Контракты и наследование

Интерфейсы

Так, допустим, у нас есть некая реализация динамического массива целых чисел. Мы можем безболезненно вставлять элемент по любому неотрицательному индексу и массив автоматически расширится.

```
import java.util.*

class ArrayIntList {
    private var arr = IntArray(16)
    private var capacity = 16
    private var size: Int = 0

    fun getSize() : Int = size

    operator fun contains(element: Int): Boolean {
        for (i in 0 until size) if (arr[i] == element) return true
        return false
    }

    fun containsAll(elements: Collection<Int>): Boolean {
        for (el in elements) {
            if (!contains(el)) return false
        }
        return true
    }

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

    fun isEmpty(): Boolean = size == 0

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

    fun addAll(elements: Collection<Int>) {
        if (capacity < size + elements.size) {
               arr = Arrays.copyOf(arr, maxOf(capacity + capacity / 2 + 1, size +
elements.size))
            capacity = arr.size
        }
        for (el in elements) set(size++, el)
    }

    fun clear() {
        size = 0
```

```
    }

    operator fun set(index: Int, element: Int) {
        if (capacity <= index) {
            arr = Arrays.copyOf(arr, maxOf(capacity + capacity / 2 + 1, index + 1))
            capacity = arr.size
        }
        arr[index] = element
        if (index >= size) size = index + 1
    }

    override fun toString(): String {
        val sb = StringBuilder("[")
        var empties = 0
        for (i in 0 until getSize()) {
            if (this[i] == 0) empties++ else {
                if(i != 0) sb.append(", ")
                if(empties != 0) sb.append("<$empties empty slots>, ")
                empties = 0
                sb.append(this[i])
            }
        }
        if(empties != 0) sb.append("<$empties empty slots>")
        sb.append("]")
        return sb.toString()
    }
}
```

Что у нас есть:

- getSize — возвращает текущий размер массива
- contains — проверяет, содержится ли такой элемент
- containsAll — проверяет, содержатся ли все элементы из коллекции
- get — берёт по индексу, по умолчанию возвращает ноль
- isEmpty — проверяет, не пуст ли массив
- add — добавляет элемент в конец
- addAll — добавляет все элементы в конец
- clear — очищает массив (в данном случае чистить `arr` необходимости нет, всё лежащее за `size` теперь будет игнорироваться)
- set — кладёт по индексу
- toString — щадаёт строковое представление. Это на лекции не обсуждалось, я дописал для наглядности. Устроен несложно, можете разобраться.

Ничего интеллектуального. Что здесь изменилось с прошлой лекции? Теперь у нас у объектов класса есть состояние, которое хранится в `var`'ах, вместо просто постоянных данных, которые хранились в `val`'ах

Давайте напишем функцию, заполняющую заданный отрезок массива заданным числом.

```
fun fill(list: ArrayIntList, fromIndex: Int, toIndex: Int, value: Int) {
    for (i in fromIndex until toIndex) {
        list[i] = value
    }
}
```

Проверим, что всё означенное дело работает.

```
val arr = ArrayIntList()
arr.addAll(listOf(1, 2, 3, 4, 5))
arr[9] = 566
arr
```

```
=> : ArrayIntList = [1, 2, 3, 4, 5, <4 empty slots>, 566]
```

Поведение ожидаемое.

Теперь `fill`:

```
fill(arr, 20, 30, -1)
arr
```

```
=> : ArrayIntList = [1, 2, 3, 4, 5, <4 empty slots>, 566, <10 empty slots>, -1, -1,
-1, -1, -1, -1, -1, -1, -1, -1]
```

Теперь такой момент. Это хорошо, когда массив небольшого размера. А если захочется сделать так?

```
fill(arr, 5, 30, 0)
arr[1000000] = 1
arr
```

```
=> : ArrayIntList = [1, 2, 3, 4, 5, <999995 empty slots>, 1]
```

