладається.

```
>>> listOf(1, 2, 3, 4).asSequence()
...                    .map { print("map($it) "); it * it }
...                    .filter { print("filter($it) "); it % 2 == 0 }
```

- Код нічого не виведе в консоль – перетворення map() та filter() відкладені.

```
>>> listOf(1, 2, 3, 4).asSequence()
...                    .map { print("map($it) "); it * it }
...                    .filter { print("filter($it) "); it % 2 == 0 }
...                    .toList()
map(1) filter(1) map(2) filter(4) map(3) filter(9) map(4) filter(16)
```

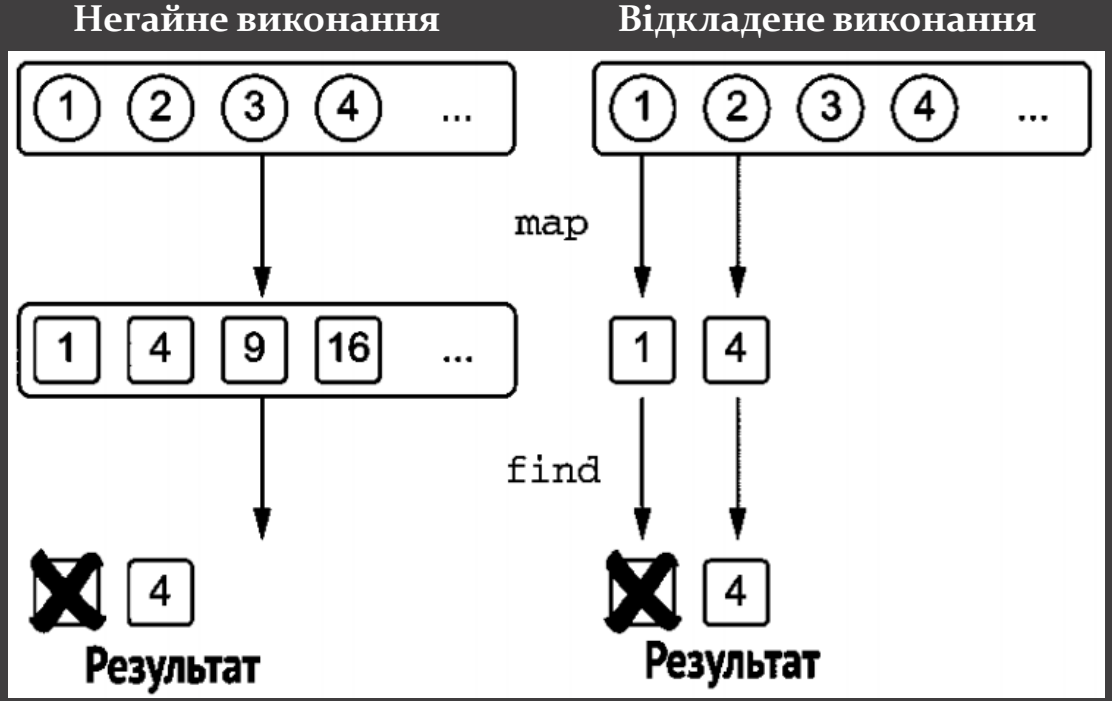
- Завершальна операція змушує виконатись усі відкладені обчислення.

Важлива особливість – порядок обчислень.

- Негайний спосіб виконає кожну операцію для всієї колекції.
- Відкладений спосіб буде обробляти елементи один за одним.

Послідовність дій при негайному та відкладеному виконанні

```
>>> println(listOf(1, 2, 3, 4).asSequence()
                              .map { it * it }.find { it > 3 })
4
```



Спочатку обчислимо квадрат числа, а потім знайдемо перший елемент, що більше 3.

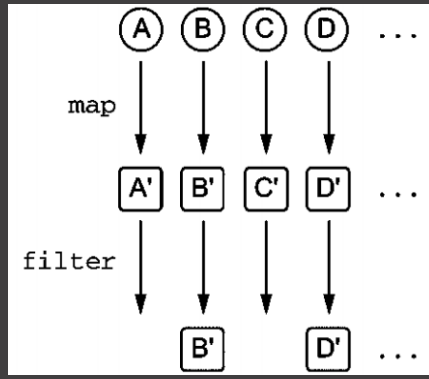
Якщо застосувати до колекції, а не послідовності, спочатку виконається map().

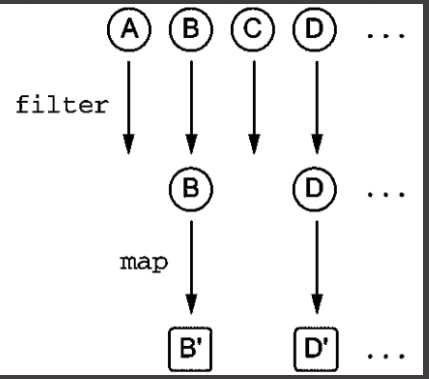
- Далі буде знайдено елемент проміжної колекції, що задовольняє предикат.

Порядок виконання впливає на продуктивність

Нехай є колекція людей, імена яких треба вивести за умови обмеженості довжини.

- Потрібно відобразити кожен об'єкт Person в ім'я людини та вибрати достатньо короткі з них.
- Результат не залежить від порядку filter() і map(), проте буде різна продуктивність.






```
>>> val people = listOf(Person("Alice", 29), Person("Bob", 31),
... Person("Charles", 31), Person("Dan", 21))
```

```
>>> println(people.asSequence().map(Person::name)
... .filter { it.length < 4 }.toList())
[Bob, Dan]
```

```
>>> println(people.asSequence().filter { it.name.length < 4 }
... .map(Person::name).toList())
[Bob, Dan]
```

Створення послідовностей

Можна не лише за допомогою asSequence(), але й за допомогою функції generateSequence().

```
>>> val naturalNumbers = generateSequence(0) { it + 1 }
>>> val numbersTo100 = naturalNumbers.takeWhile { it <= 100 }
>>> println(numbersTo100.sum())
5050
```

Все отложенные операции выполнятся при обращении к «sum»

- naturalNumbers та numbersTo100 – послідовності з відкладеним виконанням операцій.

Інший варіант використання – послідовність батьків.

- Якщо в елемента є батьківський елемент того ж типу, цікаво дізнатись властивості всіх його предків у послідовності.
- Приклад: чи знаходиться файл у прихованому каталозі?

```
fun File.isInsideHiddenDirectory() =
        generateSequence(this) { it.parentFile }.any { it.isHidden }

>>> val file = File("/Users/svtk/.HiddenDir/a.txt")
>>> println(file.isInsideHiddenDirectory())
true
```

Використання функціональних інтерфейсів Java

Приклад функціонального інтерфейсу в Java

Уже бачили передачу лямбда-виразу в Java-метод:

```
button.setOnClickListener {/* дії по кліку */}
```

- Метод setOnClickListener(OnClickListener l) класу Button

```
/* Java */
public class Button {
    public void setOnClickListener(OnClickListener l) { ... }
}
```

- В інтерфейсі OnClickListener присутній єдиний метод onClick()

```
/* Java */
public interface OnClickListener {
    void onClick(View v);
}
```

- До Java 8 потрібно створювати новий екземпляр анонімного класу та передати його методу setOnClickListener():

```
button.setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
      ...
    }
}
```

Передача лямбда-виразу в мові Kotlin

Передача лямбда-виразу в мові Kotlin: `button.setOnClickListener {view -> …}`

```
public interface OnClickListener {
    void onClick(View v);
}
```

→ `{ view -> ... }`

- Можливо завдяки тому, що інтерфейс є функціональним (SAM – Single Abstract Method).

На відміну від Java, в мові Kotlin є справжні типи функцій.

- У Kotlin використовуються вони, а не типи функціональних інтерфейсів.
- Kotlin не підтримує автоматичного перетворення лямбда-виразів в об'єкти, які реалізують інтер фейси.

Передача лямбда-виразів у Java-метод

Передача лямбда-виразу в Java-метод можлива тоді, коли цей метод приймає функціональний інтерфейс.

- Приклад:

```
/* Java */
void postponeComputation(int delay, Runnable computation);
```

- У Kotlin можна викликати цей метод, передавши в аргументі лямбда-вираз:

```
postponeComputation(1000) { println(42) } // створюється 1 екземпляр Runnable
```

- Код з аналогічним ефектом:
- При оголошенні об'єкта при кожному виклику створюється новий екземпляр.

```
postponeComputation(1000, object : Runnable {
    override fun run() {
      println(42)
    }
})
```

```
val runnable = Runnable { println(42) }
fun handleComputation() {
    postponeComputation(1000, runnable)
}
```

Компилируется в глобальную переменную; в программе существует только один экземпляр

В каждый вызов метода handleComputation будет передаваться один и тот же экземпляр

Особливості реалізації лямбда-виразів

Коли лямбда-вираз захоплює змінні з оточуючого контексту, повторно використовувати один і той же екземпляр в кожному виклику стає неможливо.

- Компілятор створює новий об'єкт для кожного виклику.
- Наприклад, у функції кожен її виклик використовує новий екземпляр Runnable:

```
fun handleComputation(id: String) {
    postponeComputation(1000) { println(id) }
}
```

Лямбда-выражение захватывает переменную «id»

Для каждого вызова handleComputation создается новый экземпляр Runnable

- У Kotlin 1.0 кожний лямбда-вираз, що не є вбудованим, компілюється в анонімний клас.
- З підтримкою генерації байт-коду Java 8 компілятору не потрібно створювати окремі .class-файли для кожного лямбда-виразу.

SAM-конструктори

Явне перетворення лямбда-виразів у функціональні інтерфейси

SAM-конструктор – це згенерована компілятором функція, яка дозволяє явно перетворити лямбда-вираз в екземпляр функціонального інтерфейсу.

