List<E> и ArrayList<E>

Интерфейс: `java.util.List`

Класс: `java.util.ArrayList`

Описание

Интерфейсный тип `List<E>` описывает абстракцию "конечный упорядоченный список объектов типа `E`". Здесь `E` может быть любым объектным типом, в том числе и списочным. Например:

- `List<String>` — это список строк;
- `List<LocalDate>` — список дат;
- `List<List<String>>` — список списков строк;
- `List<Object>` — список, который может хранить объекты любого типа.

Класс `ArrayList<E>` является конкретной реализацией интерфейса `List<E>`. Он реализует список с *константным временем доступа по индексу* (то есть время на извлечение элемента по индексу не зависит от длины списка и положения в нём элемента), доступный для записи и динамически расширяющийся при добавлении в него новых элементов.

Интерфейс и реализация

Тип `List` — это абстрактный тип. Сам по себе он не реализует алгоритм хранения списка и операции с ним. Нельзя просто так взять и создать экземпляр типа `List` *(Boromir.jpg)*. Если точнее, то `List` — это *интерфейс*. Можно создать только экземпляр конкретного класса, реализующего интерфейс `List`; одним из таких классов, и наиболее часто используемым, является `ArrayList`. Например, мы могли бы создать список строк вот так:

```
List<String> strings = new ArrayList<>();
```

Пустые угловые скобки — это просто синтаксический сахар, означающий "использовать справа тот тип, который ожидается слева". Можно, но не нужно, записать ту же инструкцию более длинным способом:

```
List<String> strings = new ArrayList<String>();
```

Как видно, переменная интерфейсного типа может ссылаться на объект любого класса, реализующего этот интерфейс. На практике принято работать именно с абстрактным типом `List`, а конкретный тип считать деталью реализации, "забыв" о нём сразу после создания. Например, почти все классы

стандартной библиотеки принимают и возвращают именно тип `List`, а не `ArrayList` или другой конкретный тип. Это позволяет в случае необходимости легко сменить реализацию списка, не трогая интерфейс, например:

```
List<String> strings = new LinkedList<>();
```

Не все реализации интерфейса `List` поддерживают все методы, объявленные в нём. Например, если вы реализуете список фиксированного размера, то для него будет бессмысленна операция `add`, добавляющая элемент в список. Библиотечные списки фиксированного размера при попытке вызова `add` бросают `UnsupportedOperationException`. Точно так же в списках, доступных только для чтения, то же исключение бросает операция `set`, заменяющая элемент списка. А вот `ArrayList` поддерживает все доступные операции.

Создание

Как мы знаем, конструктор `ArrayList` без параметров создаёт пустой список:

`ArrayList<E>()`

К сожалению, списки могут хранить только объекты; нельзя создать список, хранящий числовые типы или тип `boolean`. Поэтому вместо примитивных типов нужно использовать классы-обёртки:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(5);
```

Если `T1` и `T2` — разные типы, то компилятор не даст присвоить переменной типа `List<T1>` выражение типа `List<T2>`, даже если переменной типа `T1` можно присвоить выражение типа `T2`. В самом деле, если бы это было возможно, мы могли бы испортить список, добавив в него объекты не того типа:

```
String str = "Hello";
Object obj = str;                           // можно

List<String> strList = new ArrayList<>();
List<Object> objList = strList;             // нельзя, но давайте представим...
objList.add(LocalDate.of(2016, 2, 24));     // упс, положили не строку
System.out.println(strList.get(0).length()); // ой-ой-ой
```

Это означает что нельзя присвоить переменной типа `List<List<E>>` выражение типа `ArrayList<ArrayList<E>>` (подумайте, почему). Зато можно создать копию списка, используя *конструктор копии*, который принимает любую коллекцию (в том числе и список) любого типа, который можно положить в наш список:

`ArrayList<E>(Collection<любой_подтип_E> source)`

Например:

```
List<String> strList = new ArrayList<>();
strList.add("Hello");
List<String> listCopy = new ArrayList<>(strList);

List<Object> objList = new ArrayList<>(strList);  // вот так уже можно, это копия
objList.add(LocalDate.of(2016, 2, 24));
```

Длина и доступ по индексу

Любой список поддерживает две операции: получение текущего размера списка, то есть числа элементов в нём, и получение элемента списка по индексу:

`int size()`

`E get(int index)`

Например:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(5);                      // int -> Integer
numbers.add(100);                    // int -> Integer
System.out.println(numbers.size());  // 2
System.out.println(numbers.get(0));  // 5

int secondElement = numbers.get(1);  // Integer -> int
System.out.println(secondElement);   // 100
```

Аналогично строкам и массивам, возможные индексы начинаются с 0 и заканчиваются `size() - 1`.