Окей, работает. Попробуем

```
fill(arr, 5, 30, 0)
arr[2147000000 /*Don't remember*/] = 1
arr
```

```
=> : ArrayIntList = [1, 2, 3, 4, 5, <999995 empty slots>, 1]
```

Хотелось бы увидеть здесь ответ

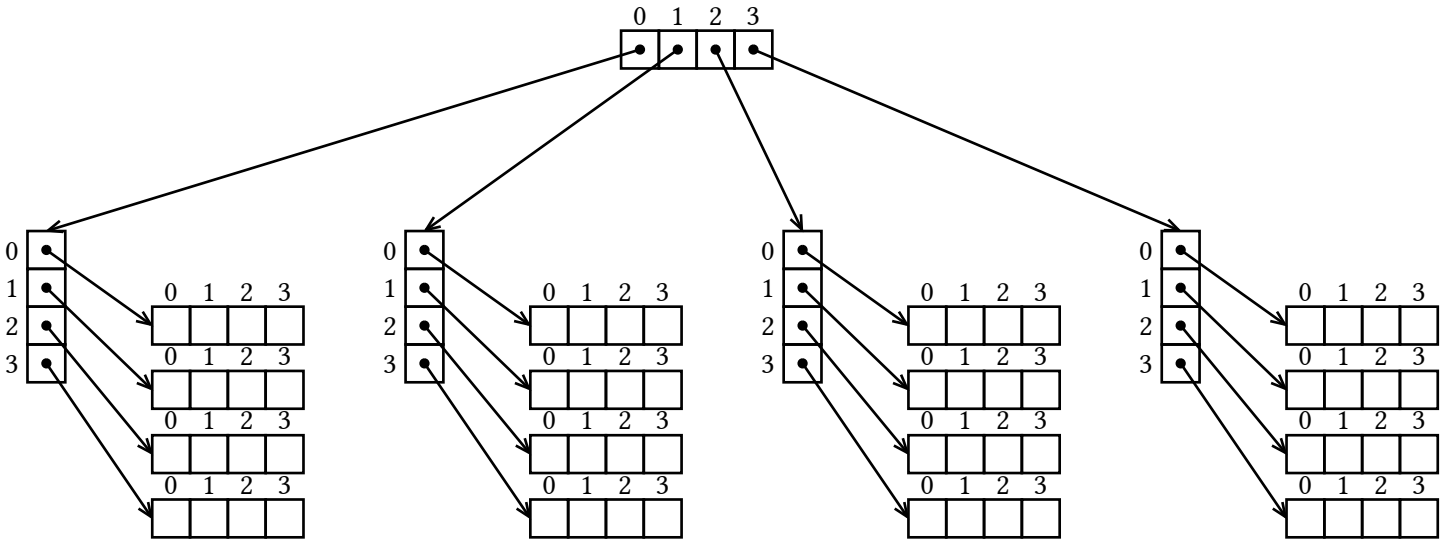
```
=> : ArrayIntList = [1, 2, 3, 4, 5, <999995 empty slots>, 1, <2145999999 empty
slots>, 1]
```