- Можна використовувати тоді, коли компілятор не виконує автоматичного перетворення:

```
fun createAllDoneRunnable(): Runnable {
    return Runnable { println("All done!") }
}

>>> createAllDoneRunnable().run()
All done!
```

- Назва SAM-конструктора співпадає з назвою функціонального інтерфейсу.
- Приймає 1 аргумент – лямбда-вираз.
- Повертає екземпляр класу, що реалізує даний інтерфейс.

SAM-конструктори

Збереження у змінній екземпляру функціонального інтерфейсу, утвореного з лямбда-виразу

Нехай потрібно використовувати один обробник для кількох кнопок.

- Можна оголосити обробник подій, використовуючи оголошення об’єкту, що реалізує інтерфейс OnClickListener.
- SAM-конструктор – більш лаконічний.

```
val listener = OnClickListener { view ->
    val text = when (view.id) {
      R.id.button1 -> "First button"
      R.id.button2 -> "Second button"
      else -> "Unknown button"
    }
    toast(text)                    <— Выводит
}
button1.setOnClickListener(listener)
button2.setOnClickListener(listener)
```

У лямбда-виразі, на відміну від анонімного об’єкта, немає посилання this.

- З точки зору компілятора лямбда-вираз – це блок коду, а не об’єкт.
- Посилання this у лямбда-виразі відноситься до зовнішнього класу.
- Для відміни підписки обробника на події підчас обробки лямбда-вирази недоречні – потрібен анонімний об’єкт.

Лямбда-вирази з отримувачами

Функції with() та apply()

Лямбда-вирази з отримувачами

Можливість виклику методів іншого об’єкта в тілі лямбда-виразів без додаткових кваліфікаторів

Унікальна особливість Kotlin.

- Конструкція with використовує лямбда-вираз з отримувачем.
- Приклад: генерація алфавіту.
- У змінній result викликається кілька методів об’єкта

```
fun alphabet(): String {
    val result = StringBuilder()
    for (letter in 'A'..'Z') {
      result.append(letter)
    }
    result.append("\nNow I know the alphabet!")
    return result.toString()
}
>>> println(alphabet())
ABCDEFGHIJKLMNOPQRSTUVWXYZ
Now I know the alphabet!
```

- Використання функції with для генерації алфавіту:

```
fun alphabet(): String {
    val stringBuilder = StringBuilder()
    return with(stringBuilder) {
      for (letter in 'A'..'Z') {
        this.append(letter)
      }
      append("\nNow I know the alphabet!")
      this.toString()
    }
}
```

Определяется получатель, методы которого будут вызываться

Вызов метода получателя с помощью ссылки «this»

Вызов метода без ссылки «this»

Возврат значения из лямбда-выражения

Структура with – функція, що приймає 2 аргументи

- Тут – змінну stringBuilder та лямбда-вираз.
- Функція with перетворює перший аргумент в отримувач лямбда-виразу з другого аргументу.

Лямбда-вирази з отримувачем та функції-розширення

Посилання this в останньому лістингу вказує на екземпляр stringBuilder (1й аргумент).

- До методів stringBuilder можна звертатись явно (через this) або напряму.

У тілі функції-розширення посилання this вказує на екземпляр типу, який розширяє функція.

- Його можна опускати, звертаючись до членів отримувача напряму.
- Можна використовувати аналогію:

Звичайна функція	Функція-розширення
Звичайний лямбда-вираз	Лямбда-вираз з отримувачем

Реорганізуємо фунцію alphabet()

Усунемо додаткову змінну stringBuilder

```
fun alphabet(): String {
    val stringBuilder = StringBuilder()
    return with(stringBuilder) {
      for (letter in 'A'..'Z') {
        this.append(letter)
      }
      append("\nNow I know the alphabet!")
      this.toString()
    }
}
```

Возврат значения из лямбда-выраж

```
fun alphabet() = with(StringBuilder()) {
    for (letter in 'A'..'Z') {
      append(letter)
    }
    append("\nNow I know the alphabet!")
    toString()
}
```

Іноді потрібно, щоб результатом виклику став сам об'єкт-отримувач, а не результат виконан ня лямбда виразу.

- Застосовується бібліотечна функція apply().
- Завжди повертає переданий в аргументі об'єкт.
- Оголошена як функція-розширення.
- Корисна при створенні екземпляру, в якого потрі бно відразу ініціалізувати властивості.

```
fun alphabet() = StringBuilder().apply {
    for (letter in 'A'..'Z') {
      append(letter)
    }
    append("\nNow I know the alphabet!")
}.toString()
```

Робота функції apply()

Приклад: створення Android-компоненту TextView з певними зміненими атрибутами.

```
fun createViewWithCustomAttributes(context: Context) =
    TextView(context).apply {
      text = "Sample Text"
      textSize = 20.0
      setPadding(10, 0, 0, 0)
    }
```

- Функція apply() дозволяє використати компактний синтаксис тіла-виразу.
- У переданому в функцію apply() лямбда-виразі екземпляр TextView стає отримувачем.
- Завдяки цьому можна викликати його методи та змінювати властивості.

За допомогою стандартної бібліотечної функції buildString() можна ще більше спростити генерацію алфавіту.

```
fun alphabet() = StringBuilder().apply {
    for (letter in 'A'..'Z') {
      append(letter)
    }
    append("\nNow I know the alphabet!")
}.toString()
```

```
fun alphabet() = buildString {
    for (letter in 'A'..'Z') {
      append(letter)
    }
    append("\nNow I know the alphabet!")
}
```

Система типів у мові програмування Kotlin

Підтримка null-значення

У порівнянні з Java система типів Kotlin має особливості, спрямовані на підвищення надійності коду.

- Типи з підтримкою порожнього значення.
- Доступні тільки для читання колекції.
- Деякий Java-синтаксис прибрано, зокрема, для масивів.

Підтримка null-значення дозволяє уникати NullPointerException.

- Багато потенційних помилок знаходяться ще на етапі компіляції.

Чи безпечна Java-функція?

- Чи може функція приймати null-значення?
- Чи варто додавати перевірку на null?

```
/* Java */
int strLen(String s) {
  return s.length();
}
```

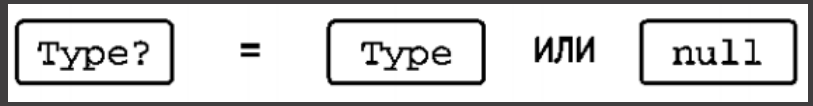
```
fun strLen(s: String) = s.length
```

```
>>> strLen(null)
ERROR: Null can not be a value of a non-null type String
```

Дозволяємо виклик з будь-якими аргументами, в т. ч. null

Після назви типу додається знак питання:

```
fun strLenSafe(s: String?) = ...
```



- Набір доступних операцій з таким значенням значно звужується.
- Наприклад, більше не викликаються методи.

```
>> fun strLenSafe(s: String?) = s.length()
ERROR: only safe (?.) or non-null asserted (!!.) calls are allowed
on a nullable receiver of type kotlin.String?
```

- Також неможливо присвоїти значення змінній, тип якої не підтримує null:

```
>>> val x: String? = null
>>> var y: String = x
ERROR: Type mismatch: inferred type is String? but String was expected
```

- Не можна передавати значення з підтримкою null у функції, параметр яких ніколи не може бу ти null

```
>>> strLen(x)
ERROR: Type mismatch: inferred type is String? but String was expected
```

Що робити?

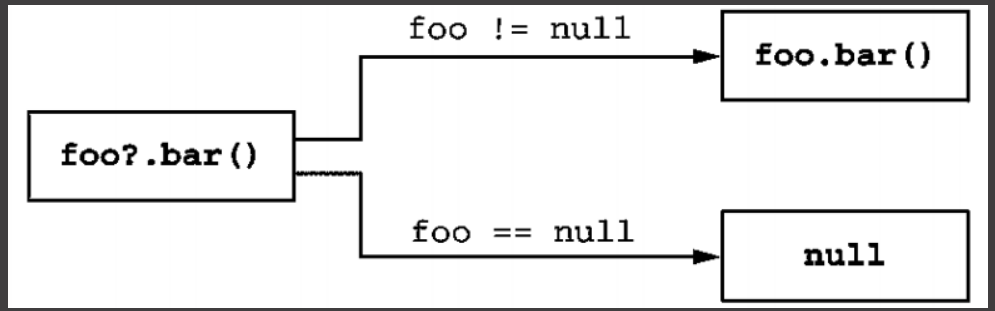
Порівняти значення з null.

- Компілятор запам'ятає результат та вважатиме, що змінна не може мати значення null в області дії порівняння.
- Код розпухне від перевірок.

```
fun strLenSafe(s: String?): Int =
    if (s != null) s.length else 0
>>> val x: String? = null
>>> println(strLenSafe(x))
0
>>> println(strLenSafe("abc"))
3
```

Оператор безпечного виклику ?. поєднує перевірку на null та виклик методу.

- Заміняє перевірки рівності з null.




```
fun printAllCaps(s: String?) {
    val allCaps: String? = s?.toUpperCase()
    println(allCaps)
}

>>> printAllCaps("abc")
ABC
>>> printAllCaps(null)
null
```

Інші способи боротьби з NullPointerException

У Java є інструменти, які дозволяють боротись з NullPointerException.

- Анотації @Nullable, @NotNull вказують на допустимість null-значення.
- В Intellij IDEA існують інструменти, які знаходять можливі місця виникнення NullPointerException за допомогою цих анотацій.
- Проте вони не є стандартом і не вирішують проблему в цілому.

Інший підхід – заборона використання null-значення, застосування спеціальних обгорток (як тип Optional у Java 8) для представлення значень, що можуть бути або не бути визначен ими.

- Недоліки: код стає ще громіздкішим, падає продуктивність, непослідовне застосування в екосистемі.
- JDK, фрейморки Android та інші сторонні бібліотеки працюють з null.

