# Алгоритмизация и программирование

## 4. Списки и строки

Глухих Михаил Игоревич
mailto: glukhikh@mail.ru

# Составные типы данных

- Включают в себя вложенные элементы
  - У каждого из них свой тип

# Составные типы данных

- Включают в себя вложенные элементы
  - У каждого из них свой тип
- Списки, строки, коллекции, массивы…
  - Типы вложенных элементов одинаковы

# Составные типы данных

- Включают в себя вложенные элементы
  - У каждого из них свой тип
- Списки, строки, коллекции, массивы…
  - Типы вложенных элементов одинаковы
- Классы
  - Типы вложенных элементов произвольны

# Список

- Классическая структура данных в программировании:

# Список

- Классическая структура данных в программировании:
  - Произвольное количество вложенных **элементов** (от нуля до ...) – ограничено объёмом памяти

# Список

- Классическая структура данных в программировании:
  - Произвольное количество вложенных элементов (от нуля до ...) – ограничено объёмом памяти
  - Количество **элементов** = **размер** списка

# Список

- Классическая структура данных в программировании:
  - Произвольное количество вложенных элементов (от нуля до ...) – ограничено объёмом памяти
  - Количество **элементов** = **размер** списка
  - Тип всех элементов одинаковый

# Список

- Классическая структура данных в программировании:
  - Произвольное количество вложенных элементов (от нуля до ...) – ограничено объёмом памяти
  - Количество **элементов** = **размер** списка
  - Тип всех элементов одинаковый
  - У каждого элемента есть номер (индекс)

# Список

- Классическая структура данных в программировании:
  - Произвольное количество вложенных элементов (от нуля до ...) – ограничено объёмом памяти
  - Количество **элементов** = **размер** списка
  - Тип всех элементов одинаковый
  - У каждого элемента есть номер (индекс)
    - В Котлине – от 0 до размера списка – 1

# Список

- Классическая структура данных в программировании:
  - Произвольное количество вложенных элементов (от нуля до ...) – ограничено объёмом памяти
  - Количество **элементов** = **размер** списка
  - Тип всех элементов одинаковый
  - У каждого элемента есть номер (индекс)
    - В Котлине – от 0 до размера списка – 1
  - Элементы можно перебирать в порядке возрастания индексов

# Применение списков

- Функция может иметь **несколько** однотипных результатов

# Применение списков

- Функция может иметь **несколько** однотипных результатов
- Функция обрабатывает большое количество однотипных входных данных

# Применение списков

- Функция может иметь **несколько** однотипных результатов
- Функция обрабатывает большое количество однотипных входных данных
- И то, и другое вместе (операции над списками)

# Списки в Котлине

- Тип: List<T>, где вместо T подставляется тип элементов списка
  - `List<Int>, List<Double>, List<String>, List<List<Int>>, …`

# Списки в Котлине

- Тип: List<T>, где вместо T подставляется тип элементов списка
  - `List<Int>, List<Double>, List<String>, List<List<Int>>, …`
- Создание списка:
  - `listOf(elem1, elem2, …)`

# Пример: биквадратное уравнение

- $ax^4 + bx^2 + c = 0$

- Ранее (урок 2):
  искали один (наименьший) корень

- Функция может найти все
  (и вернуть их в виде списка)

# Алгоритм: биквадратное уравнение

- $ax^4 + bx^2 + c = 0$
- $x^2 = (-b \pm \text{sqrt}(D)) / 2a$

1. $a = 0 \rightarrow x = \pm \text{sqrt}(-c/b)$
2. $D = b^2 - 4ac$
3. $D < 0$?
4. $y_{1,2} = (-b \pm \text{sqrt}(D)) / 2a$
5. $x = \pm \text{sqrt}(y_1), \pm \text{sqrt}(y_2)$

# Решение: часть 1 (a = 0, d = 0)

```
fun biRoots(
    a: Double, b: Double, c: Double
): List<Double> {
    if (a == 0.0) {
        if (b == 0.0) return listOf()
        val bc = -c / b
        if (bc < 0.0) return listOf()
        val root = Math.sqrt(bc)
        return if (root == 0.0) listOf(root)
               else listOf(-root, root)
    }
    val d = discriminant(a, b, c)
    if (d < 0.0) return listOf()
    // ...
}
```