Но на деле мы получим

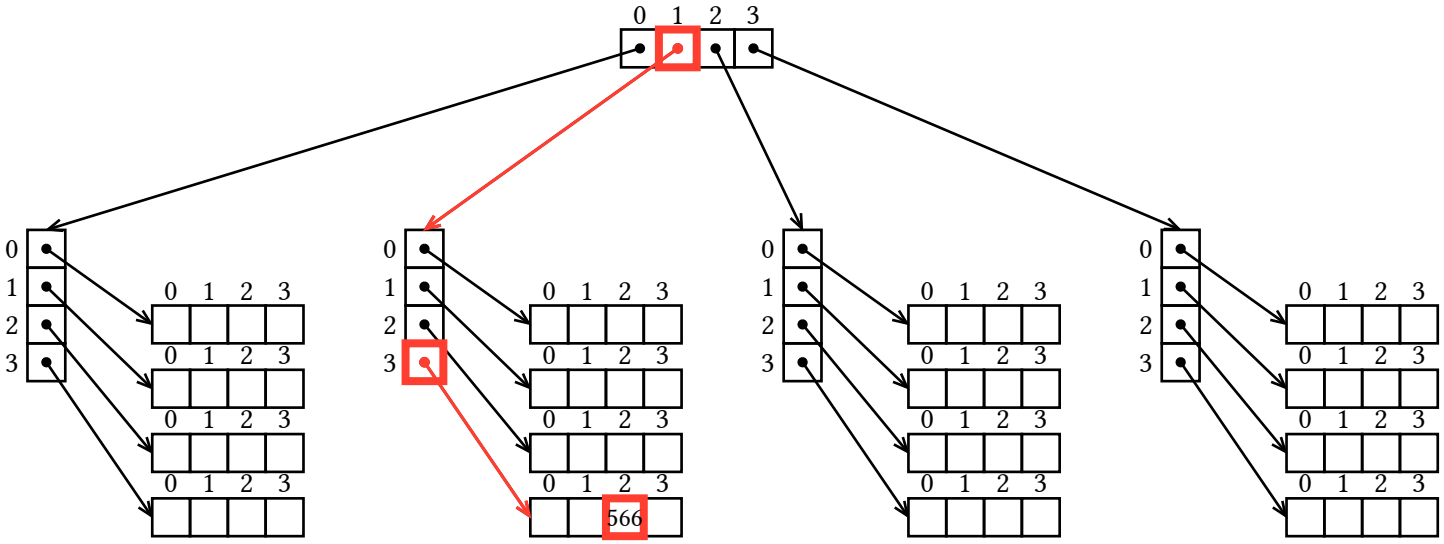
```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at java.util.Arrays.copyOf(Arrays.java:3284)
    at pack.ArrayIntList.set(Test.kts:53)
    at pack.TestKts.main(Test.kts:91)
    at pack.TestKts.main(Test.kts)
```

Нам намекают, что мы хотим слишком много памяти. $2^{31} \cdot 4$ байт, то есть 8Гб, если быть точным.

Давайте вдохновимся JS и напишем другой вариант этого списка.