Как и для строк, существует сокращённая форма проверки на пустой список:

`boolean isEmpty()`

Она аналогична `size() == 0`, но для некоторых списков (например, `LinkedList`) выполняется быстрее.

Важно! Поскольку вместо примитивных типов мы вынуждены хранить в списке объекты-обёртки, нужно помнить, что выражения типа `Integer` (а также `Long`, `Double` и т.д.) могут иметь значение `null`, тогда как примитивные типы `int`, `long`, `double` и т.д. не могут хранить `null`. Таким образом, нужна осторожность при работе со списками, допускающими хранение значений `null` (а `ArrayList` как раз из их числа). Например, следующий код не сработает:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(null);
int i = numbers.get(0);  // бросает NullPointerException
```

Если вы принимаете списки из внешнего кода, будьте готовы к тому, что они могут содержать `null`.

Итерация (обход списка)

В принципе методов `size` и `get` достаточно, чтобы написать код, обходящий все элементы списка от начала к концу:

```
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```

Но так делать обычно не стоит. Не все реализации `List` поддерживают константное время доступа по индексу, как это делает `ArrayList`. Например, для связанного списка `LinkedList` операция `get` занимает время, пропорциональное индексу элемента (то есть выполняется тем дольше, чем дальше элемент от начала). Говорят, что эта операция имеет *линейную сложность* по времени. Если мы применим

к `LinkedList` наивный цикл обхода вроде показанного выше, то время выполнения этого цикла будет пропорционально уже квадрату размера списка *(квадратичная сложность)*.

И это ужасно. *(LexLuthor.jpg)*

К счастью, каждый список умеет эффективно обходить сам себя. Операция обхода списка *(итерация)* является настолько базовой, что для неё даже есть поддержка на уровне языка Java. Вот так мы можем эффективно обойти любой список — хоть `ArrayList`, хоть `LinkedList`, хоть любой другой — за линейное время:

```
for (E e : list) {
    // код, использующий e
}
```

Такой цикл называется циклом *for-each*. Вот пример его применения:

```
List<String> strings = new ArrayList<>();
strings.add("Hello");

for (String str : strings) {
    System.out.println(str);
}
```

Для совсем дотошных скажу, что такая запись является синтаксическим сахаром, а под капотом происходит вот какой ужас:

```
for (Iterator<String> it = strings.iterator(); it.hasNext(); ) {
    String str = it.next();
    System.out.println(str);
}
```

Но до итераторов мы ещё доберёмся. Они понадобятся нам, если мы захотим написать свой класс, поддерживающий итерацию циклом for-each.

Обычный же цикл for с переменной-счётчиком может понадобиться, если внутри цикла нужен доступ не только к элементам списка, но и к их индексам. Только нужно обходить таким образом только списки, поддерживающие константное время доступа по индексу. Например, `ArrayList`.

Печать списка

Быстро распечатать весь список (например, в отладочных целях) можно, получив его строковое представление с помощью метода `toString`. Он возвращает строку со всеми элементами списка через запятую и в квадратных скобках.

```
List<Integer> numbers = new ArrayList<>();
numbers.add(5);
numbers.add(100);

String str = numbers.toString();  // "[5, 100]"
System.out.println(str);
System.out.println(numbers);      // Так тоже можно: println сам вызовет toString
```

Напечатать каждый элемент на отдельной строке можно циклом, а можно и с помощью метода `forEach`, с которым мы познакомимся позже (как и со странным синтаксисом, использованным здесь).

```
numbers.forEach(System.out::println);
```

Результат:

```
5
100
```

Линейный и двоичный поиск

Метод `indexOf` реализует линейный поиск элемента в списке (по `equals`) от начала к концу, аналогично строкам:

```
int indexOf(E element)
```

Он возвращает индекс первого вхождения элемента или -1, если элемент не найден.