Оператор безпечного виклику

Використовується і для доступу до властивостей.

```
class Employee(val name: String, val manager: Employee?)

fun managerName(employee: Employee): String? = employee.manager?.name

>>> val ceo = Employee("Da Boss", null)
>>> val developer = Employee("Bob Smith", ceo)
>>> println(managerName(developer))
Da Boss
>>> println(managerName(ceo))
null
```

Безпечні виклики та ієрархія об’єктів, властивості яких здатні приймати null

Нехай зберігається інформація про людину, її компанію та адресу компанії.

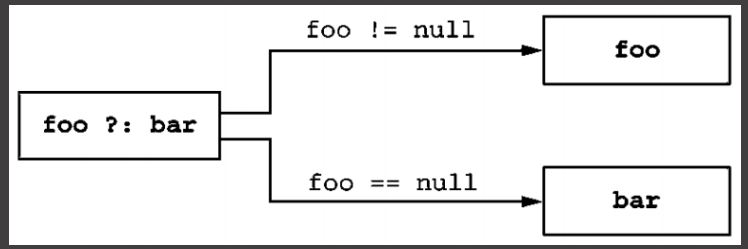
- Інформація щодо компанії та адреси може бути пропущена.
- Оператор ?. Дозволить без додаткових перевірок визначити, з якої країни людина.

```
class Address(val streetAddress: String, val zipCode: Int,
        val city: String, val country: String)

class Company(val name: String, val address: Address?)

class Person(val name: String, val company: Company?)

fun Person.countryName(): String {
    val country = this.company?.address?.country
    return if (country != null) country else "Unknown"
}
>>> val person = Person("Dmitry", null)
>>> println(person.countryName())
Unknown
```

```
fun Person.countryName() =
    company?.address?.country ?: "Unknown"
```

Оператор «Елвіс» ?:
(оператор об'єднання з null-значенням)

```
fun foo(s: String?) {
    val t: String = s ?: ""
}
```



Оператор «Елвіс» часто використовується з оператором безпечного виклику, щоб повернути не-null-значення, коли об'єкт, метод якого викликається, сам може повернути null.

```
fun strLenSafe(s: String?): Int = s?.length ?: 0

>>> println(strLenSafe("abc"))
3
>>> println(strLenSafe(null))
0
```

Особенно удобным оператор «Элвис» делает то обстоятельство, что такие операции, как `return` и `throw`, действуют как выражения и, следовательно, могут находиться справа от оператора. То есть если значение слева окажется равным `null`, функция немедленно вернет управление или возбудит исключение. Это полезно для проверки предусловий в функции.

Реалізуємо функцію друку поштової наклейки з адресою компанії

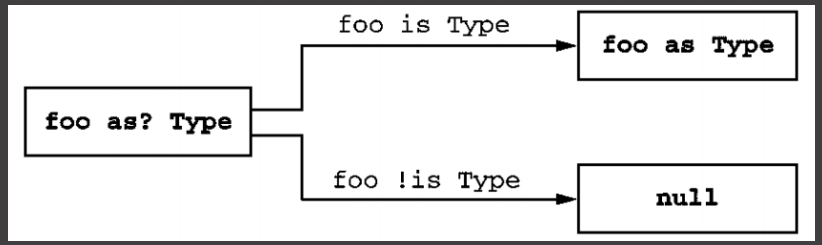
```
class Address(val streetAddress: String, val zipCode: Int,
      val city: String, val country: String)

class Company(val name: String, val address: Address?)

class Person(val name: String, val company: Company?)

fun printShippingLabel(person: Person) {
    val address = person.company?.address
      ?: throw IllegalArgumentException("No address")
    with (address) {
      println(streetAddress)
      println("$zipCode $city, $country")
    }
}
```

Вызовет исключение в отсутствие адреса

```
>>> val address = Address("Elsestr. 47", 80687, "Munich", "Germany")
>>> val jetbrains = Company("JetBrains", address)
>>> val person = Person("Dmitry", jetbrains)

>>> printShippingLabel(person)
Elsestr. 47
80687 Munich, Germany

>>> printShippingLabel(Person("Alexey", null))
java.lang.IllegalArgumentException: No address
```

Если все правильно, функция `printShippingLabel` напечатает наклейку. Если адрес отсутствует, она не просто сгенерирует исключение `NullPointerException` с номером строки, но вернет осмысленное сообщение об ошибке. Если адрес присутствует, текст наклейки будет состоять из адреса, почтового индекса, города и страны. Обратите внимание, как здесь используется функция `with`, которую мы видели в предыдущей главе: она помогает избежать повторения переменной `address` четыре раза подряд.

Безпечне зведення типів: оператор as?



Повертає null, якщо звести значення до вказаного типу неможливо

Оператор безопасного приведения часто используется в сочетании с оператором «Элвис». Например, это может пригодиться для реализации метода equals.

```
class Person(val firstName: String, val lastName: String) {
    override fun equals(o: Any?): Boolean {
      val otherPerson = o as? Person ?: return false

      return otherPerson.firstName == firstName &&
         otherPerson.lastName == lastName
    }

    override fun hashCode(): Int =
      firstName.hashCode() * 37 + lastName.hashCode()
}
```

Проверит тип и вернет false, если указанный тип недопустим

После безопасного приведения типа переменная otherPerson преобразуется к типу Person

Этот шаблон позволяет легко убедиться, что параметр обладает правильным типом, выполнить его приведение, вернуть false, если значение имеет несовместимый тип, – и все в одном выражении. Автоматическое приведение типов также применимо в данном контексте: после проверки типа и отсеивания значения null компилятор знает, что переменная

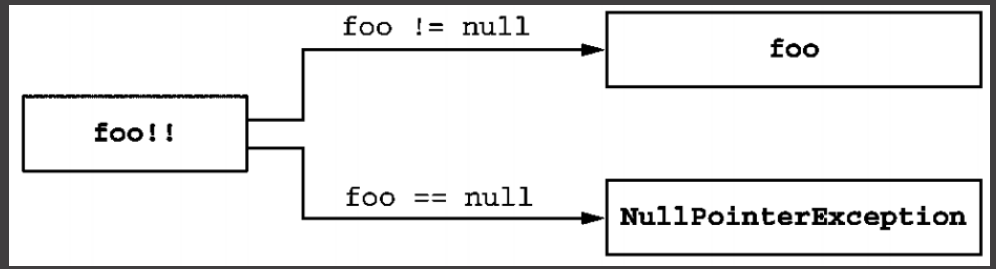
```
>>> val p1 = Person("Dmitry", "Jemerov")
>>> val p2 = Person("Dmitry", "Jemerov")
>>> println(p1 == p2)
true
>>> println(p1.equals(42))
false
```

Перевірка на null: твердження !!

Перетворює будь-яке значення до типу, який не підтримує null-значення.

- При null-значенні виникає виключення.

```
fun ignoreNulls(s: String?) {
    val sNotNull: String = s!!
    println(sNotNull.length)
}
```




```
>>> ignoreNulls(null)
Exception in thread "main" kotlin.KotlinNullPointerException
  at <...>.ignoreNulls(07_NotnullAssertions.kt:2)
```

- Виключення посилається на рядок з твердженням, а не на інструкцію виклику
- Корисно, коли перевірка значення відбувається в іншій функції, звідки передається в поточну ф-цію.
  - Компілятор про це не знає, тому можна застосувати твердження, що значення не може бути null

Практичний приклад: реалізація обробника в Swing

Клас-обробник у Swing має окремі методи для оновлення стану та виконання дії.

- Об'єкт класу CopyRowAction повинен скопіювати в буфер обміну значення з обраного у списку рядка.
- Багато коду опущено, залишено лише той, що перевіряє факт вибору рядка та отримує обраний рядок
- Інтерфейс Action API передбачає, що метод actionPerformed() можна викликати лише якщо поверне true метод isEnabled().

```
class CopyRowAction(val list: JList<String>) : AbstractAction() {
    override fun isEnabled(): Boolean =
      list.selectedValue != null

    override fun actionPerformed(e: ActionEvent) {
      val value = list.selectedValue!!
      // copy value to clipboard
    }
}
```

Метод actionPerformed будет вызван, только если isEnabled вернет «true»

- При небажанні використовувати твердження !! тут можна написати
- **`val value = list.selectedValue ?: return`**
- Якщо **`list.selectedValue`** поверне null, це призведе до дострокового виходу з функції, value ніколи не отримає значення null

Особливості використання оператору !!

Коли використовується твердження !! і в результаті отримується виключення, трасування стеку покаже тільки номер рядка, в якому воно виникло, а не конкретний вираз.

- Для з'ясування, де значення виявилось null краще уникати кількох тверджень !! в одному рядку:

- `person.company!!.address!!.country  // не пишіть такого коду!`

- Неможливо сказати, яке з полів отримало null-значення.

Функція let()

Що робити, якщо треба передати аргумент, який може набути null-значення, у функцію, яка не очікує null-значень?

- Компілятор не дозволить зробити це без перевірки.
- Допоміжна функція є в стандартній бібліотеці Kotlin – let().
- Разом з оператором безпечного виклику дозволяє обчислити вираз, перевірити результат на null та зберегти його в змінній.
- Нехай функція sendEmailTo() приймає 1 рядковий параметр та відправляє лист на цю адресу:

```
fun sendEmailTo(email: String) { /*...*/ }
```

Вы не сможете передать в эту функцию значение типа с поддержкой null:

```
>>> val email: String? = ...
>>> sendEmailTo(email)
ERROR: Type mismatch: inferred type is String? but String was expected
```

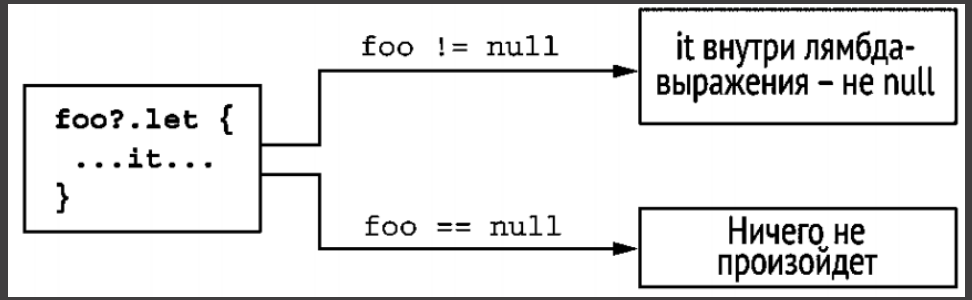
Вы должны явно проверить, что значение не равно null:

```
if (email != null) sendEmailTo(email)
```

Інший шлях – поєднання let() з оператором безпечного виклику

Функція let() перетворює об'єкт виклику в параметр лямбда-виразу.