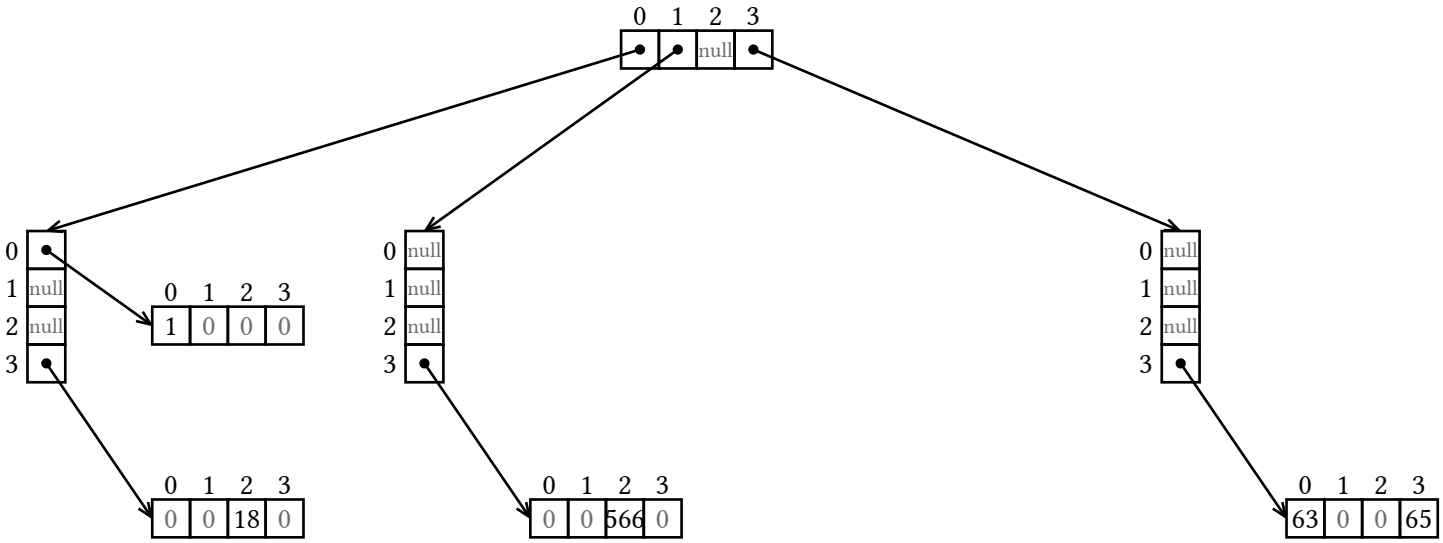


Будем хранить в памяти вот такую иерархическую структуру. Допустим, мы хотим положить элемент 566 по индексу 30. Для этого переводим 30 в четверичную систему: $30_{10} = 132_4$. В верхнем массиве выберем элемент по индексу 1, там лежит ссылка на блок. Там выберем элемент по индексу 3, получим ссылку на массив. По индексу 2 положим туда 566.



Так, а чем это лучше? Мы же совершаем кучу лишних индирекций по оперативной памяти?

Хотя бы тем, что мы можем не хранить лишние элементы структуры, заменив их на `null`.



Преимущества на лицо! Существенно меньше памяти, если данные сильно распределены. Можно, конечно, было бы воспользоваться какой-нибудь хэш-таблицей, но это ещё сложнее.

Давайте по-быстрому это реализуем. Только вместо массивов длины 3 будем использовать длиной 256, а всю структуру сделаем глубиной 4. $256^4 = 2^{32}$, тогда как массивы позволяют индексы до $2^{31} - 1$. Ну, останется место при желании занумеровать элементы отрицательными индексами.

```
class ChunkedIntList {
    private var arr: Array<Array<Array<IntArray?>?>?>? = null
    private var size: Int = 0

    fun getSize(): Int = size

    operator fun contains(element: Int): Boolean {
        if (arr == null) return false
        for (i in 0 until 256) {
            if (arr!![i] == null) continue
            for (j in 0 until 256) {
                if (arr!![i]!![j] == null) continue
                for (k in 0 until 256) {
                    if (arr!![i]!![j]!![k] == null) continue
                    for (l in 0 until 256) {
                        if (arr!![i]!![j]!![k]!![l] == element) return true
                    }
                }
            }
        }
        return false
    }

    fun containsAll(elements: Collection<Int>): Boolean {
        for (el in elements) {
            if (!contains(el)) return false
        }
        return true
    }
```

```
    operator fun get(index: Int): Int {
        if (index < 0) throw ArrayIndexOutOfBoundsException("$index")
        if (index >= size) return 0
       return arr?.get(index shr 24)?.get(index shr 16 and 255)?.get(index shr 8 and
255)?.get(index and 255) ?: 0
    }

    fun isEmpty(): Boolean = size == 0

    fun add(element: Int) {
        val i = size shr 24
        val j = size shr 16 and 255
        val k = size shr 8 and 255
        val l = size and 255
        if (arr == null) arr = Array(256) { null }
        if (arr!![i] == null) arr!![i] = Array(256) { null }
        if (arr!![i]!![j] == null) arr!![i]!![j] = Array(256) { null }
        if (arr!![i]!![j]!![k] == null) arr!![i]!![j]!![k] = IntArray(256)
        arr!![i]!![j]!![k]!![l] = element
        size++
    }

    fun addAll(elements: Collection<Int>) {
        for (el in elements) add(el)
    }

    fun clear() {
        size = 0
        arr = null
    }

    operator fun set(index: Int, element: Int) {
        val i = index shr 24
        val j = index shr 16 and 255
        val k = index shr 8 and 255
        val l = index and 255
        if (arr == null) arr = Array(256) { null }
        if (arr!![i] == null) arr!![i] = Array(256) { null }
        if (arr!![i]!![j] == null) arr!![i]!![j] = Array(256) { null }
        if (arr!![i]!![j]!![k] == null) arr!![i]!![j]!![k] = IntArray(256)
        arr!![i]!![j]!![k]!![l] = element
        if (index >= size) size = index + 1
    }
}
```

