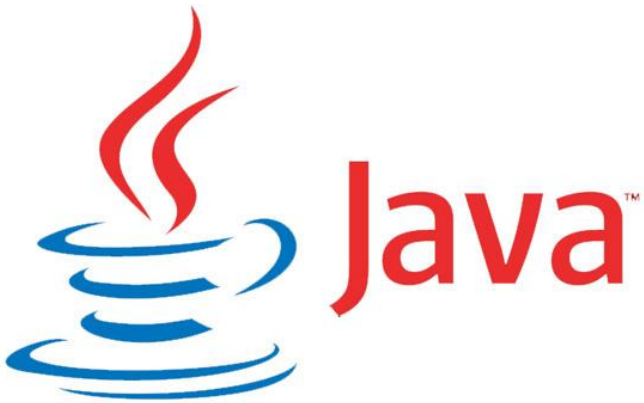
Тема 11.
Коллекции (часть 1)



Понятие коллекции

Коллекция - это объект, способный хранить группу одинаковых элементов. Она содержит большое количество готовых методов для работы с однородными данными.

Для чего нам нужны коллекции если у нас уже есть массивы, которые могут хранить другие объекты? Все дело в простоте и удобстве использования. Мы просто берем и используем готовые решения.

В чем еще преимущество использования коллекций:

- удобство тестирования кода;
- структуры данных на любой вкус и потребность;
- повторное использование кода.

Понятие коллекции

Классы коллекций располагаются в пакете **java.util**, поэтому перед применением коллекций следует подключить данный пакет.

Хотя в Java существует множество коллекций, но все они образуют стройную и логичную систему. В основе всех коллекций лежит применение того или иного интерфейса, который определяет базовый функционал.

Понятие коллекции

Основу библиотеки составляют открытые интерфейсы, которые **можно** использовать для создания собственных коллекций, либо же пользоваться уже существующими коллекциями.

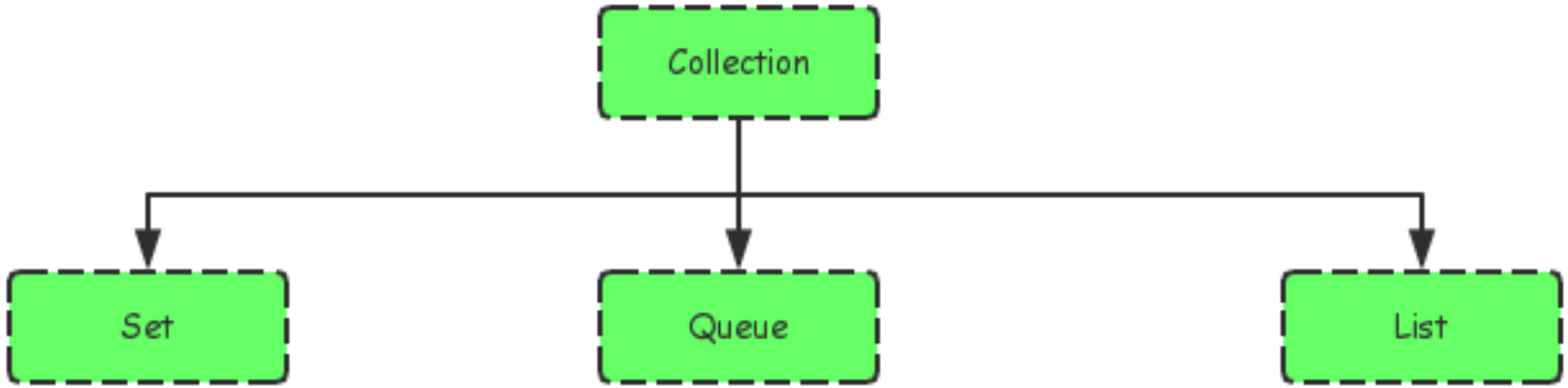
Collection – хранит набор объектов в виде к которому мы уже привыкли изучая массивы: есть объект он помещается в ячейку.