- Разом з безпечним викликом об'єкт перетворюється з типу з підтримкою null у тип без його підтримки:



- Функція let() буде викликана тільки тоді, коли значення адреси пошти не дорівнює null, тому
- `email?.let { email -> sendEmailTo(email) }`

- Разом з автоматично створеним іменем it синтаксис стане ще коротшим:
- `email?.let { sendEmailTo(it) }`

```
fun sendEmailTo(email: String) {
    println("Sending email to $email")
}
```

```
>>> var email: String? = "yole@example.com"
>>> email?.let { sendEmailTo(it) }
Sending email to yole@example.com
>>> email = null
>>> email?.let { sendEmailTo(it) }
```

Нотація let() особливо зручна, коли потрібно використати значення великого виразу, що не дорівнює null

Не доведеться створювати окрему змінну:

- Порівняємо код:

```
val person: Person? = getTheBestPersonInTheWorld()
if (person != null) sendEmailTo(person.email)
```

```
fun getTheBestPersonInTheWorld(): Person? = null
```

```
getTheBestPersonInTheWorld()?.let { sendEmailTo(it.email) }
```

Эта функция всегда возвращает null, поэтому лямбда-выражение никогда не будет выполнено.

- Для перевірки кількох значень на null можна використати вкладені виклики let().
- Проте код стає складним для розуміння – зазвичай простіше використати простий оператор if.

Властивості з відкладеною ініціалізацією

Багато фреймворків ініціалізують об'єкти у спеціальних методах, які викликаються після створення екземпляру.

- Наприклад, в Android – у методі OnCreate().
- Junit – методи з анотацією @Before.

Залишати без ініціалізації в конструкторі властивості, які не підтримують null, не можна.

- Якщо немає можливості надати таке значення, доведеться використовувати типи з підтримкою null.
- При кожному зверненні до властивості потрібно буде виконувати перевірку на null або твердження !!

```
class MyService {
    fun performAction(): String = "foo"
}
```

```
class MyTest {
    private var myService: MyService? = null
    @Before fun setUp() {
      myService = MyService()
    }
    @Test fun testAction() {
      Assert.assertEquals("foo",
        myService!!.performAction())
    }
}
```

Объявление свойства с типом, поддерживающим null, чтобы инициализировать его значением null

Настоящее значение присваивается в методе setUp

Из-за такого объявления приходится использовать !! или ?

Застосування тверджень !! для доступу до поля з підтримкою null

Это неудобно, особенно когда приходится обращаться к свойству много раз. Чтобы решить эту проблему, можно объявить, что поле `myService` поддерживает *отложенную инициализацию*. Это делается с помощью модификатора `lateinit`.

```
class MyService {
    fun performAction(): String = "foo"
}
```

```
class MyTest {
    private lateinit var myService: MyService

    @Before fun setUp() {
      myService = MyService()
    }

    @Test fun testAction() {
      Assert.assertEquals("foo",
        myService.performAction())
    }
}
```

Объявление свойства с типом, не поддерживающим null, без инициализации

Инициализация в методе setup такая же, как раньше

Обращение к свойству без лишних проверок на null

Обратите внимание, что поля с отложенной инициализацией всегда объявляются как `var`, потому что их значения изменяются за пределами конструктора, в то время как свойства `val` компилируются в финальные поля, которые должны инициализироваться в конструкторе. Поля с отложенной инициализацией больше не требуется инициализировать в конструкторе, даже если их типы поддерживают `null`. Попытка обратиться к такому свойству прежде, чем оно будет инициализировано, возбудит исключение «lateinit property myService has not been initialized» (свойство myService с модификатором lateinit не было инициализировано). Оно четко сообщит, что произошло, и понять его гораздо легче, чем обычное исключение `NullPointerException`.

Розширення типів з підтримкою null

Замість перевірки змінної на рівність null перед викликом методу можна дозволити виклики функцій, де в ролі отримувача виступає null.

- Можливо тільки для функцій-розширень – виклики звичайних класів направляються об'єкту (не null)

Приклад: функції isEmpty() та isBlank(), що розширяють клас String.

- 1) Перевіряє рядок на порожність
- 2) Перевіряє, чи порожній або пробільний рядок

- Корисними можуть бути методи isEmptyOrNull(), isBlankOrNull(), що можуть викликатись з отримувачем типу String?
- Функцію-розширення, оголошену як допускаючу

Функцию-расширение, объявленную допускающей использование `null` в качестве получателя, можно вызывать без оператора безопасного вызова (см. рис. 6.7). Функция сама обработает возможные значения `null`.

```
fun verifyUserInput(input: String?) {
    if (input.isNullOrBlank()) {
      println("Please fill in the required fields")
    }
}

>>> verifyUserInput(" ")
Please fill in the required fields
>>> verifyUserInput(null)
Please fill in the required fields
```

Если вызвать isNullOrBlank с null в качестве получателя, это не приведет к исключению

Коли функція-розширення оголошується для типу з підтримкою null, посилання this може набути значення null.

- Це слід явно перевіряти, оскільки Java не пі дтримує такої можливості.
- Для перевірки аргументів на нерівність null недоречна функція let() – без оператора без печного виклику аргумент лямбда-виразу теж буде підтримувати null

```
>>> val person: Person? = ...
>>> person.let { sendEmailTo(it) }
ERROR: Type mismatch: inferred type is Person? but Person was expected
```

Небезопасный вызов, поэтому «it» может оказаться равным null

```
fun String?.isNullOrBlank(): Boolean =
    this == null || this.isBlank()
```

Расширение для типа String с поддержкой null

Во втором обращении к «this» применяется автоматическое приведение типов

Типові

По умолчанию все типовые параметры функций и классов в Kotlin автоматически поддерживают null. Любой тип, в том числе с поддержкой null, может быть использован как типовой параметр – в этом случае объявления, использующие типовые параметры, должны предусматривать поддержку null, даже если имя параметра T не оканчивается вопросительным знаком. Рассмотрим следующий пример.

За замовчуванням всі типові параметри функцій і класів у Kotlin автоматично підтримують null.

- Можливо тільки для функцій-розширень – виклики звичайних класів направляються об'єкту (не null)

Приклад: функції isEmpty() та isBlank(), що розширяють клас String.

- 1) Перевіряє рядок на порожність
- 2) Перевіряє, чи порожній або пробільний рядок

- К… …ожуть викликатись з отриму-
  в…
- Ф…

```
fun <T> printHashCode(t: T) {
    println(t?.hashCode())
}
```

Обязательно должен использоваться безопасный вызов, поскольку «t» может хранить null

```
>>> printHashCode(null)
null
```

Параметр «T» определяется как тип «Any?»

Для вызова функции printHashCode в листинге 6.13 компилятор определит параметр T как тип Any? с поддержкой null. Соответственно, параметр t может оказаться равным null даже притом, что в имени типа T отсутствует вопросительный знак.

Чтобы запретить параметру принимать значение `null`, необходимо определить соответствующую верхнюю границу. Это не позволит передать `null` в качестве аргумента.

```
fun <T: Any> printHashCode(t: T) {
    println(t.hashCode())
}
```

Теперь «T» не поддерживает null

```
>>> printHashCode(null)
Error: Type parameter bound for 'T' is not satisfied
>>> printHashCode(42)
42
```

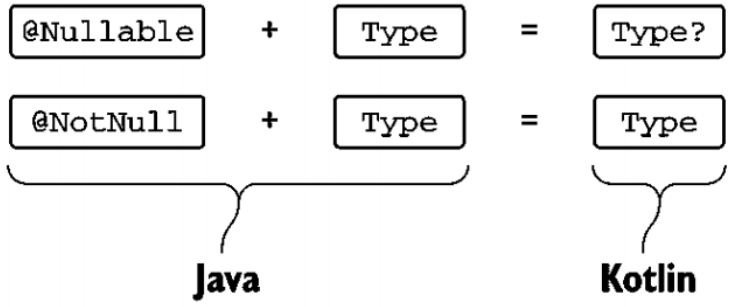
Этот код не скомпилируется: нельзя передать null, поскольку это запрещено

Обратите внимание, что типовые параметры – это единственное исключение из правила «вопросительный знак в конце имени типа разрешает значение `null`, а типы с именами без знака вопроса не допускают `null`».

Допустимість null-значень і Java

null. Что же происходит, когда вы объединяете Kotlin и Java? Сохранится ли безопасность или же вам потребуется проверять каждое значение на null? И есть ли лучшее решение? Давайте выясним это.

Во-первых, как уже упоминалось, код на Java может содержать информацию о допустимости значений null, выраженную с помощью аннотаций. Если эта информация присутствует в коде, Kotlin воспользуется ею. То есть объявление @Nullable String в Java будет представлено в Kotlin как String?, а объявле

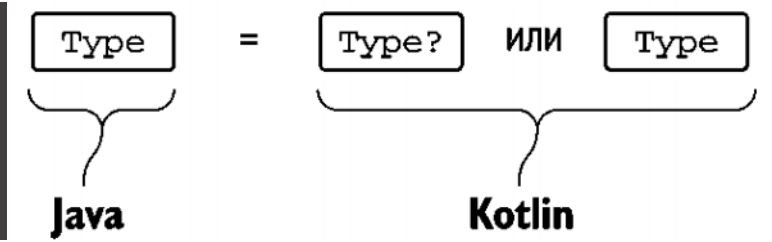


. рис. 6.8).