`toString` будет выглядеть так же, не будем занимать лишнее место в конспекте. Что здесь есть нового? `IntArray` — это почти `Array<Int>`, только эффективнее (это связано с внутренним устройством JVM). `shr`, `and` — битовые операции над числами (shift right — битовый сдвиг вправо, and — битовое “и”). При очищении обнуляем `size` и удаляем ссылку на `arr`. Остальное за нас сборщик мусора подберёт.

Давайте проверять!

```
val arr2 = ChunkedIntList()
arr2.addAll(listOf(1, 2, 3, 4, 5))

arr2[1000000] = 1
arr2[2147000000] = 1
arr2
```

```
=> : ChunkedIntList = [1, 2, 3, 4, 5, <999995 empty slots>, 1, <2145999999 empty slots>, 1]
```

Отличнейшим образом работает.

```
fill(arr2, 10, 20, -1)
```

```
Kotlin: Type mismatch: inferred type is ChunkedIntList but ArrayIntList was expected
```

Э-э-э… А? Ну, впрочем, ничего удивительного. Мы ведь объявляли функцию как работающую с `ArrayIntList`, откуда бы компилятору знать, что `ChunkedIntList` умеет всё то же самое и не хуже? Надо ему рассказать. Для этого существуют интерфейсы.

```
interface IntList {
      fun getSize(): Int
      operator fun contains(element: Int): Boolean
      fun containsAll(elements: Collection<Int>): Boolean
      operator fun get(index: Int): Int
      fun isEmpty(): Boolean
      fun add(element: Int)
      fun addAll(elements: Collection<Int>)
      fun clear()
      operator fun set(index: Int, element: Int)
      override fun toString(): String
  }
```

Здесь мы описываем сигнатуры, модификаторы и возвращаемые типы. Всё. Теперь у нас есть отдельный тип под названием `IntList`, с которым можно делать всё перечисленное. Но реализации методов у нас нигде не прописаны. Как нигде и не указано, что `ArrayIntList` и `ChunkedIntList` реализуют их не просто так, а чтобы подходить под требование `IntList`. Для этого надо написать это в заголовке класса, а ко всем соответствующим методам дописать модификатор `override`.

```
class ArrayIntList : IntList {
      private var arr = IntArray(16)
      private var capacity = 16
      private var size: Int = 0

      override fun getSize(): Int = size

      override operator fun contains(element: Int): Boolean {
          for (i in 0 until size) if (arr[i] == element) return true
          return false
      }

      // et cetera
  }
```

```
class ChunkedIntList : IntList {
      private var arr: Array<Array<Array<IntArray?>?>?>? = null
      private var size: Int = 0

      override fun getSize(): Int = size

      override operator fun contains(element: Int): Boolean {
          if (arr == null) return false
          for (i in 0 until 256) {
              if (arr!![i] == null) continue
              for (j in 0 until 256) {
                  if (arr!![i]!![j] == null) continue
                  for (k in 0 until 256) {
                      if (arr!![i]!![j]!![k] == null) continue
                      for (l in 0 until 256) {
                          if (arr!![i]!![j]!![k]!![l] == element) return true
                      }
                  }
              }
          }
          return false
      }

      // et cetera
  }
```

Теперь вполне можем написать

```
  private fun fill(list: IntList, fromIndex: Int, toIndex: Int, value: Int) {
    for (i in fromIndex until toIndex) {
        list[i] = value
    }
  }

  val arr = ChunkedIntList()
  fill(arr, 5, 10, -1)
  arr[2147000000] = -2
  arr
```

```
  => : ChunkedIntList =  [, <5 empty slots>, -1, -1, -1, -1, -1, <2146999990 empty
  slots>, -2]
```

It's alive!