Аналогично работает метод `lastIndexOf`, который ищет не с начала, а с конца списка, возвращая индекс последнего вхождения (или -1).

```
int lastIndexOf(E element)
```

Если же список отсортирован, можно применить более быстрый алгоритм двоичного поиска с помощью статического метода `Collections.binarySearch`.

```
static int binarySearch(List<сравнимый_тип T> list, T key)
```

```
static int binarySearch(List<T> list, T key, Comparator<любой_супертип_T> comparator)
```

Здесь "сравнимый тип" означает тип, реализующий *естественный порядок* с помощью интерфейса `Comparable`; кроме того, можно явно передать компаратор, по которому будет осуществляться сравнение элементов. Подробнее о компараторах написано в разделе "Сортировка".

У этого метода возвращаемое значение более хитрое, чем у семейства `indexOf`. Он тоже возвращает неотрицательное значение, если элемент найден, и отрицательное, если элемент не найден, но это отрицательное значение не всегда равно -1. Если точнее, то оно равно `-insertionPoint - 1`, где `insertionPoint` — это тот индекс, при вставке по которому искомого элемента с помощью метода `add(beforeIndex, element)` список останется отсортированным.

Изменение списка

Теперь мы рассмотрим операции, которые не читают из списка, а пишут в него. Их поддерживают не все списки. К счастью, `ArrayList` реализует все эти операции.

Метод `add` добавляет элемент в конец списка или, если первым параметром указан индекс, *перед* этим индексом:

```
boolean add(E element)
```

```
void add(int beforeIndex, E element)
```

Например, `list.add(0, element)` добавляет элемент в начало списка. При добавлении элемента не в конец списка все следующие за ним элементы сдвигаются вправо.

Кстати, что ещё за `boolean` такой? Дело в том, что любой список является коллекцией, но не любая коллекция является списком. Метод `add` с одним параметром возвращает `true`, если элемент действительно был добавлен. А некоторые коллекции (они называются *множествами*) не хранят более одного одинакового элемента, и для них `add` возвращает `false`, если такой элемент во множестве уже есть. Но списки всегда хранят ровно столько элементов, сколько в них было добавлено (даже если они

повторяются), и ровно в том же порядке, поэтому для них `add` всегда возвращает `true`.

Добавлять элементы по одному не всегда удобно (и не всегда эффективно). Метод `addAll` позволяет скопировать в список содержимое целой коллекции (в том числе и другого списка) — если, конечно, она совместима по типу.

`boolean addAll(Collection<любой_подтип_E> source)`

`boolean addAll(int beforeIndex, Collection<любой_подтип_E> source)`

Кстати, конструктор копии для класса `ArrayList` аналогичен вызову конструктора без параметров с последующим вызовом `addAll`. Например, такой код

```
List<String> strList = new ArrayList<>();
List<Object> objList = new ArrayList<>(strList);
```

делает то же, что и

```
List<String> strList = new ArrayList<>();
List<Object> objList = new ArrayList<>();
objList.addAll(strList);
```

Парным к методу `add` является `remove`. Одна его форма удаляет элемент по индексу, а вторая удаляет *первый найденный* элемент, равный переданному:

`boolean remove(int index)`

`boolean remove(Object o)`

Если элемент удаляется не из конца списка, то следующие за ним элементы сдвигаются влево.

Важно! Сравнение элементов осуществляется по `equals`, а не по `==`. Таким образом, приведённый ниже код удалит из списка строк единственный его элемент, несмотря на то, что элемент списка и параметр метода `remove` являются разными объектами:

```
List<String> strList = new ArrayList<>();
strList.add("Hello");
strList.remove(new String("Hello"));
```

Важно! Будьте осторожны с методом `remove` в списках типа `List<Integer>`. Если вы передадите `int` вместо `Integer` и наоборот, может вызваться не та версия метода `remove`, которая нужна:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(5);
numbers.remove(5);    // remove(int) по индексу; IndexOutOfBoundsException

