Generics

Мотивация

Давайте рассмотрим `ArrayIntList`, который писали в ондой из прошлых лекций. Оставим в нём только существенные методы, остальное по аналогии:

```
import java.util.*

class ArrayIntList {
    private var arr = IntArray(16)

    private var size: Int = 0
    fun getSize() : Int = size

    operator fun get(index: Int): Int {
        if (index >= size || index < 0) {
           throw ArrayIndexOutOfBoundsException("$index is out of bounds 0..<$size")
        }
        return arr[index]
    }

    fun add(element: Int) {
        if (size == arr.size) {
            arr = Arrays.copyOf(arr, arr.size + arr.size / 2 + 1)
        }
        arr[size] = element
        size++
    }

    fun clear() {
        size = 0
    }

    operator fun set(index: Int, element: Int) {
        if (index >= size || index < 0) {
           throw ArrayIndexOutOfBoundsException("$index is out of bounds 0..<$size")
        }
        arr[index] = element
    }

    override fun toString(): String = "[" + arr.joinToString() + "]"
}
```

... да и “значение по умолчанию” тоже уберём. В общем, получим практически такой `List<Int>`, как в стандартной библиотеке.

Но вот допустим, нам понадобилось то же самое, но для `Long`. Что теперь, копировать целиком весь код, заменяя в нём “`Int`” на “`Long`”? Выглядит как *не очень* идея.

В C++, например, есть template:

```
template<typename T>
class vector {
    int _size;
    int _capacity
    T* _arr;
    // ...
}
```

Фактически, при необходимости создаются копии этого кода, где `T` заменено на то, на что надо (`int`, `long` и т.д.).

И у нас есть механизм обобщения! Он называется generics. В объявлении класса пишем

```
class ArrayList<ElementType> {
```

И теперь внутри пользуемся `ElementType`, как будто это какой-то конкретный тип:

```
class ArrayList<ElementType> {
    private var arr : Array<Any?> = arrayOf()

    private var size: Int = 0
    fun getSize(): Int = size

    operator fun get(index: Int): ElementType {
        if (index >= size || index < 0) {
           throw ArrayIndexOutOfBoundsException("$index is out of bounds 0..<$size")
        }
        return arr[index] as ElementType
    }

    fun add(element: ElementType) {
        if (size == arr.size) {
            arr = Arrays.copyOf(arr, arr.size + arr.size / 2 + 1)
        }
        arr[size] = element
        size++
    }

    fun clear() {
        size = 0
    }

    operator fun set(index: Int, element: ElementType) {
        if (index >= size || index < 0) {
           throw ArrayIndexOutOfBoundsException("$index is out of bounds 0..<$size")
        }
        arr[index] = element
    }

    override fun toString(): String = "[" + arr.joinToString() + "]"
}
```

Есть один нюанс: создать массив элементов неизвестного заранее типа нам не позволят — это связано с некоторыми особенностями внутреннего устройства JVM. Поэтому создаём массив `Any?`, которые, как мы знаем, есть наиболее общий тип, и складываем всё туда. Ну и по схожим причинам начальный размер массива делаем `0`, а не `16`, как в случае с `Int`.

Из-за того, что массив теперь хранит `Any?`, нам приходится явно приводить его элементы (`as ElementType`), когда мы их возвращаем. Это не проблема, так как мы-то туда клали только `ElementType`, а значит, только они там и лежат!

Проверим, что работает:

```
  val intlist = ArrayList<Int>()
  intlist.add(1)
  intlist.add(2)
  intlist.add(3)
  intlist
```

```
  => : ArrayList<Int> = [1, 2, 3]
```

```
intlist[1]
```

```
  => : Int = 2
```

```
intlist[1] = 4
  intlist
```

```
  => : ArrayList<Int> = [1, 4, 3]
```

```
intlist.add("abc")
```

```
  Type mismatch. Required: Int. Found: String.
```

А вот добавить строку в лист чисел у нас не выйдет, что логично: функция требует на вход `ElementType`, который в данном случае `Int`.

И наоборот,

```
  val strlist = ArrayList<String>()
  strlist.add("abc")
  strlist
```

```
  => : ArrayList<String> = [abc]
```

```
  strlist[0] = 1
```

```
  Type mismatch. Required: String. Found: Int.
```

..., в лист строк положить число не выйдет.

Удобно? А как это работает?

А очень просто. Параметры типов (те самые, написанные в треугольных скобках) существуют только в момент компиляции. Компилятор знает, что в `ArrayList<String>` класть `Int` и наоборот запрещено. А вот в момент исполнения они все заменяются на `Any?`. То есть, у нас есть фактически одна реализация листа — `ArrayList<Any?>`, но за счёт подсказок компилятору мы избегаем лишних приведений типов и проверок.

Собственно, как вы могли догадаться, встроенные `List`, `MutableList` и `Array` используют ровно этот же механизм.

Pair

Мы умеем создавать структуры из нескольких элементов с помощью `data class`. Но это имеет смысл, если это не просто набор элементов, а набор, несущий совокупный смысл. Например, это не просто тройка чисел — это вектор. Или это не просто пара чисел — это границы промежутка. А что если мы просто хотим временно похранить пару чего-нибудь? Создавать под каждую отдельный класс? Нет, ведь мы уже умеем сделать вот так:

```
data class Pair<F, S>(val first: F, val second: S) {
      override fun toString() = "($first, $second)"
  }
```

... собственно, на этом содержательная часть пары заканчивается. Единственное, что появилось нового — здесь два параметра типов. Ну и обычно их называют, всё же, одной заглавной буквой — чтобы легко в коде отличать от содержательных типов.