Обратим внимание теперь, что у нас очень много похожего кода. `addAll`, `containsAll`, `toString`, `isEmpty` — вообще одинаково, в `get` отличается одна строчка. Учитывая, что `set` автоматически создаст необходимое место, оба `add` можно было бы заменить на вызов `set`. Давайте что-нибудь с этим сделаем, а то ведь если мы захотим новую какую-нибудь реализацию (хоть бы и на хэш-таблице), что, снова всё перепечатывать?

Справедливости ради, можно было бы сделать следующим образом: убрать их из `IntList` и объявить как внешние (external):

```
fun IntList.containsAll(elements: Collection<Int>): Boolean {
  for (el in elements) {
      if (!contains(el)) return false
  }
  return true
}
```

...как мы могли сделать и с `fill`, в общем-то... Но вот, например, с `toString` такой фокус не пройдёт — почему, поймёте чуть позже. Если мы объявим её как внешнюю, мы продолжим получать абракадабру вроде `ChunkedIntList@610455d6`. Поэтому делать будем по-другому.

Абстрактные классы

Это сущность, очень похожая на интерфейс, но с двумя существенными отличиями — она может описывать не только сигнатуру методов, но и их реализацию; а также может иметь собственное состояние, хранимое в переменных. Чаще всего они используются для того как раз, чтобы задать “частичную” реализацию интерфейса.

```
import java.util.*

interface IntList {
    fun getSize(): Int
    operator fun contains(element: Int): Boolean
    fun containsAll(elements: Collection<Int>): Boolean
    operator fun get(index: Int): Int
    fun isEmpty(): Boolean
    fun add(element: Int)
    fun addAll(elements: Collection<Int>)
    fun clear()
    operator fun set(index: Int, element: Int)
}
abstract class AbstractIntList : IntList {
    override fun containsAll(elements: Collection<Int>): Boolean {
        for (el in elements) {
            if (!contains(el)) return false
        }
        return true
    }

    override fun isEmpty(): Boolean = getSize() == 0

    override fun addAll(elements: Collection<Int>) {
        for (el in elements) add(el)
    }

    override fun add(element: Int) {
        set(getSize(), element)
    }
```

```
    override fun toString(): String {
        val sb = StringBuilder("[")
        var empties = 0
        for (i in 0 until getSize()) {
            if (this[i] == 0) empties++ else {
                if (i != 0) sb.append(", ")
                if (empties != 0) sb.append("<$empties empty slots>, ")
                empties = 0
                sb.append(this[i])
            }
        }
        if (empties != 0) sb.append("<$empties empty slots>")
        sb.append("]")
        return sb.toString()
    }
}
class ArrayIntList : AbstractIntList() {
    private var arr = IntArray(16)
    private var capacity = 16
    private var size: Int = 0

    override fun getSize(): Int = size

    override operator fun contains(element: Int): Boolean {
        for (i in 0 until size) if (arr[i] == element) return true
        return false
    }

    override operator fun get(index: Int): Int {
        if (index < 0) throw ArrayIndexOutOfBoundsException("$index")
        if (index >= size) return 0
        return arr[index]
    }

    override fun clear() {
        size = 0
    }

    override operator fun set(index: Int, element: Int) {
        if (capacity <= index) {
            arr = Arrays.copyOf(arr, maxOf(capacity + capacity / 2 + 1, index + 1))
            capacity = arr.size
        }
        arr[index] = element
        if (index >= size) size = index + 1
    }
}
```

```
class ChunkedIntList : AbstractIntList() {
      private var arr: Array<Array<Array<IntArray?>?>?>? = null
      private var size: Int = 0

      override fun getSize(): Int = size

      override operator fun contains(element: Int): Boolean {
          if (arr == null) return false
          for (i in 0 until 256) {
              if (arr!![i] == null) continue
              for (j in 0 until 256) {
                  if (arr!![i]!![j] == null) continue
                  for (k in 0 until 256) {
                      if (arr!![i]!![j]!![k] == null) continue
                      for (l in 0 until 256) {
                          if (arr!![i]!![j]!![k]!![l] == element) return true
                      }
                  }
              }
          }
          return false
      }

      override operator fun get(index: Int): Int {
          if (index < 0) throw ArrayIndexOutOfBoundsException("$index")
          if (index >= size) return 0
         return arr?.get(index shr 24)?.get(index shr 16 and 255)?.get(index shr 8 and
  255)?.get(index and 255) ?: 0
      }

      override fun clear() {
          size = 0
          arr = null
      }

      override operator fun set(index: Int, element: Int) {
          val i = index shr 24
          val j = index shr 16 and 255
          val k = index shr 8 and 255
          val l = index and 255
          if (arr == null) arr = Array(256) { null }
          if (arr!![i] == null) arr!![i] = Array(256) { null }
          if (arr!![i]!![j] == null) arr!![i]!![j] = Array(256) { null }
          if (arr!![i]!![j]!![k] == null) arr!![i]!![j]!![k] = IntArray(256)
          arr!![i]!![j]!![k]!![l] = element
          if (index >= size) size = index + 1
      }
  }

  fun fill(list: IntList, fromIndex: Int, toIndex: Int, value: Int) {
      for (i in fromIndex until toIndex) {
          list[i] = value
      }
  }
```