# Решение: часть 2 (a ≠ 0, d ≠ 0)

```
fun biRoots(
    a: Double, b: Double, c: Double
): List<Double> {
    // ...
    val y1 = (-b + Math.sqrt(d)) / (2 * a)
    val y2 = (-b - Math.sqrt(d)) / (2 * a)
    val part1 = if (y1 < 0) listOf()
                else if (y1 == 0.0) listOf(0.0)
                else {
                    val x1 = Math.sqrt(y1)
                    listOf(-x1, x1)
                }
    // ...
}
```

# Стоп!

- Уже несколько раз встречается следующая задача:
  - Решить уравнение $x^2 = y$

# Стоп!

- Уже несколько раз встречается следующая задача:
  - Решить уравнение $x^2 = y$

- $=>$
  Необходимо написать отдельную функцию

# Решение $x^2 = y$

```
fun sqRoots(y: Double) =
        if (y < 0) listOf()
        else if (y == 0.0) listOf(0.0)
        else {
            val root = Math.sqrt(y)
            // Результат!
            listOf(-root, root)
        }
```

# И возвращаемся к исходной задаче...

```
fun biRoots(
    a: Double, b: Double, c: Double
): List<Double> {
    if (a == 0.0) {
        if (b == 0.0) return listOf()
        else return sqRoots(-c / b)
    }
    val d = discriminant(a, b, c)
    if (d < 0.0) return listOf()
    if (d == 0.0) return sqRoots(-b / (2 * a))
    val y1 = (-b + Math.sqrt(d)) / (2 * a)
    val y2 = (-b - Math.sqrt(d)) / (2 * a)
    // Сложение списков
    return sqRoots(y1) + sqRoots(y2)
}
```

# Как проверить?

# Как проверить?

- Рассмотреть хотя бы один случай из каждой ветки

# Как проверить?

- Рассмотреть хотя бы один случай из каждой ветки
  1. $0x^4 + 0x2 + 1 = 0$ (корней нет)
  2. $0x^4 + 1x^2 + 2 = 0$ (корней нет)
  3. $0x^4 + 1x^2 - 4 = 0$ (корни –2, 2)
  4. $1x^4 - 2x^2 + 4 = 0$ (корней нет)
  5. $1x^4 - 2x^2 + 1 = 0$ (корни –1, 1)
  6. $1x^4 + 3x^2 + 2 = 0$ (корней нет)
  7. $1x^4 - 5x^2 + 4 = 0$ (корни –2, –1, 1, 2)

# Как проверить?

```
fun biRoots() {
    // Почему listOf<Double>?
    assertEquals(listOf<Double>(),  biRoots(0.0,  0.0,  1.0))
    assertEquals(listOf<Double>(),  biRoots(0.0,  1.0,  2.0))
    assertEquals(listOf(-2.0, 2.0), biRoots(0.0,  1.0, -4.0))
    assertEquals(listOf<Double>(),  biRoots(1.0, -2.0,  4.0))
    assertEquals(listOf(-1.0, 1.0), biRoots(1.0, -2.0,  1.0))
    assertEquals(listOf<Double>(),  biRoots(1.0,  3.0,  2.0))
    assertEquals(listOf(-2.0, -1.0, 1.0, 2.0),
                 biRoots(1.0, -5.0, 4.0))
}
```

# Последний случай (7)

```
org.opentest4j.AssertionFailedError:

    expected: <[-2.0, -1.0, 1.0, 2.0]>

    but was:  <[-2.0, 2.0, -1.0, 1.0]>
```

# Последний случай (7)

```
org.opentest4j.AssertionFailedError:

    expected: <[-2.0, -1.0, 1.0, 2.0]>

    but was:  <[-2.0, 2.0, -1.0, 1.0]>
```

Что произошло?