Map – хранит данные в виде пары “ключ-значение”.

В обоих случаях возможны базовые манипуляции: удаление, вставка, поиск и т.д.

Collection

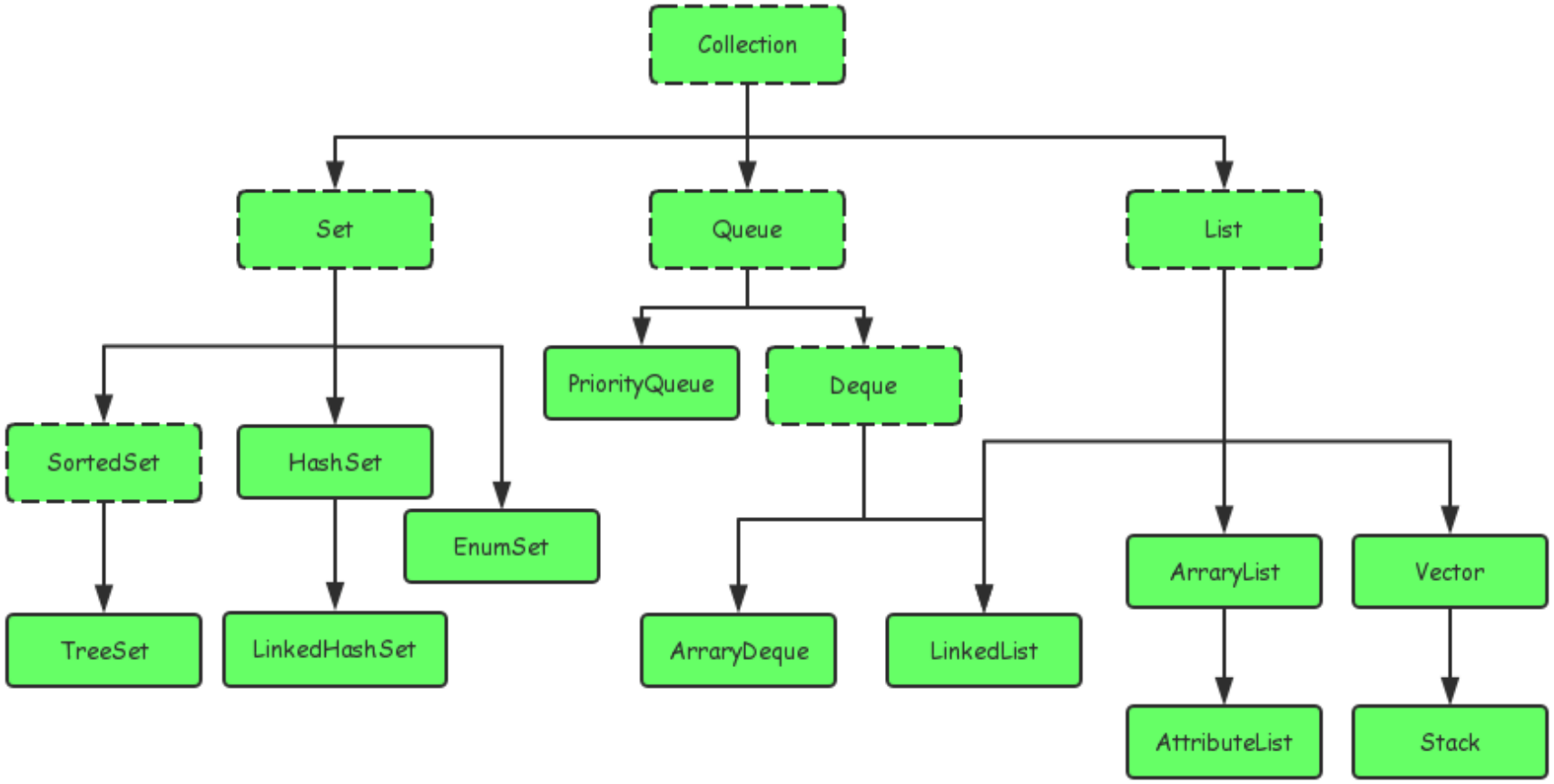
Интерфейс **Collection** наследуется другими интерфейсами, которые в свою очередь имплементируются классами, в которых реализована та или иная структура данных.