Теперь мы пишем, что классы `ArrayIntList` и `ChunkedIntList` *наследуют* `AbstractIntList`. То есть, все реализации методов, описанные в `AbstractIntList`, отработают ровно так же и на

наследниках. Впрочем, класс-наследник их может переопределить (`override`), то есть написать свою реализацию, которая имеет приоритет над “родительской” реализацией.

Обратите внимание, что класс-предок ничего не знает о своих наследниках: он, в некотором смысле, описывает “общий случай”. Поэтому в нём обращение к приватной переменной `size` заменено на `getSize` — откуда ему знать, что, по факту, все его наследники хранят `size` в просто переменной? Точно так же класс не должен знать о своих “братьях” — далеко не всегда можно гарантировать, что ты знаешь обо всех наследниках твоего предка.

Также здесь внезапно появился вызов родительского *конструктора* — круглые скобки после `: AbstractIntList`. Мы не объявили там никакого конструктора, поэтому по умолчанию создался пустой, без аргументов, просто создающий объект без всяких внешних данных.

Можно реализовывать *сколько угодно* интерфейсов, но наследовать *не более чем у одного* класса. Это связано с тем, что если вы наследуете несколько реализаций методов с одинаковой сигнатурой, не вполне понятно, в каком случае какую из них вызывать (Языки с множественным наследованием существуют, но там бывают довольно… запутанные правила).

Также в абстрактных классах можно требовать от наследников реализации определённых методов, написав их сигнатуру с модификатором `abstract`.

Например, можно сделать так:

```
abstract class AbstractIntList : IntList {
    // blah-blah

    override operator fun get(index: Int): Int {
        if (index < 0) throw ArrayIndexOutOfBoundsException("$index")
        if (index >= size) return 0
        return take(index)
    }

    protected abstract fun take(index: Int) : Int
}

class ArrayIntList : AbstractIntList() {
    // blah-blah

    override take(index: Int) : Int = arr[index]
}

class ChunkedIntList : AbstractIntList() {
    // blah-blah

    override fun take(index: Int) = arr
            ?.get(index shr 24)
            ?.get(index shr 16 and 255)
            ?.get(index shr 8 and 255)
            ?.get(index and 255) ?: 0
}
```

В целом, абстрактный класс может не оставлять нереализованных методов, но тогда не очень понятно, зачем ему вообще быть абстрактным. Единственное его отличие в таком случае — нельзя создать объект этого типа (это логично — абстрактность позволяет иметь нереализованные методы, а что мы будем делать, если попытаемся его вызвать…). В таком случае можно объявить класс `open`. *Открытые* классы можно наследовать и при этом

можно создавать объекты (инстанцировать — от английского instance). При этом вы можете разрешить переопределять методы `open` класса, объявив их также `open`