# Возможное исправление

```
fun biRoots() {
    // Почему listOf<Double>?
    assertEquals(listOf<Double>(),  biRoots(0.0,  0.0,  1.0))
    assertEquals(listOf<Double>(),  biRoots(0.0,  1.0,  2.0))
    assertEquals(listOf(-2.0, 2.0), biRoots(0.0,  1.0, -4.0))
    assertEquals(listOf<Double>(),  biRoots(1.0, -2.0,  4.0))
    assertEquals(listOf(-1.0, 1.0), biRoots(1.0, -2.0,  1.0))
    assertEquals(listOf<Double>(),  biRoots(1.0,  3.0,  2.0))
    assertEquals(listOf(-2.0, -1.0, 1.0, 2.0),
                 biRoots(1.0, -5.0, 4.0).sorted())
}
```

# Функция с получателем

- `list.sorted()`

# Функция с получателем

- `list.sorted()`
- Здесь list – *получатель* (receiver)

# Функция с получателем

- `list.sorted()`
- Здесь list – *получатель* (receiver)
- Получатель дополнительно подчёркивает, что данная операция **выполняется над ...** (в данном случае – над списком)

# Списки – распространённые операции

- `list1 + list2`
- `list  + newElement`

# Списки – распространённые операции

- `list1 + list2`
- `list  + newElement`
- `list.size`
- `list.isEmpty(), list.isNotEmpty()`

# Списки – распространённые операции

- `list1 + list2`
- `list  + newElement`
- `list.size`
- `list.isEmpty(), list.isNotEmpty()`
- `list[i]`
- `list.sublist(fromIndex, toIndex)`

# Списки – распространённые операции

- `list1 + list2`
- `list  + newElement`
- `list.size`
- `list.isEmpty(), list.isNotEmpty()`
- `list[i]`
- `list.sublist(fromIndex, toIndex)`
- `if (element in list)`
- `for (element in list)`
- `... (см. Chapter04)`

# Мутирующий список

- Так называемый **подтип** списка

# Мутирующий список

- Так называемый **подтип** списка
  - Может всё, что может список
  - + кое-что ещё

# Мутирующий список

- Так называемый **подтип** списка
  - Может всё, что может список
  - + кое-что ещё
    - `list[i] = element`

# Мутирующий список

- Так называемый **подтип** списка
  - Может всё, что может список
  - + кое-что ещё
    - `list[i] = element`
    - `list.add(element), list.remove(element)`

# Мутирующий список

- Так называемый **подтип** списка
  - Может всё, что может список
  - + кое-что ещё
    - `list[i] = element`
    - `list.add(element), list.remove(element)`
    - `list.add(index, element), list.removeAt(index)`

# Мутирующий список

- Так называемый **подтип** списка
  - Может всё, что может список
  - + кое-что ещё
    - `list[i] = element`
    - `list.add(element), list.remove(element)`
    - `list.add(index, element), list.removeAt(index)`
  - Может использоваться везде, где нужен список

# Мутирующий список

- Так называемый **подтип** списка
  - Может всё, что может список
  - + кое-что ещё
    - `list[i] = element`
    - `list.add(element), list.remove(element)`
    - `list.add(index, element), list.removeAt(index)`
  - Может использоваться везде, где нужен список
  - Тип: **MutableList<T>**

# Мутирующий список

- Так называемый **подтип** списка
  - Может всё, что может список
  - + кое-что ещё
    - `list[i] = element`
    - `list.add(element), list.remove(element)`
    - `list.add(index, element), list.removeAt(index)`
  - Может использоваться везде, где нужен список

  - Тип: **MutableList<T>**
  - Создание: mutableListOf(x, y, z)

# Пример: фильтрация

- Получение всех отрицательных чисел из исходного списка

```
fun negativeList(list: List<Int>): List<Int> {
    val result = mutableListOf<Int>()
    for (element in list) {
        if (element < 0) {
            result.add(element)
        }
    }
    return result
}
```

# Пример: обратить все положительные числа

```
fun invertPositives(list: MutableList<Int>) {
    for (i in 0..list.size - 1) {
        val element = list[i]
        if (element > 0) {
            list[i] = -element
        }
    }
}
```