Kotlin распознает множество разных аннотаций, описывающих допустимость значения null, в том числе аннотации из стандарта JSR-305 (javax.annotation), из Android (android.support.annotation) и те, что поддерживаются инструментами компании JetBrains (org.jetbrains.annotations). Интересно, а что происходит, когда аннотация отсутствует? В этом случае тип Java становится *платформенным типом* в Kotlin.

Платформні типи

Платформенный тип – это тип, для которого Kotlin не может найти информацию о допустимости `null`. С ним можно работать как с типом, допускающим `null`, или как с типом, не допускающим `null` (см. рис. 6.9).



Это означает, что, точно как в Java, вы несете полную ответственность за операции с этим типом. Компилятор разрешит вам делать всё. Он также не станет напоминать об избыточных безопасных операциях над такими значениями (как он делает обычно, встречая безопасные операции над значениями, которые не могут быть `null`). Если известно, что значение может быть равно `null`, вы можете сравнить его с `null` перед использованием. Если известно, что оно не может быть равно `null`, вы сможете использовать его напрямую. Но, как и в Java, если вы ошибетесь, то получите исключение `NullPointerException` во время использования.

Припустимо, що клас Person оголошено в Java

```
/* Java */
public class Person {
    private final String name;

    public Person(String name) {
      this.name = name;
    }

    public String getName() {
      return name;
    }
}
```

Может ли метод getName вернуть значение null? Компилятор Kotlin ничего не знает о допустимости null для типа String в данном случае, поэтому вам придется справиться с этим самостоятельно. Если вы уверены, что имя не будет равно null, можете разыменовать его в обычном порядке (как в Java) без дополнительных проверок, но будьте готовы столкнуться с исключением.

```
fun yellAt(person: Person) {
    println(person.name.toUpperCase() + "!!!")
}

>>> yellAt(Person(null))
java.lang.IllegalArgumentException: Parameter specified as non-null
is null: method toUpperCase, parameter $receiver
```

Получатель метода toUpperCase() – поле person.name – равен null, поэтому будет возбуждено исключение

Обратите внимание, что вместо обычного исключения `NullPointerException` вы получите более подробное сообщение об ошибке: что метод `toUpperCase` не может вызываться для получателя, равного `null`.

На самом деле для общедоступных функций Kotlin компилятор генерирует проверки для каждого параметра (и для получателя в том числе), который не должен принимать значения `null`. Поэтому попытка вызвать такую функцию с неверными аргументами сразу закончится исключением. Обратите внимание, что проверка значения выполняется сразу же при вызове функции, а не во время использования параметра. Это гарантирует обнаружение некорректных вызовов на ранних стадиях и избавляет от непонятных исключений, возникающих после того, как значение `null` прошло через несколько функций в разных слоях кода.

Другой вариант – интерпретация типа значения, возвращаемого методом `getName()`, как допускающего значение `null`, и обращение к нему с помощью безопасных вызовов.

```
fun yellAtSafe(person: Person) {
    println((person.name ?: "Anyone").toUpperCase() + "!!!")
}

>>> yellAtSafe(Person(null))
ANYONE!!!
```

В этом примере значения `null` обрабатываются должным образом, и во время выполнения никаких исключений не возникнет.

Будьте осторожны при работе с Java API. Большинство библиотек не использует аннотаций, поэтому, если вы будете интерпретировать все типы как не поддерживающие `null`, это может привести к ошибкам. Во избежание ошибок читайте документацию с описанием методов Java, которые используете (а если нужно, то и их реализацию), выясните, когда они могут возвращать `null`, и добавьте проверку для таких методов.

Вы не можете объявить переменную платформенного типа в Kotlin – эти типы могут прийти только из кода на Java. Но вы можете встретить их в сообщениях об ошибках или в IDE:

```
>>> val i: Int = person.name
ERROR: Type mismatch: inferred type is String! but Int was expected
```

С помощью нотации `String!` компилятор Kotlin обозначает платформенные типы, пришедшие из кода на Java. Вы не можете использовать этого синтаксиса в своем коде, и чаще всего этот восклицательный знак не связан с источником проблемы, поэтому его можно игнорировать. Он просто подчеркивает, что допустимость значения `null` не была установлена.

Как мы уже сказали, платформенные типы можно интерпретировать как угодно – как допускающие или как не допускающие `null`, – поэтому следующие объявления допустимы:

```
>>> val s: String? = person.name
>>> val s1: String = person.name
```

Получатель метода toUpperCase() – поле person.name – равен null, поэтому будет возбуждено исключение...

...или как не допускающее

Как и при вызове методов, в этом случае вы должны быть уверены, что понимаете, где может появиться `null`. При попытке присвоить пришедшее из Java значение `null` переменной Kotlin, которая не может хранить `null`, вы понимаете, где получите исключение в месте вызова.

Наслідування

Переопределяя Java-метод в коде на Kotlin, вы можете выбрать, будут ли параметры и возвращаемое значение поддерживать `null` или нет. К примеру, давайте взглянем на интерфейс `StringProcessor` в Java.

```
/* Java */
interface StringProcessor {
    void process(String value);
}
```

```
class StringPrinter : StringProcessor {
    override fun process(value: String) {
      println(value)
    }
}
```

```
class NullableStringPrinter : StringProcessor {
    override fun process(value: String?) {
      if (value != null) {
        println(value)
      }
    }
}
```

Обратите внимание, что при реализации методов классов или интерфейсов Java важно правильно понимать, когда можно получить `null`. Поскольку методы могут быть вызваны из кода не на Kotlin, компилятор сгенерирует проверки, что значение не может быть равно `null`, для каждого параметра, объявление которого запрещает присваивание `null`. Если Java-код передаст в метод значение `null`, то сработает проверка, и вы получите исключение, даже если ваша реализация вообще не обращается к значению параметра.

Примітивні та інші базові типи

Kotlin не различает примитивных типов и типов-оберток. Вы всегда используете один и тот же тип (например, Int):

```
val i: Int = 1
val list: List<Int> = listOf(1, 2, 3)
```

Это удобно. Более того, такой подход позволяет вызывать методы на значениях числовых типов. Для примера рассмотрим нижеследующий фрагмент – в нём используется функция coerceIn из стандартной библиотеки, ограничивающая значение указанным диапазоном:

```
fun showProgress(progress: Int) {
    val percent = progress.coerceIn(0, 100)
    println("We're ${percent}% done!")
}

>>> showProgress(146)
We're 100% done!
```

Во время выполнения числовые типы представлены наиболее эффективным способом. В большинстве случаев – для переменных, свойств, параметров и возвращаемых значений – тип `Int` в Kotlin компилируется в примитивный тип `int` из Java. Единственный случай, когда это невозможно, – обобщенные классы, например коллекции. Примитивный тип, указанный в качестве аргумента типа обобщенного класса, компилируется в соответствующий тип-обертку в Java. Поэтому если в качестве аргумента типа коллекции указан `Int`, то коллекция будет хранить экземпляры соответствующего типа-обертки `java.lang.Integer`.

- *целочисленные типы* – `Byte`, `Short`, `Int`, `Long`;
- *числовые типы с плавающей точкой* – `Float`, `Double`;
- *символьный тип* – `Char`;
- *логический тип* – `Boolean`.

Такие Kotlin-типы, как `Int`, за кулисами могут компилироваться в соответствующие примитивные типы Java, поскольку значения обоих типов не могут хранить ссылку на `null`. Обратное преобразование работает аналогично: при использовании Java-объявлений в Kotlin примитивные типы становятся типами, не допускающими `null` (а не платформенными типами), поскольку они не могут содержать значения `null`. Теперь рассмотрим те же типы, но уже допускающие значение `null`.

Примітивні типи з підтримкою null

Int?, Boolean? та ін.

Kotlin-типы с поддержкой `null` не могут быть представлены в Java как примитивные типы, поскольку в Java значение `null` может храниться только в переменных ссылочных типов. Это означает, что всякий раз, когда в коде на Kotlin используется версия простого типа, допускающая значение `null`, она компилируется в соответствующий тип-обертку.

Чтобы увидеть такие типы в действии, давайте вернемся к примеру в начале этой книги и вспомним объявление класса `Person`. Класс описывает человека, чье имя всегда известно и чей возраст может быть указан или нет. Добавим функцию, которая проверяет, является ли один человек старше другого.

```
data class Person(val name: String,
        val age: Int? = null) {

    fun isOlderThan(other: Person): Boolean? {
      if (age == null || other.age == null)
        return null
      return age > other.age
    }
}

>>> println(Person("Sam", 35).isOlderThan(Person("Amy", 42)))
false
>>> println(Person("Sam", 35).isOlderThan(Person("Jane")))
null
```

Обратите внимание, что здесь применяются обычные правила работы со значением `null`. Нельзя просто взять и сравнить два значения типа `Int?`, поскольку одно из них может оказаться `null`. Вместо этого вам нужно убедиться, что оба значения не равны `null`, и после этого компилятор позволит работать с ними как обычно.

Значение свойства `age`, объявленного в классе `Person`, хранится как `java.lang.Integer`. Но эта деталь имеет значение только тогда, когда вы работаете с этим классом в Java. Чтобы выбрать правильный тип в Kotlin, нужно понять только то, допустимо ли присваивать значение `null` переменной или свойству.