Integer obj = 5;
numbers.remove(obj);  // remove(Object) по значению; OK
```

Аналогично `addAll` есть метод `removeAll`, удаляющий все элементы переданной коллекции, найденные в списке:

`boolean removeAll(Collection<любой_тип> collection)`

и ещё есть метод `retailAll`, который, наоборот, оставляет в списке только те элементы, которые есть в переданной коллекции (по сути, реализуя операцию пересечения):

`boolean retainAll(Collection<любой_тип> collection)`

Методы `remove`, `removeAll` и `retainAll` возвращают `true`, если список был изменён, то есть какие-то элементы действительно были найдены и удалены.

Метод `clear` очищает весь список, делая его пустым:

`void clear()`

И, наконец, метод `set` не изменяет длину списка, а заменяет элемент по указанному индексу другим, возвращая тот элемент, который находился по этому индексу до этого:

`E set(int index, E element)`

Например:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(5);
numbers.add(100);

System.out.println(numbers);             // [5, 100]
System.out.println(numbers.set(0, 42));  // 5
System.out.println(numbers);             // [42, 100]
```

Как пример, с помощью двоичного поиска и методов `add` и `remove` с индексом можно реализовать поверх `ArrayList` отсортированное множество. (Но не делайте так! Библиотечный класс `TreeSet` намного эффективнее.)

```
public class HorribleSortedSet<E extends Comparable<? super E>>
        extends AbstractSet<E> {
    private final List<E> list = new ArrayList<>();

    @Override
    public Iterator<E> iterator() {
        return list.iterator();
    }

    @Override
    public boolean contains(E element) {
        return Collections.binarySearch(list, element) >= 0;
    }

    @Override
    public boolean add(E element) {
        int index = Collections.binarySearch(list, element);

        if (index >= 0) {
            return false;  // Элемент уже присутствует
        } else {
            list.add(-index - 1, element);
            return true;
        }
```

```
    }

    @Override
    public boolean remove(E element) {
        int index = Collections.binarySearch(list, element);

        if (index >= 0) {
            list.remove(index);
            return true;
        } else {
            return false;  // Элемент не найден
        }
    }
}
```

Важно! Разные реализации списков оптимизированы для разных целей. Например, `ArrayList` оптимизирован для добавления элементов в конец списка и для доступа по индексу, но вставка и удаление не из конца списка имеют линейную сложность, потому что приводят к сдвигу элементов массива, который `ArrayList` использует внутри. А `LinkedList` оптимизирован для вставки и удаления в любое место списка, но потребляет больше памяти; кроме того, для доступа к элементу по индексу N ему нужно сначала пройти N - 1 предыдущих элементов, поэтому операции с индексами имеют для него линейную сложность.

В типичных приложениях, как правило, используется всё же `ArrayList`, потому что приложения, как правило, чаще читают из произвольных мест списков, чем изменяют их.

Списки и массивы

К сожалению, во многих старых API в Java, созданных до появления удобной абстракции списков, используются массивы там, где уместнее был бы список. Поэтому иногда приходится преобразовывать списки в массивы и наоборот.

Копию списка в виде массива можно получить с помощью метода `toArray`:

`E[] toArray(E[] array)`

По историческим причинам у этого метода довольно странный синтаксис. Он принимает массив и заполняет его начало, если длины переданного массива хватает, чтобы вместить копию всего списка, а в противном случае создаёт новый массив того же типа и нужной длины.

```
List<Integer> numbers = new ArrayList<>();
numbers.add(5);
numbers.add(100);

Integer[] array1 = new Integer[1];             // слишком мало
Integer[] result1 = numbers.toArray(array1);
System.out.println(array1 == result1);         // false
System.out.println(Arrays.toString(array1));   // [0]
System.out.println(Arrays.toString(result1));  // [5, 100]

Integer[] array2 = new Integer[4];             // больше, чем нужно
Integer[] result2 = numbers.toArray(array2);
System.out.println(array2 == result2);         // true
```

```
System.out.println(Arrays.toString(array2));   // [5, 100, 0, 0]
```

Рекомендуемой идиомой является передача в метод `toArray` нового массива с размером, равным размеру списка:

```
numbers.toArray(new Integer[numbers.size()])
```

Важно! В примере используется массив объектов-обёрток `Integer[]`, а не чисел `int[]`. К сожалению, преобразовать список `List<Integer>` в массив `int[]` не так просто. Можно написать такой метод самостоятельно или использовать метод `Ints.toArray` из библиотеки Google Guava.