# Пример: обратить все положительные числа

```
fun invertPositives(list: MutableList<Int>) {
    for (i in 0..list.size - 1) {
        val element = list[i]
        if (element > 0) {
            list[i] = -element
        }
    }
}

// Нет результата → Побочный эффект !
```

# Пример: обратить все положительные числа

```
fun invertPositives(list: MutableList<Int>) {
    for ((index, element) in list.withIndex()) {
        if (element > 0) {
            list[index] = -element
        }
    }
}
```

# Пример: обратить все положительные числа

```
fun invertPositives(list: MutableList<Int>) {
    for ((index, element) in list.withIndex()) {
        if (element > 0) {
            list[index] = -element
        }
    }
}

// for ((index, element) in ...) { ... }
// Перебор всех ПАР в списке (индекс, элемент)
```

# Строки

- «Почти что» списки
  - Произвольное количество *символов*

# Строки

- «Почти что» списки
  - Произвольное количество *символов*
  - Количество символов = **длина** строки

# Строки

- «Почти что» списки
  - Произвольное количество *символов*
  - Количество символов = **длина** строки
  - Тип всех элементов (символов) = Char

# Строки

- «Почти что» списки
  - Произвольное количество *символов*
  - Количество символов = **длина** строки
  - Тип всех элементов (символов) = Char
  - У каждого символа есть номер (индекс)

# Строки

- «Почти что» списки
  - Произвольное количество *символов*
  - Количество символов = **длина** строки
  - Тип всех элементов (символов) = Char
  - У каждого символа есть номер (индекс)
  - Символы можно перебирать в порядке возрастания индексов

# Строки

- «Почти что» списки
  - Произвольное количество *символов*
  - Количество символов = **длина** строки
  - Тип всех элементов (символов) = Char
  - У каждого символа есть номер (индекс)
  - Символы можно перебирать в порядке возрастания индексов

  - Строка НЕ подтип списка – хотя они очень похожи

# Строковые литералы (константы) в Котлине

```
"Some string\n\"with\" $something"
```

# Строковые литералы (константы) в Котлине

```
"Some string\n\"with\" $something"

// $something = строковый шаблон
// \n = новая строка
// \" = "
```

# Строковые литералы (константы) в Котлине

```
"Some string\n\"with\" $something"

""" Raw
    string
    literal
"""
```

# Строки – распространённые операции

- `str1 + str2`
- `str  + newChar`
- `str.length` *// вместо size у списка*
- `str.isEmpty(), str.isNotEmpty()`
- `str[i]`
- `str.substring(fromIndex, toIndex)`
- `if (char in str)`
- `for (char in str)`
- `...` (см. Chapter04)

# Пример: строка = палиндром?

- Палиндром: первый символ равен последнему, второй предпоследнему и т.п.

- «А роза упала на лапу Азора»

# Пример: строка = палиндром?

```
fun isPalindrome(str: String): Boolean {
    val lowerCase = str.toLowerCase().replace(" ", "")
    for (i in 0..lowerCase.length / 2) {
        if (lowerCase[i] !=
            lowerCase[lowerCase.length - i - 1]) {
            return false
        }
    }
    return true
}
```

# Список → Строка

```
// Библиотечная функция
fun <T> List<T>.joinToString(
        separator: String = ", ",
        prefix: String = "",
        postfix: String = "",
        limit: Int = -1,
        truncated: String = "..."
): String { ... }
```

# Список → Строка: пример

- Хотим из списка вида [3, 5, 6, 1, 4] сделать строку "3 + 5 + 6 + 1 + 4 = 19«

```
fun buildSumExample(list: List<Int>) =
    list.joinToString(
        separator = " + ",
        postfix   = " = ${list.sum()}"
    )
```

# Упражнения к лекции

- См. lesson4/task1 в обучающем проекте
- Решите хотя бы одно из заданий
- Протестируйте решение с помощью готовых тестов
- Добавьте ещё хотя бы один тестовый случай
- Добавьте коммит в свой репозиторий
- Создайте Pull Request и убедитесь в правильности решения