Как уже упоминалось, обобщенные классы – это ещё одна ситуация, когда на сцену выходят оберточные типы. Если в качестве аргумента типа указан простой тип, Kotlin будет использовать обертку для данного типа. Например, следующее объявление создаст список значений типа `Integer`, даже если вы не указывали, что тип допускает значение `null`, и не использовали самого значения `null`:

```
val listOfInts = listOf(1, 2, 3)
```

Это происходит из-за способа реализации обобщенных классов в виртуальной машине Java. JVM не поддерживает использования примитивных типов в качестве аргументов типа, поэтому обобщенные классы (как в Java, так и в Kotlin) всегда должны использовать типы-обертки. Следовательно, когда требуется обеспечить эффективное хранение больших коллекций простых типов, то приходится использовать сторонние библиотеки (такие как Trove4J, `http://trove.starlight-systems.com`), поддерживающие такие коллекции, или хранить их в массивах. Мы подробно обсудим массивы в конце этой главы.

Числові перетворення

из одного типа в другой, даже когда другой тип охватывает более широкий диапазон значений. Например, в Kotlin следующий код не будет компилироваться:

```
val i = 1
val l: Long = i          <— Ошибка: несоответствие типов
```

Вместо этого нужно применить явное преобразование:

```
val i = 1
val l: Long = i.toLong()
```

Функции преобразования определены для каждого простого типа (кроме `Boolean`): `toByte()`, `toShort()`, `toChar()` и т. д. Функции поддерживают преобразование в обоих направлениях: расширение меньшего типа к большему, как `Int.toLong()`, и усечение большего типа до меньшего, как `Long.toInt()`.

Kotlin делает преобразование явным, чтобы избежать неприятных неожиданностей, особенно при сравнении обернутых значений. Метод `equals` для двух обернутых значений проверяет тип обертки, а не только хранящееся в ней значение. Так, выражение `new Integer(42).equals(new Long(42))` в Java вернет `false`. Если бы Kotlin поддерживал неявные преобразования, вы могли бы написать что-то вроде этого:

```
val x = 1                          <— Переменная типа Int
val list = listOf(1L, 2L, 3L)      <— Список значений типа Long
x in list                          <— Если бы Kotlin поддерживал неявные преобразования типов, это выражение вернуло бы false
```

Вопреки всем ожиданиям эта функция вернула бы `false`. Поэтому строка `x in list` в этом примере не скомпилируется. Kotlin требует явного преобразования типов, поэтому сравнивать можно только значения одного типа:

```
>>> val x = 1
>>> println(x.toLong() in listOf(1L, 2L, 3L))
true
```

Если вы одновременно используете различные числовые типы в коде и хотите избежать неожиданного поведения – явно преобразуйте переменные.

Обратите внимание, что при записи числового литерала обычно не нужно использовать функции преобразования. Допускается использовать специальный синтаксис для явного обозначения типа константы, `42L` или `42.0f`. Но даже если его не использовать, компилятор автоматически применит к числовому литералу необходимое преобразование для инициализации переменной известного типа или передачи его в качестве аргумента функции. Кроме того, арифметические операторы перегружены для всех соответствующих числовых типов. Например, следующий код работает правильно без каких-либо явных преобразований:

```
fun foo(l: Long) = println(l)

>>> val b: Byte = 1
>>> val l = b + 1L
>>> foo(42)
42
```

Значение константы получит корректный тип

Оператор + работает с аргументами типа Byte и Long

Компилятор интерпретирует 42 как значение типа Long

Обратите внимание, что при переполнении арифметические операторы в Kotlin действуют так же, как в Java, – Kotlin не привносит накладных расходов для проверки переполнения.

Кореневі типи

Подобно тому, как тип `Object` является корнем иерархии классов в Java, тип `Any` – это супертип всех типов в Kotlin, не поддерживающих `null`. Но в Java тип `Object` – это супертип для всех ссылочных типов, а примитивные типы не являются частью его иерархии. Это означает, что когда требуется экземпляр `Object`, то для представления значений примитивных типов нужно использовать типы-обертки, такие как `java.lang.Integer`. В Kotlin тип `Any` – супертип для всех типов, в том числе для примитивных, таких как `Int`.

Так же как в Java, присваивание примитивного значения переменной типа `Any` вызывает автоматическую упаковку значения:

```
val answer: Any = 42
```

Значение 42 будет упаковано, поскольку тип Any – ссылочный

Обратите внимание, что тип `Any` не поддерживает значения `null`, поэтому переменная типа `Any` не может хранить `null`. Если нужна переменная, способная хранить любое допустимое значение в Kotlin, в том числе `null`, используйте тип `Any?`.

На уровне реализации тип `Any` соответствует типу `java.lang.Object`. Тип `Object`, используемый в параметрах и возвращаемых значениях методов Java, рассматривается в Kotlin как тип `Any`. (Точнее, он рассматри-

Тип Unit: тип «відсутнього» значення

Тип `Unit` играет в Kotlin ту же роль, что и `void` в Java. Он может использоваться в качестве типа возвращаемого значения функции, которая не возвращает ничего интересного:

```
fun f(): Unit { ... }
```

Синтаксически это определение равноценно следующему, где отсутствует объявление типа тела функции:

```
fun f() { ... }
```

Явное объявление типа Unit опущено

В большинстве случаев вы не заметите разницы между типами `Unit` и `void`. Если ваша Kotlin-функция объявлена как возвращающая тип `Unit` и она не переопределяет обобщенную функцию, она будет скомпилирована в старую добрую функцию `void`. Если вы переопределяете её в Java, то переопределяющая функция просто должна возвращать `void`.

Тогда чем тип `Unit` в Kotlin отличается от типа `void` в Java? В отличие от `void`, тип `Unit` – это полноценный тип, который может использоваться как аргумент типа. Существует только один экземпляр данного типа – он тоже называется `Unit` и возвращается *неявно*. Это полезно при переопределении функции с обобщенным параметром, чтобы заставить её возвращать значение типа `Unit`:

```
interface Processor<T> {
    fun process(): T
}
```

```
class NoResultProcessor : Processor<Unit> {
    override fun process() {
      // сделать что-то
    }
}
```

Возвращает значение типа Unit, но в объявлении это не указано

Не требуется писать инструкцию return

Сигнатура интерфейса требует, чтобы функция `process` возвращала значение, а поскольку тип `Unit` не имеет значения, вернуть его из метода не проблема. Но вам не нужно явно писать инструкцию `return` в функции `NoResultProcessor.process`, потому что `return Unit` неявно добавляется

Сравните это с Java, где ни одно из решений проблемы указания отсутствия аргумента типа не выглядит так элегантно, как в Kotlin. Один из вариантов – использовать отдельные интерфейсы (такие как `Callable` и `Runnable`) для представления элементов, возвращающих и не возвращающих значения. Другой заключается в использовании специального типа `java.lang.Void` в качестве параметра типа. Если вы выбрали второй вариант, вам всё равно нужно добавить инструкцию `return null` для возвращения единственно возможного значения этого типа, поскольку если тип возвращаемого значения не `void`, то вы всегда должны использовать оператор `return`.

Вас может удивить, почему мы выбрали другое имя для `Unit` и не назвали его `Void`. Имя `Unit` традиционно используется в функциональных языках и означает «единственный экземпляр», а это именно то, что отличает тип `Unit` в Kotlin от `void` в Java. Мы могли бы использовать обычное имя `Void`, но в Kotlin есть тип `Nothing`, выполняющий совершенно другую функцию. Наличие двух типов с именами `Void` и `Nothing` («пустота» и «ничто») сбивало бы с толку, поскольку значения этих слов очень похожи. Так что же это за тип – `Nothing`? Давайте выясним.

Тип Nothing: функція, що не завершується

Для некоторых функций в Kotlin понятие возвращаемого значения просто не имеет смысла, поскольку они никогда не возвращают управления. Например, во многих библиотеках для тестирования есть функция `fail`, которая генерирует исключение с указанным сообщением и заставляет текущий тест завершиться неудачей. Функция с бесконечным циклом также никогда не завершится.

При анализе кода, вызывающего такую функцию, полезно знать, что она не возвращает управления. Чтобы выразить это, в Kotlin используется специальный тип возвращаемого значения `Nothing`:

```
fun fail(message: String): Nothing {
    throw IllegalStateException(message)
}
>>> fail("Error occurred")
java.lang.IllegalStateException: Error occurred
```

Тип `Nothing` не имеет значений, поэтому его имеет смысл использовать только в качестве типа возвращаемого значения функции или аргумента типа для обозначения типа возвращаемого значения обобщенной функции. Во всех остальных случаях объявление переменной, в которой нельзя сохранить значение, не имеет смысла.

Обратите внимание, что функции, возвращающие Nothing, могут использоваться справа от оператора «Элвис» для проверки предусловий:

```
val address = company.address ?: fail("No address")
println(address.city)
```

Этот пример показывает, почему наличие Nothing в системе типов крайне полезно. Компилятор знает, что функция с таким типом не вернет управления, и использует эту информацию при анализе кода вызова функции. В предыдущем примере компилятор сообщит, что тип поля address не допускает значений null – потому что ветка, где значение равно null, всегда возбуждает исключение.

Масиви та колекції

может ли коллекция хранить значения null, чем знать, может ли переменная хранить null. К счастью, Kotlin позволяет указать допустимость null в аргументах типов. Как имя типа переменной может заканчиваться символом ?, свидетельствующим о допустимости значения null, так и аргумент типа может быть помечен таким образом. Чтобы понять суть, рассмотрим функцию, которая читает строки из файла и пытается представить каждую строку как число.