- **Set** используется для хранения множеств **уникальных объектов**
- **List** представляет функциональность **простых списков**
- **Queue** представляет функционал для структур данных в виде **очереди**

Collection

Далее, от этих трех основных интерфейсов порождается ряд классов, каждый со своими особенностями.



Collection

Интерфейс Collection является базовым для всех коллекций, определяя основной функционал:

- **boolean add (E item)**: добавляет в коллекцию объект item. При удачном добавлении возвращает true, при неудачном - false
- **boolean addAll (Collection<? extends E> col)**: добавляет в коллекцию все элементы из коллекции col. При удачном добавлении возвращает true, при неудачном - false
- **void clear ()**: удаляет все элементы из коллекции
- **boolean contains (Object item)**: возвращает true, если объект item содержится в коллекции, иначе возвращает false
- **boolean isEmpty ()**: возвращает true, если коллекция пуста, иначе возвращает false

Collection

- **Iterator<E> iterator ():** возвращает объект Iterator для обхода элементов коллекции
- **boolean remove (Object item):** возвращает true, если объект item удачно удален из коллекции, иначе возвращается false
- **boolean removeAll (Collection<?> col):** удаляет все объекты коллекции col из текущей коллекции. Если текущая коллекция изменилась, возвращает true, иначе возвращается false
- **boolean retainAll (Collection<?> col):** удаляет все объекты из текущей коллекции, кроме тех, которые содержатся в коллекции col. Если текущая коллекция после удаления изменилась, возвращает true, иначе возвращается false
- **int size ():** возвращает число элементов в коллекции
- **Object[] toArray ():** возвращает массив, содержащий все элементы коллекции

Collection

Все эти и остальные методы, которые имеются в интерфейсе Collection, реализуются всеми коллекциями, поэтому в целом общие принципы работы с коллекциями будут одни и те же.

Единообразный интерфейс упрощает понимание и работу с различными типами коллекций.

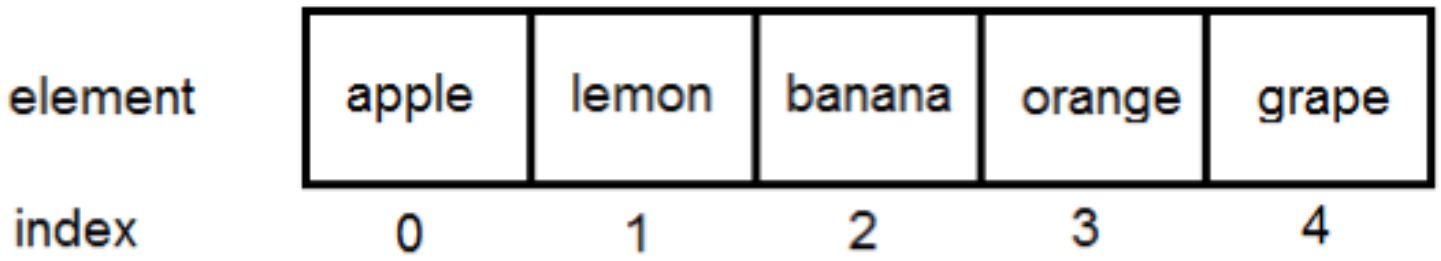
Так, добавление элемента будет производиться с помощью метода **add()**, который принимает добавляемый элемент в качестве параметра.

Для удаления вызывается метод **remove()**.