Важно! Для печати элементов массива нужно использовать `Arrays.toString`. Вызов же метода `toString` у самого массива вернёт малоинтересную тарабарщину:

```
System.out.println(array2);  // [C@279f2327
```

Обратную же операцию — преобразование массива в список — можно осуществить статическим методом `Arrays.asList`.

```
List<T> asList(T... array)
```

Этот метод возвращает не `ArrayList`, а представление массива в виде списка фиксированной длины. У такого списка нельзя вызывать никакие методы, изменяющие его длину (то есть `add`, `addAll`, `remove`, `removeAll`, `retainAll` и `clear`). Но можно вызывать методы `get` и `set`, причём изменения в массиве отразятся на списке и наоборот:

```
String[] stringArray = { "First", "Second" };
List<String> stringList = Arrays.asList(stringArray);

stringArray[0] = "Первый";
System.out.println(stringList);                  // [Первый, Second]
stringList.set(0, "Второй");
System.out.println(Arrays.toString(stringArray)); // [Первый, Второй]
```

Если же нужен именно `ArrayList` (например, для добавления и удаления элементов), всегда можно воспользоваться конструктором копии, но полученный список будет только копией массива, не связанной с ним.

```
List<String> stringArrayList = new ArrayList<>(stringList);
stringArrayList.set(0, "FIRST");
System.out.println(stringArrayList);             // [FIRST, Второй]
System.out.println(Arrays.toString(stringArray)); // [Первый, Второй]
```

Сортировка

Для сортировки используется метод `sort`. Конечно, он работает, только если список доступен для записи (метод `set`).

```
void sort(Comparator<любой_супертип_E> comparator)
```

Для работы алгоритма сортировки нужен *компаратор* — нечто, что умеет сравнивать и упорядочивать два объекта по отношению "больше-меньше". Где взять компаратор? Для некоторых классов уже реализован *естественный порядок*, совпадающий с их интуитивным упорядочением (для чисел — по возрастанию, для строк — по алфавиту, для дат — из прошлого в будущее). Оказывается, что классы `Integer`, `String` и большинство классов даты и времени (например, `LocalDate`) реализуют интерфейс `Comparable`, и у них есть метод `compareTo`. Списки элементов такого типа можно передать другой форме метода сортировки, который является статическим методом класса `Collections`, либо с `null` вместо компаратора:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(100);
numbers.add(5);
numbres.add(42);
Collections.sort(numbers); // 5, 42, 100; стиль Java 1.2
numbers.sort(null);        // то же самое; стиль Java 8
```

А если нам нужен порядок сортировки, отличный от естественного, или же тип элемента не реализует `Comparable`? Тогда нам нужно передать методу `sort` свой компаратор. В стандартной библиотеке есть несколько уже имеющихся компараторов. Например, в классе `Collections` есть метод, возвращающий компаратор, сортирующий в обратном порядке:

```
static <T> Comparator<T> reverseOrder()
```

С его помощью мы можем отсортировать наш список `numbers` по убыванию:

```
numbers.sort(Collections.reverseOrder()); // 100, 42, 5
```

Также в качестве компаратора можно передать ссылку на любой метод, принимающий два параметра нужного нам типа и возвращающий int (при этом в качестве первого параметра может выступать неявный параметр `this`). Иными словами, такой метод может иметь одну из двух форм:

```
int E.имяМетода(E element2)
```

```
static int ЛюбойКласс.имяМетода(E element1, E element2)
```

Соглашения о возвращаемом значении такие же, как для `compareTo`:

- < 0, если первый элемент меньше второго
- 0, если два элемента равны
- > 0, если первый элемент больше второго

Например, такую форму имеет метод `String.compareToIgnoreCase`. Поэтому, если мы хотим отсортировать список имён файлов в папке, то, скорее всего, пользователь захочет видеть их упорядоченными без учёта регистра. Это можно сделать двумя способами:

```
String[] array = { "FILE.txt", "user.bin", "abc.txt" };
List<String> fileNames = Arrays.asList(array);

fileNames.sort(String.CASE_INSENSITIVE_ORDER);  // Стиль Java 1.2
fileNames.sort(String::compareToIgnoreCase);    // Стиль Java 8
// Оба метода дадут один и тот же результат:
// abc.txt, FILE.txt, user.bin
```

Ещё в качестве компаратора можно передать *лямбда-выражение* или экземпляр любого класса, реализующего интерфейс `Comparator`, но этого мы пока касаться не будем.