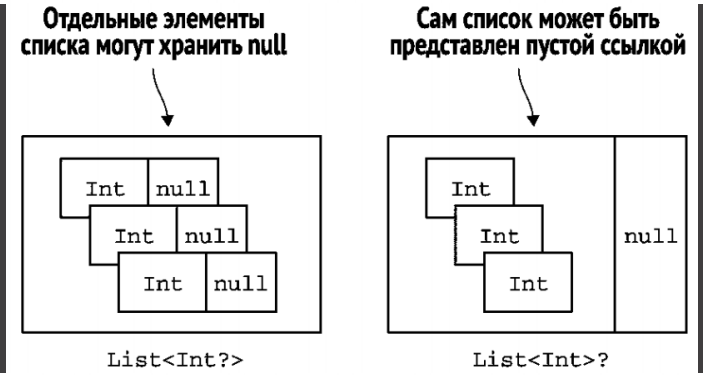
```
fun readNumbers(reader: BufferedReader): List<Int?> {
    val result = ArrayList<Int?>()
    for (line in reader.lineSequence()) {
      try {
        val number = line.toInt()ч
        result.add(number)
      }
      catch(e: NumberFormatException) {
        result.add(null)
      }
    }
    return result
}
```

Создание списка значений типа Int с поддержкой null

Добавление в список целочисленного значения (не равного null)

Добавление значения null в список, поскольку текущая строка не может быть преобразована в число

Обратите внимание, что допустимость значения `null` для самой переменной отличается от допустимости значения `null` для типа, который используется в качестве аргумента типа. Разница между списком, поддерживающим элементы типа `Int` и `null`, и списком элементов `Int`, который сам может оказаться пустой ссылкой `null`, показана на рис. 6.10.



В первом случае сам список не может оказаться пустой ссылкой, но его элементы могут хранить `null`. Переменная второго типа может содержать значение `null` вместо ссылки на экземпляр списка, но элементы в списке гарантированно не будут хранить `null`.

В другом контексте вы можете захотеть объявить переменную, содержащую пустую ссылку на список с элементами, которые могут иметь значения `null`. В Kotlin такой тип записывается с двумя вопросительными знаками: `List<Int?>?`. При этом вам придется выполнять проверки на `null`

Чтобы понять, как работать со списком элементов, способных хранить `null`, напишем функцию для сложения всех корректных чисел и подсчета некорректных чисел отдельно.

```
fun addValidNumbers(numbers: List<Int?>) {
    var sumOfValidNumbers = 0
    var invalidNumbers = 0
    for (number in numbers) {
      if (number != null) {
        sumOfValidNumbers += number
      } else {
        invalidNumbers++
      }
    }
    println("Sum of valid numbers: $sumOfValidNumbers")
    println("Invalid numbers: $invalidNumbers")
}
```

Чтение из списка значения, которое может оказаться равным null

Проверка значения на null

```
>>> val reader = BufferedReader(StringReader("1\nabc\n42"))
>>> val numbers = readNumbers(reader)
>>> addValidNumbers(numbers)
Sum of valid numbers: 43
Invalid numbers: 1
```

Обработка коллекции значений, которые могут быть равны null, и последующая фильтрация таких элементов – очень распространенная операция. Поэтому для её выполнения в Kotlin есть стандартная функция filterNotNull. Вот как использовать её для упрощения предыдущего примера.

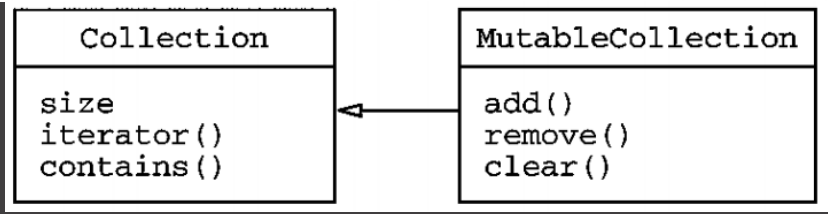
```
fun addValidNumbers(numbers: List<Int?>) {
    val validNumbers = numbers.filterNotNull()
    println("Sum of valid numbers: ${validNumbers.sum()}")
    println("Invalid numbers: ${numbers.size - validNumbers.size}")
}
```

Конечно, фильтрация тоже влияет на тип коллекции. Коллекция validNumbers имеет тип List<Int>, потому что фильтрация гарантирует, что коллекция не будет содержать значений null.

Змінювані та незмінювані колекції

Важная черта, отличающая коллекции в Kotlin от коллекций в Java, – разделение интерфейсов, открывающих доступ к данным в коллекции только для чтения и для изменения. Это разделение начинается с базового интерфейса коллекций – `kotlin.collections.Collection`. С помощью этого интерфейса можно выполнить обход элементов коллекции, узнать её размер, проверить наличие определенного элемента и выполнить другие операции чтения данных из коллекции. Но в этом интерфейсе отсутствуют методы добавления или удаления элементов.

Чтобы получить возможность изменения данных в коллекции, используйте интерфейс `kotlin.collections.MutableCollection`. Он наследует интерфейс `kotlin.collections.Collection`, добавляя к нему методы для добавления и удаления элементов, очистки коллекции и т. д. На рис. 6.11 показаны основные методы, присутствующие в этих двух интерфейсах.



Как правило, вы везде должны использовать интерфейсы с доступом только для чтения. Применяйте изменяемые варианты, только если собираетесь изменять коллекцию.

Такое разделение интерфейсов позволяет быстрее понять, что происходит с данными в программе. Если функция принимает параметр типа `Collection`, но не `MutableCollection`, можно быть уверенным, что она не изменяет коллекцию и будет только читать данные из нее. Но если функция требует передачи аргумента `MutableCollection`, можно предположить, что она собирается изменять данные. Если у вас есть коллекция, которая хранит внутреннее состояние вашего компонента, то вам может понадобиться сделать копию этой коллекции перед передачей в такую функцию (этот шаблон обычно называют *защитным копированием*).

Например, в следующем примере видно, что функция `copyElements` будет изменять целевую коллекцию, но не исходную.

```
fun <T> copyElements(source: Collection<T>,
         target: MutableCollection<T>) {
    for (item in source) {
      target.add(item)
    }
}

>>> val source: Collection<Int> = arrayListOf(3, 5, 7)
>>> val target: MutableCollection<Int> = arrayListOf(1)
>>> copyElements(source, target)
>>> println(target)
[1, 3, 5, 7]
```

Цикл по всем элементам исходной коллекции

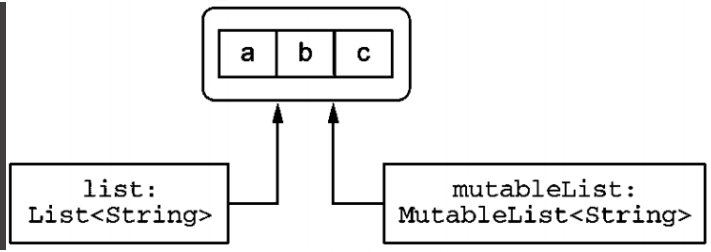
Добавление элементов в изменяемую целевую коллекцию

Вы не сможете передать в аргументе `target` переменную с типом коллекции, доступной только для чтения, даже если она ссылается на изменяемую коллекцию:

```
>>> val source: Collection<Int> = arrayListOf(3, 5, 7)
>>> val target: Collection<Int> = arrayListOf(1)
>>> copyElements(source, target)
Error: Type mismatch: inferred type is Collection<Int>
but MutableCollection<Int> was expected
```

Ошибочный аргумент «target»

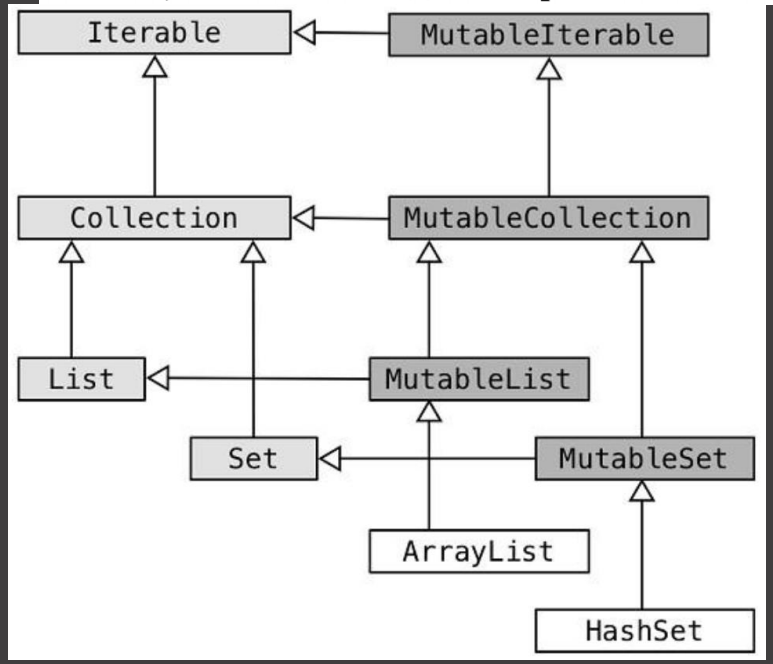
Самое главное, что нужно помнить при работе с интерфейсами коллекций, доступных только для чтения, – они *необязательно неизменяемы*[1]. Если вы работаете с переменной-коллекцией, интерфейс которой дает доступ только для чтения, она может оказаться лишь одной из нескольких ссылок на одну и ту же коллекцию. Другие ссылки могут иметь тип изменяемого интерфейса, как показано на рис. 6.12.



Вызывая код, хранящий другую ссылку на вашу коллекцию, или запуская его параллельно, вы все ещё можете столкнуться с ситуацией, когда коллекция меняется под воздействием другого кода, пока вы работаете с ней. Это приводит к появлению исключения `ConcurrentModificationException` и другим проблемам. Поэтому важно понимать, что *коллекции с доступом только для чтения не всегда потокобезопасны*. Если вы работаете с данными в многопоточной среде, убедитесь, что ваш код правильно синхронизирует доступ к данным или использует структуры данных, поддерживаю-

Колекції Kotlin та мова програмування Java

го интерфейса коллекции Java. При переходе между Kotlin и Java никакого преобразования не происходит, и нет необходимости ни в создании оберток, ни в копировании данных. Но для каждого интерфейса Java-коллекций в Kotlin существуют два представления: только для чтения и для чтения/записи, как можно видеть на рис. 6.13.