Метод **clear()** будет очищать коллекцию, а метод **size()** возвращать количество элементов в коллекции.

List

Для создания простых списков применяется интерфейс List, который расширяет функциональность интерфейса Collection.



List

Некоторые наиболее часто используемые методы интерфейса List:

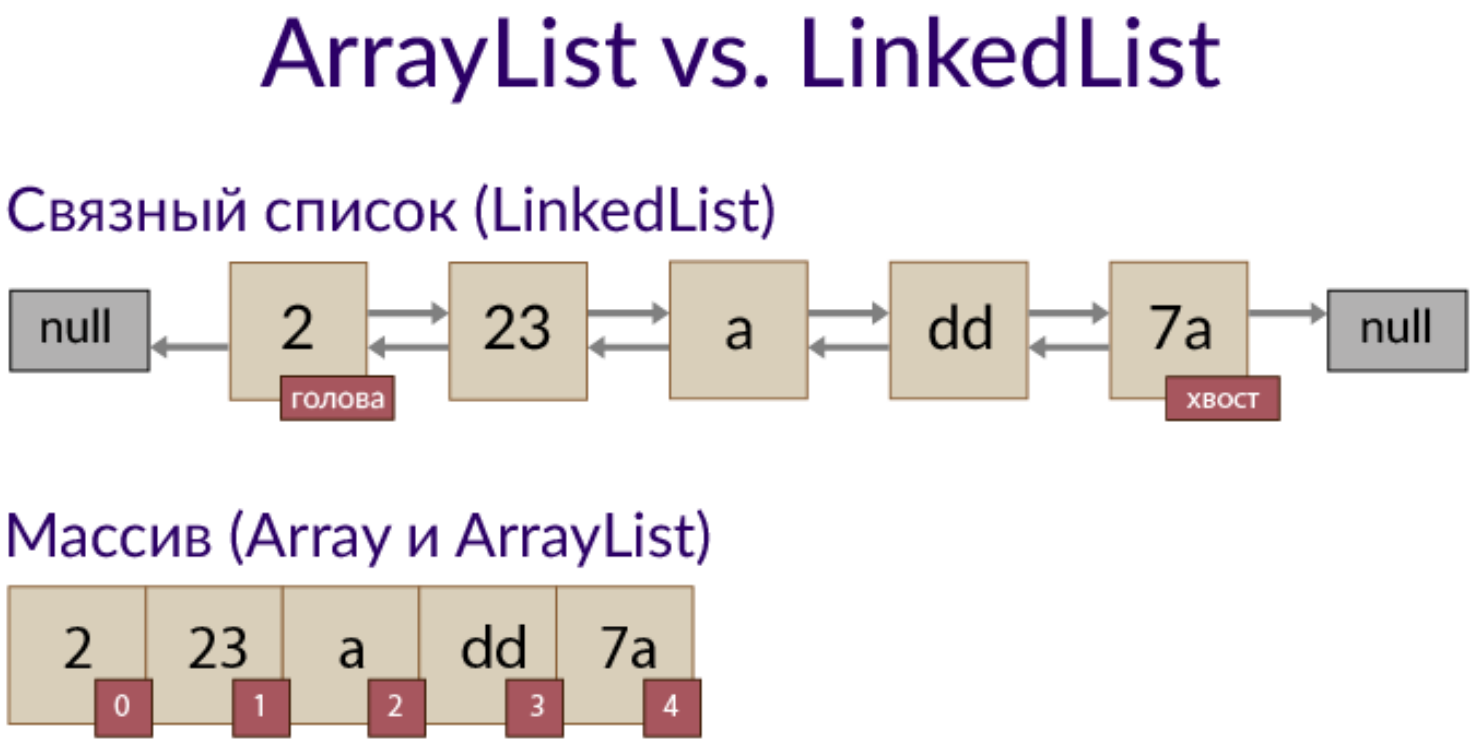
- **void add(int index, E obj):** добавляет в список по индексу index объект obj
- **boolean addAll(int index, Collection<? extends E> col):** добавляет в список по индексу index все элементы коллекции col. Если в результате добавления список был изменен, то возвращается true, иначе возвращается false
- **E get(int index):** возвращает объект из списка по индексу index
- **int indexOf(Object obj):** возвращает индекс первого вхождения объекта obj в список. Если объект не найден, то возвращается -1
- **int lastIndexOf(Object obj):** возвращает индекс последнего вхождения объекта obj в список. Если объект не найден, то