Хороший пример такого класса — `Any`. Это открытый класс, который содержит три открытых метода — `equals`, `hashCode`, `toString`. Первый отвечает за сравнение на равенство (==), второй — за вычисление хэш-кода, третий — за строковое представление. Как раз их мы переопределяем, если хотим, чтобы == и вывод на объектах этих классов работал адекватно. Переопределённые функции автоматически отмечаются `open`

Виртуальное наследование

Допустим, у нас есть следующая схема.

```
abstract class Parent {
    abstract fun foo()
}

class ChildA : Parent() {
    override fun foo() {
        println("ChildA: foo")
    }
}

open class ChildB : Parent() {
    override fun foo() {
        println("ChildB: foo")
    }
}

class Grandchild : ChildB() {
    override fun foo() {
        println("Grandchild: foo")
    }
}

val obj : Parent = Grandchild()
obj.foo()
```

```
Grandchild: foo
```

Почему так?

Вернее, хорошо, что оно так работает, но как? Откуда на самом деле компилятор знает, какой вариант функции вызвать, если тип переменной — `Parent`? А он и не знает. Уже во время исполнения происходит переход по ссылке, лежащей в переменной — туда, где лежит сам объект, читается его заголовок (где, в частности, прописан его настоящий класс), после чего, в зависимости от результата, переход в нужную версию функции.

Такой вариант наследования называется виртуальным — таблица, где выполняется поиск нужной реализации, соответственно, таблицей виртуальных функций. В JVM всё наследование виртуальное.

Собственно, это отвечает на вопрос, почему определённая в качестве external функция `toString` никоим образом не помогает. Объявленные внутри класса методы транслируются в… методы java-классов, а внешние функции — в функции, принимающие `receiver type` первым аргументом. Поэтому когда `println` вызовет `toString` на вашем объекте, будет вызван собственный метод, а не внешняя функция.

Бонус 1: Интерфейсы-маркеры

Никто, в общем-то, не запрещает делать интерфейс, который не содержит методов вовсе. Они служат двум целям: либо объединить в себе несколько других (интерфейс наследует другой интерфейс — означает, что он требует реализовать то, что описано в нём самом, и то, что описано в его предках), либо просто пометить класс как принадлежащий какой-то группе. Как раз маркером является, например, интерфейс Serializable — его требования не формализуемы в виде просто набора методов, но если вы помечаете класс как Serializable — это означает, что в нём есть всё необходимое, чтобы объект этого класса можно было записать в файл и восстановить после прочтения.

Бонус 2: Абстрактные постоянные

На самом деле, когда вы объявляете в классе переменную или постоянную, вы делаете две вещи: собственно заводите переменную или постоянную, и заводите метод `getИмя` и `setИмя` (set — понятное дело, только для переменной). И всегда, когда вы обращаетесь по синтаксису через точку — вы на самом деле вызываете соответствующий метод (так это сделано в Котлине, в большинстве языков по-другому). В обычном случае они ничего не делают умного — просто записывают или возвращают значение. Но можно также их изменить. Для этого есть ключевые слова `get` и `set`.

```
var variable : Int = 0
    get() {
        return field
    }
    set(value) {
        field = value
    }
```

Собственно, здесь описано в явном виде то, что они делают по умолчанию. Но можно добавить туда, например, вывод сообщения при каждом обращении, или какие-то проверки, и так далее.

А ещё можно модифицировать область видимости этих методов отдельно:

```
var variable : Int = 0
    private set
```

Это, например, переменная, которую можно читать из любого места (по умолчанию модификатор `public`), а записывать только изнутри класса (`private`). Ещё может быть `protected` — доступно только наследникам, и `internal` — внутри одного пакета.

Всё это даёт нам возможность провернуть любопытный фокус: можно в интерфейсе объявить постоянную (то есть как бы потребовать реализовать только геттер), а в реализации сделать её переменной с приватным сеттером:

```
interface IntList {
    val size : Int
    // et cetera
}

class ArrayIntList : IntList {
    override var size : Int
        private set
    // et cetera
}
```