# Iterable<T>, Iterator<T>



Интерфейс: java.lang.Iterable

Интерфейс: java.util.Iterator

## Описание

Класс, реализующий интерфейс `Iterable<T>`, представляет абстракцию "последовательность элементов". Такие классы можно обходить циклом for-each.

Все коллекции (`Collection`), в том числе множества (`Set`) и списки (`List`), реализуют интерфейс `Iterable`. Именно поэтому для них работает цикл for-each.

```
StringSequence sequence = <создание объекта>;

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

## Реализация интерфейса Iterable

Чтобы реализовать интерфейс `Iterable<T>` для конкретного типа `T`, нужно объявить с помощью ключевого слова `implements`, что наш класс реализует интерфейс, и объявить в нём один метод, создающий новый объект — итератор:

`public Iterator<T> iterator()`

Например:

```
public class StringSequence implements Iterable<String> {
    @Override
    public Iterator<String> iterator() {
        // реализация
    }
}
```

Где взять итератор? Если наш класс оборачивает другой класс, реализующий интерфейс `Iterable`, то итератор можно позаимствовать у него:

```
public class StringSequence implements Iterable<String> {
    private final List<String> list;

    public StringSequence(Collection<String> data) {
        list = new ArrayList<>(data);
    }

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

Поскольку `ArrayList` реализует `Iterable` и содержит в себе точно те элементы, по которым мы хотим пройтись, его итератор подойдёт для нашей реализации (с небольшой оговоркой, на которой мы остановимся позже).

Но что, если мы реализуем структуру данных с нуля, а не как обёртку над чужим классом?

Тогда нам нужно самим создавать итератор. Тип `Iterator` — это опять-таки интерфейс, который должен реализовывать уже другой класс.

## Реализация интерфейса Iterator

Итератор — это вспомогательный объект, предназначенный для обхода последовательности. Он реализует абстракцию "указатель на текущий элемент" и умеет продвигаться по последовательности вперёд (но не назад).

Сразу после создания итератор указывает на начало последовательности. Любой итератор поддерживает две операции:

`boolean hasNext()`
: Проверяет, не достигли ли мы конца последовательности (остались ли ещё элементы для чтения).

`T next()`
: Возвращает следующий элемент последовательности и продвигается на один элемент вперёд. Если мы пытаемся выйти за границы последовательности, выбрасывает `NoSuchElementException`.

Таким образом, цикл for-each

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

эквивалентен такой записи (и именно так реализуется под капотом):

```
for (Iterator<String> iter = sequence.iterator(); iter.hasNext(); ) {
    String str = iter.next();

    System.out.println(str);
}
```

Зная это, мы можем написать свой итератор. Поскольку `Iterator<T>` — это тоже интерфейс, мы должны объявить его в разделе `implements` класса-итератора.

Итератору, как правило, нужен доступ к деталям реализации последовательности, которую он обходит. Обычно итератор реализуется как *внутренний класс*, но это необязательно. Можно реализовать итератор и как видимый в пределах пакета (package-private) обычный класс, передав ему детали реализации в конструкторе.

Шаш класс `StringSequence` использует для реализации `ArrayList`. Напишем свой собственный итератор для `ArrayList`, не пользуясь стандартным.

```
class StringSequenceIterator implements Iterator<String> {
    private final List<String> list;  // список, который обходим
    private int index;                // индекс текущего элемента

    StringSequenceIterator(List<String> list) {
        this.list = list;
        index = 0;  // встаём в начало
    }

    @Override
    public boolean hasNext() {
        return index < list.size();
    }

    @Override
    public String next() {
        if (!hasNext()) {  // условие конца последовательности
            throw new NoSuchElementException();
        }

        String result = list.get(index);
        index++;
        return result;
        // Три строки выше можно записать одной:
        // return list.get(index++);
    }
}
```

И тогда в самом классе `StringSequence` мы будем создавать этот итератор так:

```
public class StringSequence implements Iterable<String> {
    private final List<String> list;

    public StringSequence(Collection<String> data) {
        list = new ArrayList<>(data);
    }

    @Override
    public Iterator<String> iterator() {
        return new StringSequenceIterator(list);
    }
}
```

## Удаление элементов

Класс-итератор может реализовать также необязательную операцию `remove`, которая удаляет *последний возвращённый методом `next` элемент*. Кроме того, операцию `remove` можно

вызвать только один раз между вызовами `next`; чтобы удалить элемент, следующий за только что удалённым, нужно сначала вызвать `next`, потом снова `remove`.