List

- **ListIterator<E> listIterator ():** возвращает объект ListIterator для обхода элементов списка
- **static <E> List<E> of(элементы):** создает из набора элементов объект List
- **E remove(int index):** удаляет объект из списка по индексу index, возвращая при этом удаленный объект
- **E set(int index, E obj):** присваивает значение объекта obj элементу, который находится по индексу index
- **void sort(Comparator<? super E> comp):** сортирует список с помощью компаратора comp
- **List<E> subList(int start, int end):** получает набор элементов, которые находятся в списке между индексами start и end

List

Две самые популярные реализации интерфейса List это **ArrayList** и **LinkedList**.



ArrayList & LinkedList

По сути они хранят данные в одинаковом виде, и имеют один и тот же набор методов. Зачем же тогда 2 реализации?

Все дело в том, что коллекции могут быть реализованы разными способами и нет единственного – самого правильного.

При одном подходе одни операции являются быстрыми, а остальные медленными, при другом – все наоборот. Нет одного идеального, подходящего всем решения.

Поэтому было решено сделать несколько реализаций одной и той же коллекции. И каждая реализация была оптимизирована для какого-то узкого набора операций.

ArrayList

ArrayList реализован внутри в виде обычного массива.

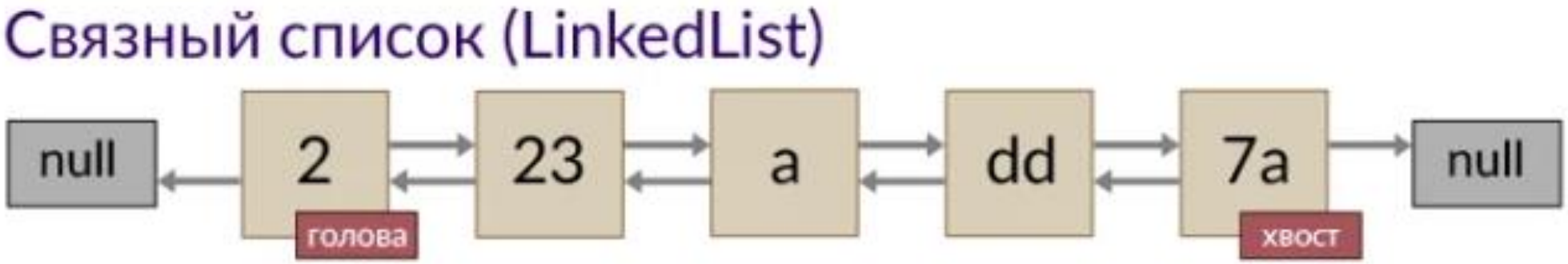
Поэтому при вставке элемента в середину, приходится сначала сдвигать на один все элементы после него, а уже затем в освободившееся место вставлять новый элемент. Это долго.

Зато в нем быстро реализованы взятие и изменение элемента – операции get, set, так как в них мы просто обращаемся к соответствующему элементу массива.

LinkedList

LinkedList реализован внутри по-другому.

Он реализован в виде **связного списка**: набора отдельных элементов, каждый из которых хранит ссылку на следующий и предыдущий элементы. Чтобы вставить элемент в середину такого списка, достаточно поменять ссылки его будущих соседей.



А вот чтобы получить 130-й по счету элемент, нужно пройтись последовательно по всем объектам от 0 до 130. Другими словами операции set и get тут реализованы очень медленно.