Интерфейсы с доступом только для чтения
Интерфейсы с доступом для чтения/записи
Классы Java

Все интерфейсы коллекций на рис. 6.13 объявлены в Kotlin. Базовая структура интерфейсов для чтения и изменения коллекций в Kotlin выстроена параллельно иерархии интерфейсов Java-коллекций в пакете `java.util`. Кроме того, каждый интерфейс, меняющий коллекцию, наследует соответствующий интерфейс только для чтения. Интерфейсы коллекций с доступом для чтения/записи непосредственно соответствуют Java-интерфейсам в пакете `java.util`, тогда как интерфейсы только для чтения лишены любых методов, которые могли бы изменить коллекцию.

Для демонстрации того, как стандартные классы Java выглядят с точки зрения Kotlin, на рис. 6.13 также показаны коллекции `java.util.ArrayList` и `java.util.HashSet`. Язык Kotlin рассматривает их так, словно они наследуют интерфейсы `MutableList` и `MutableSet` соответственно. Здесь не представлены другие реализации из Java-библиотеки (`LinkedList`, `SortedSet` и т. д.), но с точки зрения Kotlin они имеют схожие супертипы. Таким образом, вы получаете не только совместимость, но так же четкое разделение интерфейсов с доступом только для чтения и для чтения/записи.

В дополнение к коллекциям в Kotlin есть класс `Map` (который не наследует ни `Collection`, ни `Iterable`) в двух различных вариантах: `Map` и `MutableMap`. В табл. 6.1 показаны функции, которые можно использовать для создания коллекций различных типов.

Таблица 6.1. Функции для создания коллекций

Тип коллекции	Только для чтения	Изменяемая коллекция
List	`listOf`	`mutableListOf, arrayListOf`
Set	`setOf`	`mutableSetOf, hashSetOf, linkedSetOf, sortedSetOf`
Map	`mapOf`	`mutableMapOf, hashMapOf, linkedMapOf, sortedMapOf`

Обратите внимание, что функции `setOf()` и `mapOf()` возвращают экземпляры классов из стандартной библиотеки Java (по крайней мере, в Kotlin 1.0), которые на самом деле изменяемы[2]. Но не стоит полагаться на это: вполне возможно, что в будущих версиях Kotlin функции `setOf` и `mapOf` будут использовать по-настоящему неизменяемые реализации классов.

Когда нужно вызвать метод Java и передать ему коллекцию, это можно сделать непосредственно, без каких-либо промежуточных шагов. Например, если у вас есть метод Java, принимающий экземпляр `java.util.Collection`, вы можете передать в нем любой объект `Collection` или `MutableCollection`.

Это сильно влияет на изменяемость коллекций. Поскольку в Java нет различий между коллекциями только для чтения и для чтения/записи, Java-код *сможет изменить коллекцию*, даже если в Kotlin она объявлена как доступная только для чтения. Компилятор Kotlin не в состоянии полностью проанализировать, что происходит с коллекцией на стороне Java-кода, и не имеет никакой возможности отклонить передачу коллекции, доступной только для чтения, в модифицирующий её код на Java. Например, следующие два фрагмента образуют компилируемую многоязыковую программу Java/Kotlin:

```java
/* Java */
// CollectionUtils.java
public class CollectionUtils {
    public static List<String> uppercaseAll(List<String> items) {
      for (int i = 0; i < items.size(); i++) {
        items.set(i, items.get(i).toUpperCase());
      }
      return items;
    }
}
```

```kotlin
// Kotlin
// collections.kt
fun printInUppercase(list: List<String>) {
    println(CollectionUtils.uppercaseAll(list))
    println(list.first())
}
```

Объявление параметра как доступного только для чтения

Вызов Java-функции, которая изменяет коллекцию

Показывает, что коллекция изменилась

```
>>> val list = listOf("a", "b", "c")
>>> printInUppercase(list)
[A, B, C]
A
```

Поэтому, если вы пишете функцию Kotlin, которая принимает коллекцию и передает её Java-коду, *только вы отвечаете за объявление правильного типа параметра* в зависимости от того, изменяет вызываемый Java-код коллекцию или нет.

Обратите внимание, что этот нюанс также касается коллекций, элементы которых не могут содержать `null`. Если передать такую коллекцию в Java-метод, он вполне может поместить в неё значение `null` – Kotlin не может запретить или просто обнаружить такую ситуацию без ущерба для производительности. Поэтому, передавая коллекции в Java-код, который может изменить их, принимайте меры предосторожности – тогда типы Kotlin правильно отразят все возможности изменения коллекции.

Колекції як платформні типи

Вернувшись к обсуждению значения `null` в начале этой главы, вы наверняка вспомните, что типы, объявленные в Java-коде, рассматриваются в Kotlin как *платформенные типы*. Для платформенных типов Kotlin не имеет информации о поддержке значения `null`, поэтому компилятор позволяет обращаться с ними как с поддерживающими или не поддерживающими `null`. Таким же образом типы переменных и коллекций, объявленные в Java, в языке Kotlin рассматриваются как платформенные типы. Коллекция платформенного типа фактически представляет собой коллекцию с неизвестным статусом изменяемости – код на Kotlin может считать её доступной для чтения/записи или только для чтения. Обычно это неважно, поскольку все операции, которые вы можете захотеть выполнить, просто работают.

Разница становится важной при переопределении или реализации метода Java, в сигнатуре которого есть тип коллекции. Как и в случае с поддержкой `null` для платформенных типов, здесь только вы решаете, какой тип Kotlin использовать для представления типа Java в переопределяемом или реализуемом методе.

В этом случае вам нужно принять несколько решений, каждое из которых отразится на типе параметра в Kotlin:

- Может ли сама коллекция оказаться пустой ссылкой `null`?
- Может ли хотя бы один из ее элементов оказаться значением `null`?
- Будет ли ваш метод изменять коллекцию?

Чтобы понять разницу, рассмотрим следующие случаи. В первом примере интерфейс Java представляет объект, обрабатывающий текст в файле.

```
/* Java */
interface FileContentProcessor {
    void processContents(File path,
      byte[] binaryContents,
      List<String> textContents);
}
```

В Kotlin-реализации этого интерфейса нужно принять во внимание следующие соображения:

- Ссылка на список может оказаться пустой, поскольку существуют двоичные файлы, которые нельзя представить в виде текста.
- Элементы в списке не могут хранить `null`, поскольку строки в файле никогда не имеют значения `null`.
- Список доступен только для чтения, поскольку представляет неизменяемое содержимое файла.

Реалізація інтерфейсу FileContentProcessor у Kotlin

```
class FileIndexer : FileContentProcessor {
    override fun processContents(path: File,
      binaryContents: ByteArray?,
      textContents: List<String>?) {

      // ...
    }
}
```

Сравните её с другим интерфейсом. Здесь реализация интерфейса предусматривает преобразование некоторых данных из текстовой формы в список объектов (с добавлением новых объектов в выходной список), сообщает об ошибках, обнаруженных при анализе, и добавляет текст сообщений в отдельный список.

```
/* Java */
interface DataParser<T> {
    void parseData(String input,
      List<T> output,
      List<String> errors);
}
```

Здесь нужно учесть другие соображения:

- Список `List<String>` не может быть пустой ссылкой, поскольку вызывающий код всегда должен получать сообщения об ошибках.
- Среди элементов списка может оказаться значение `null`, поскольку не все элементы в выходном списке будут иметь связанные с ними сообщения об ошибках.
- Список `List<String>` будет изменяемым, поскольку реализация должна добавлять в него элементы.

Реалізація інтерфейсу DataParser мовою Kotlin

```
class PersonParser : DataParser<Person> {
    override fun parseData(input: String,
      output: MutableList<Person>,
      errors: MutableList<String?>) {

      // ...
    }
}
```

Обратите внимание, как один и тот же Java-тип `List<String>` представлен в Kotlin двумя различными типами: `List<String>?` (список строк, который может быть представлен пустой ссылкой) в одном случае и `MutableList<String?>` (изменяемый список строк с возможностью хранения `null`) в другом. Чтобы сделать правильный выбор, нужно точно знать контракт, которому должен следовать интерфейс или класс Java. Обычно это легко понять из назначения вашей реализации.

Масиви об’єктів та примітивних типів

Массив в Kotlin – это класс с параметром типа, где тип элемента определяется аргументом типа.

Создать массив в Kotlin можно следующими способами:

- Функция `arrayOf` создает массив с элементами, соответствующими аргументам функции.
- Функция `arrayOfNulls` создает массив заданного размера, где все элементы равны `null`. Конечно, эту функцию можно использовать лишь для создания массивов, допускающих хранение `null` в элементах.
- Конструктор класса `Array` принимает размер массива и лямбда-выражение, после чего инициализирует каждый элемент с помощью этого лямбда-выражения. Так можно инициализировать массив, который не поддерживает значения `null` в элементах, не передавая всех элементов непосредственно.

Приклад: створимо масив рядків від “a” до “z”

```
>>> val letters = Array<String>(26) { i -> ('a' + i).toString() }
>>> println(letters.joinToString(""))
abcdefghijklmnopqrstuvwxyz
```

Лямбда-выражение принимает индекс элемента массива и возвращает значение, которое будет помещено в массив с этим индексом. Здесь значение вычисляется путем сложения индекса с кодом символа `"a"` и преобразованием результата в строку. Тип элемента массива показан для ясности – в реальном коде его можно опустить, поскольку компилятор определит его самостоятельно.

Одна из самых распространенных причин создания массивов в Kotlin – необходимость вызова метода Java, принимающего массив, или функции Kotlin с параметром типа `vararg`. Чаще всего в таких случаях данные уже хранятся в коллекции, и вам просто нужно преобразовать их в массив. Сделать это можно с помощью метода `toTypedArray`.

Дякую за увагу!

Ваші запитання?