Метод `remove` не нужен для работы цикла for-each. Более того, он недоступен в цикле for-each, и использовать его можно только при явной работе с объектом-итератором.

Реализация `remove` по умолчанию просто выбрасывает `UnsupportedOperationException`.

**`boolean remove()`**

Удаляет последний элемент, возвращённый методом `next`. Выбрасывает `IllegalStateException`, если мы находимся в начале последовательности (то есть `next` ни разу не вызывался), либо если метод `remove` уже был вызван с момента последнего вызова `next`.

Как правило, итератор, реализующий `remove`, хранит дополнительный флаг, показывающий, вызывался ли уже метод `remove` с момента последнего вызова `next`.

Вот пример реализации итератора с поддержкой `remove`:

```java
class StringSequenceIterator implements Iterator<String> {
    private final List<String> list;  // список, который обходим
    private int index;                // индекс текущего элемента
    private boolean removeCalled;

    StringSequenceIterator(List<String> list) {
        this.list = list;
        index = 0;  // встаём в начало
        removeCalled = false;
    }

    @Override
    public boolean hasNext() {
        return index < list.size();
    }

    @Override
    public String next() {
        if (!hasNext()) {  // условие конца последовательности
            throw new NoSuchElementException();
        }

        removeCalled = false;  // сбрасываем флаг при вызове next
        return list.get(index++);
    }

    @Override
    public void remove() {
        if (removeCalled) {
            throw new IllegalStateException("remove already called");
        }

        if (index == 0) {
            throw new IllegalStateException("next never called");
        }

        // Поскольку мы ранее увеличили index в next, тот элемент,
        // который последним вернул next, находится по индексу
        // index - 1
        index--;             // сдвигаемся влево
        list.remove(index);  // удаляем последний возвращённый
        // теперь index указывает на элемент, следующий за удалённым
        // устанавливаем флаг, чтобы исключить двойной вызов
        removeCalled = true;
    }
}
```

**Важно!** Поскольку итераторы изменяемых последовательностей (например, `ArrayList`) допускают удаление элементов, неосторожное переиспользование итераторов из деталей реализации может привести к нарушению инкапсуляции. Например, наш изначальный класс `StringSequence`, не использующий свой итератор, по задумке является неизменяемым, но возврат итератора `ArrayList` позволяет удалять элементы списка, тем самым изменяя его в обход инкапсуляции:

```
StringSequence sequence = <...>;
Iterator<String> iter = sequence.iterator();
iter.next();
iter.remove();  // удалили первый элемент!
```

Лучше всего не использовать изменяемые структуры данных внутри неизменяемых, или хотя бы свести к минимуму их использование. Этой ошибки можно было бы избежать, если бы мы сделали наш внутренний список неизменяемым, обернув `ArrayList` в вызов статического метода `Collections.unmodifiableList`:

```
public StringSequence(Collection<String> data) {
    list = Collections.unmodifiableList(new ArrayList<>(data));
}
```

В библиотеке Google Guava есть класс `ImmutableList`, реализующий ту же идиому более эффективно. Он является неизменяемой реализацией списка, не оборачивающей никакой другой список, и просто копирует переданные в него данные:

```
list = ImmutableList.copyOf(data);
```

**Важно!** Во время использования итератора — будь то в цикле for-each или напрямую — нельзя изменять объект `Iterable` никакими другими способами, кроме как через сам итератор. В противном случае возникает *неопределённое поведение* (undefined behavior), то есть с последовательностью может случиться всё, что угодно, и она может необратимо испортиться. Классы, поддерживающие итерацию, не обязаны никак обрабатывать такой конфликт, хотя стандартные библиотечные коллекции (в том числе `ArrayList` и `LinkedList`) отслеживают и обрабатывают такую ситуацию:

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

for (String str: list) {
    list.add("World");  // бросает ConcurrentModificationException
}
```