Сортировка

Теперь допустим, мы хотим написать функцию, которая сортирует элементы изменяемого листа. Теперь нам нужно параметризовать функцию... и это тоже можно делать!

```
  fun <T> sort(list: MutableList<T>) {
      // Code here
  }
```

Но с элементами неизвестного типа мы мало что можем сделать. На самом деле, только то же, что с элементами `Any`? — сравнивать на равенство, преобразовывать в строку и считать хэшкод. Сравнивать их мы не умеем... Но, если помните, однажды упоминалось, что у `Int` и `String` есть общий тип — `Comparable`, что означает, что их можно сравнивать. Давайте ограничим все возможные `T` сравнимыми:

```
  fun <T : Comparable<T>> sort(list: MutableList<T>) {
      // Code here
  }
```

Тип Comparable здесь тоже параметризованный: его параметром является то, *с чем* можно сравнивать. Например, `Int` является `Comparable<Int>`, `String` является `Comparable<String>`.

> Вообще говоря, не обязательно реализовывать интерфейс `Comparable`, чтобы иметь возможность сравниваться — для этого достаточно `operator fun compareTo`. `Comparable<T>` как раз один этот метод и содержит: поэтому хорошим тоном является реализовать этот интерфейс, если возможно сравнение элементов типа.

Давайте допишем сортировку и проверим.

```
  fun <T : Comparable<T>> isSorted(list: MutableList<T>): Boolean {
      for (i in 1 until list.size) if (list[i - 1] > list[i]) return false
      return true
  }

  fun <T : Comparable<T>> sort(list: MutableList<T>) {
      while (!isSorted(list)) {
          list.shuffle()
      }
  }
```

```
val list = mutableListOf(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
sort(list)
list
```

```
=> : MutableList<Int> = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Отличнейшим образом работает.

Either

Ещё одно распространённое применение генериков. Допустим, в какой-то переменной мы хотим хранить *либо* `Int`, *либо* `String`, и ничего больше. Мы можем назначить ей тип `Comparable<*>`, конечно, но это не предохранит нас от того, чтобы туда положили `Double`, например. Так давайте напишем специальный тип, который традиционно называется `Either` (в Kotlin stdlib его нет, но его довольно часто реализуют в своих проектах разработчики. В C++ есть такой тип, это `std::variant`).

```
class Either<L, R> private constructor(
    val isLeft: Boolean, private val value: Any?
) {
    fun asLeft(): L = if (isLeft) value as L else throw IllegalStateException()
    fun asRight(): R = if (!isLeft) value as R else throw IllegalStateException()
}
```

Осталось решить с конструированием. Оставить публичным конструктор мы не можем — мало ли что туда положат. Попробуем объявить два конструктора, от L и от R

```
class Either<L, R> private constructor(
    val isLeft: Boolean, private val value: Any?
) {
    constructor(l: L) : this(true, l)
    constructor(r: R) : this(false, r)

    fun asLeft(): L = if (isLeft) value as L else throw IllegalStateException()
    fun asRight(): R = if (!isLeft) value as R else throw IllegalStateException()
}
```

```
Conflicting overloads:
public constructor Either<L, R>(l: L) defined in generics.Either,
public constructor Either<L, R>(r: R) defined in generics.Either
```

Почему конфликт? Ведь наборы аргументов разные? Во-первых, это оправдано логически: а что, если вы захотите создать `Either<Int, Int>`? Тогда они совпадут. Во-вторых, это оправдано со стороны JVM — ведь для неё, как мы помним, нет параметров типов — происходит так называемое `type erasure`. С точки зрения JVM есть два конструктора с единственным параметром типа `Any?` (на самом деле для JVM наиболее общий тип — `Object`, но не суть).

Так что нам понадобятся функции с разными именами.

```
class Either<L, R> private constructor(
    val isLeft: Boolean, private val value: Any?
) {
    constructor(l: L) : this(true, l)
    constructor(r: R) : this(false, r)

    fun asLeft(): L = if (isLeft) value as L else throw IllegalStateException()
    fun asRight(): R = if (!isLeft) value as R else throw IllegalStateException()

    companion object {
        fun <L, R> fromLeft(value: L): Either<L, R> = Either(true, value)
        fun <L, R> fromRight(value: R): Either<L, R> = Either(false, value)
    }
}
```

На самом деле, можно использовать генерики для довольно умных вещей. Например, мы хотим получить содержимое `Either`, не обращая внимания на то, “левое” оно или “правое”. Если мы просто возьмём `value`, то получим тип `Any`?... а мы хотели умное. Давайте введём ещё один параметр типа:

```
fun <C, L : C, R : C> Either<L, R>.content(): C = if(isLeft) asLeft() else asRight()
```

Что только что произошло? Мы сказали, что мы возвращаем не что-нибудь, а общий тип `L` и `R`. Обычно типы `L` и `R` компилятору известны — они следуют из того, на чём вызывается метод. А дальше он просто выводит `C` как наиболее узкий подходящий тип.

... Мы ещё вернёмся к `Either`, когда побольше узнаем про лямбды...