Пример

Вывод:

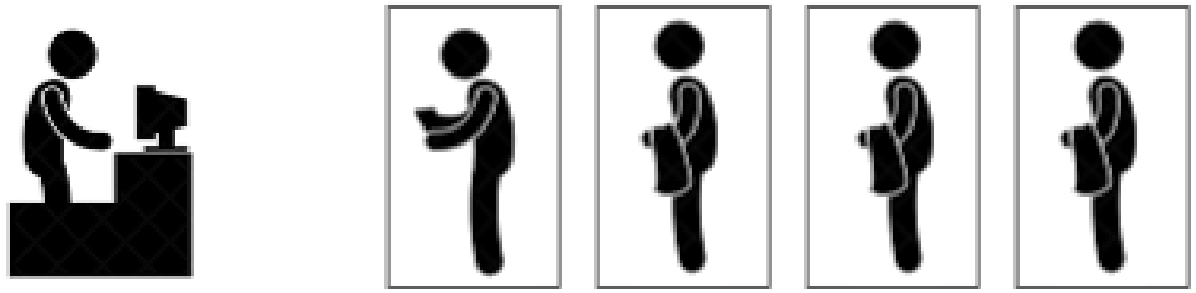
```
Bob
ArrayList has 5 elements
Tom
Robert
Alice
Kate
Sam
ArrayList contains Tom
Alice
Kate
Sam
```

```java
List<String> people = new ArrayList<>();
// добавим в список ряд элементов
people.add("Tom");
people.add("Alice");
people.add("Kate");
people.add("Sam");
people.add( index: 1,  element: "Bob"); // добавляем элемент по индексу 1

System.out.println(people.get(1));// получаем 2-й объект
people.set(1, "Robert"); // установка нового значения для 2-го объекта

System.out.printf("ArrayList has %d elements \n", people.size());
for(String person : people){

    System.out.println(person);
}
// проверяем наличие элемента
if(people.contains("Tom")){

    System.out.println("ArrayList contains Tom");
}

// удалим несколько объектов
// удаление конкретного элемента
people.remove( o: "Robert");
// удаление по индексу
people.remove( index: 0);

Object[] peopleArray = people.toArray();
for(Object person : peopleArray){

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

Queue

Очереди представляют структуру данных, работающую по принципу **FIFO (first in - first out)**.

То есть чем раньше был добавлен элемент в коллекцию, тем раньше он из нее удаляется. Это стандартная модель однонаправленной очереди.



По сути это аналог любой очереди в реальном мире. Первым стал в очереди, первым что-то купил и ушел.

Queue

Интерфейс Queue<E> расширяет базовый интерфейс Collection через следующие методы:

- **E element():** возвращает, но не удаляет, элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException

- **boolean offer(E obj):** добавляет элемент obj в конец очереди. Если элемент удачно добавлен, возвращает true, иначе - false

- **E peek():** возвращает без удаления элемент из начала очереди. Если очередь пуста, возвращает значение null

Queue

- **E poll():** возвращает с удалением элемент из начала очереди. Если очередь пуста, возвращает значение null

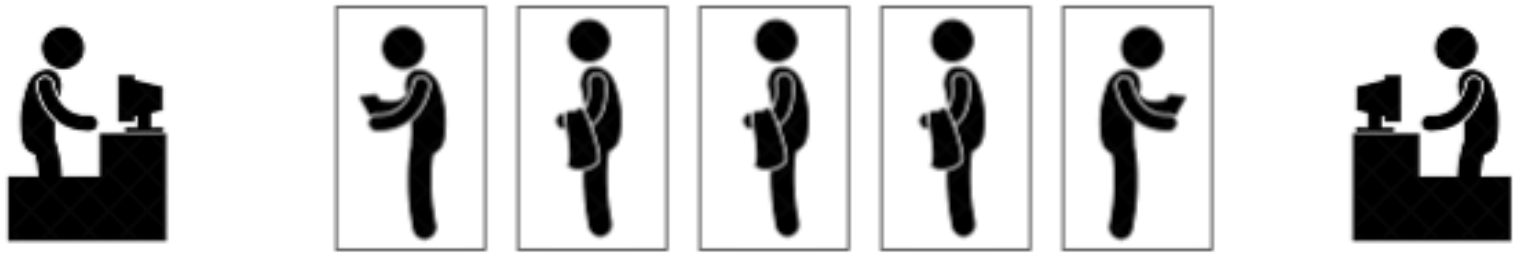
- **E remove():** возвращает с удалением элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException

Queue

Таким образом, у всех классов, которые реализуют данный интерфейс, будет метод **offer** для добавления в очередь, метод **poll** для извлечения элемента из головы очереди, и методы **peek** и **element**, позволяющие просто получить элемент из головы очереди.

Deque

Однако бывают и двунаправленные очереди - то есть такие, в которых мы можем добавить элемент не только в начала, но и в конец. И соответственно удалить элемент не только из конца, но и из начала.



Интерфейс Deque расширяет вышеописанный интерфейс Queue и определяет поведение двунаправленной очереди.

Как люди, выстраивающиеся в очередь в супермаркете, обслуживаются только первый и последний человек в очереди.

Deque

Интерфейс Deque определяет следующие методы:

- **void addFirst(E obj):** добавляет элемент в начало очереди
- **void addLast(E obj):** добавляет элемент obj в конец очереди
- **E getFirst():** возвращает без удаления элемент из головы очереди. Если очередь пуста, генерирует исключение NoSuchElementException
- **E getLast():** возвращает без удаления последний элемент очереди. Если очередь пуста, генерирует исключение NoSuchElementException
- **boolean offerFirst(E obj):** добавляет элемент obj в самое начало очереди. Если элемент удачно добавлен, возвращает true, иначе - false

Deque

- **boolean offerLast(E obj):** добавляет элемент obj в конец очереди. Если элемент удачно добавлен, возвращает true, иначе - false
- **E peekFirst():** возвращает без удаления элемент из начала очереди. Если очередь пуста, возвращает значение null
- **E peekLast():** возвращает без удаления последний элемент очереди. Если очередь пуста, возвращает значение null
- **E pollFirst():** возвращает с удалением элемент из начала очереди. Если очередь пуста, возвращает значение null
- **E pollLast():** возвращает с удалением последний элемент очереди. Если очередь пуста, возвращает значение null

Deque

- **E pop():** возвращает с удалением элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException
- **void push(E element):** добавляет элемент в самое начало очереди
- **E removeFirst():** возвращает с удалением элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException
- **E removeLast():** возвращает с удалением элемент из конца очереди. Если очередь пуста, генерирует исключение NoSuchElementException
- **boolean removeFirstOccurrence(Object obj):** удаляет первый встреченный элемент obj из очереди. Если удаление произшло, то возвращает true, иначе возвращает false.
- **boolean removeLastOccurrence(Object obj):** удаляет последний встреченный элемент obj из очереди. Если удаление произшло, то возвращает true, иначе возвращает false.

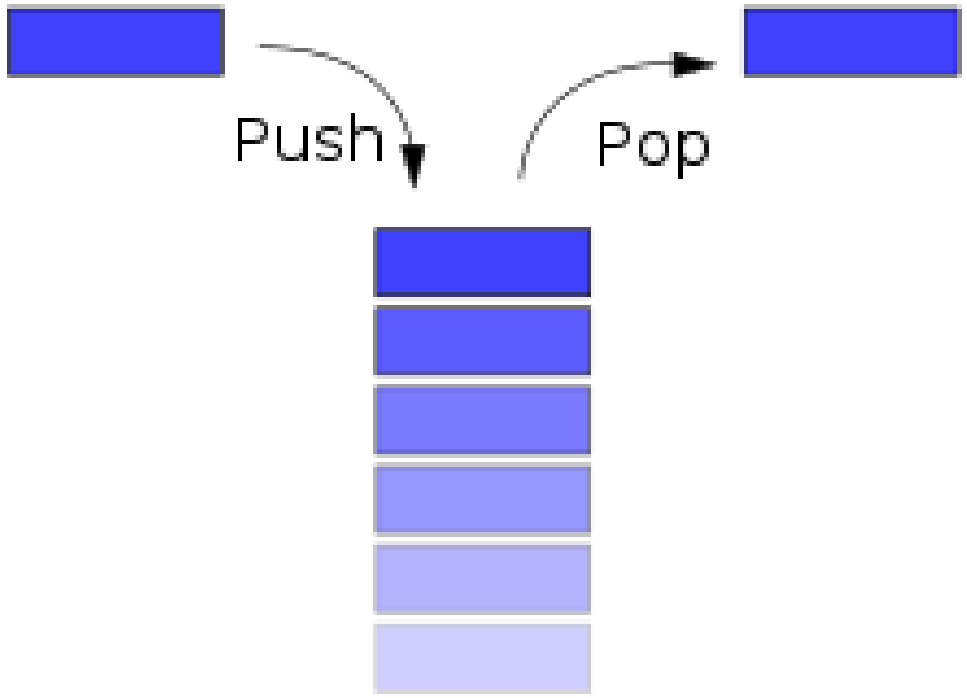
Deque

Таким образом, наличие методов **pop** и **push** позволяет классам, реализующим этот элемент, действовать в качестве стека.

В тоже время имеющийся функционал также позволяет создавать двунаправленные очереди, что делает классы, применяющие данный интерфейс, довольно гибкими.

Deque

Deque может действовать как **stack (стек)**, поскольку он предоставляет методы для работы в рамках механизма **LIFO (Last In First Out)** (последний добавленный элемент будет извлечен первым).



ArrayDeque

ArrayDeque - это класс в Java, который реализует **Deque** интерфейс.

Это специальный класс, который реализует двустороннюю структуру данных очереди, где он может вставлять и удалять элементы с обоих концов.

Он поддерживает реализацию автоматически растущего массива с изменяемым размером.

ArrayDeque

Вставка элементов в ArrayDeque

Мы можем вставлять элементы в ArrayDeque в Java, используя методы **add()** или **offer()**. Для вставки коллекции элементов мы можем использовать метод **addAll()**. Чтобы вставить значение в начало, используйте метод **addFirst()**, **offerFirst()** или **push()**, тогда как для вставки значений в конце мы можем использовать метод **addLast()** или **offerLast()**.

ArrayDeque

Удаление элементов из ArrayDeque

Мы можем удалять элементы из ArrayDeque с помощью различных методов. Методы **remove()**, **removeFirst()**, **poll()**, **pollFirst()** и **pop()** удаляют первый элемент в двухсторонней очереди. **RemoveLast()** и **pollLast()** удаляют последнее значение в двухсторонней очереди. Чтобы удалить все элементы, кроме коллекции указанных элементов, мы можем использовать метод **keepAll()**, а для удаления всех элементов в коллекции мы можем использовать метод **removeAll()**.

ArrayDeque

Доступ к элементам ArrayDeque

Чтобы проверить наличие элемента, используйте метод **contains()**. Он возвращает истину, если значение существует, иначе возвращает ложь. Чтобы получить доступ к первому элементу, мы можем использовать методы **element()**, **getFirst()**, **peek()** или **peekFirst()**, тогда как для получения последнего значения мы можем использовать методы **getLast()** или **peekLast()**.

Пример

```
public static void main(String[] args) {

    ArrayDeque<String> states = new ArrayDeque<String>();
    // стандартное добавление элементов
    states.add("Germany");
    states.addFirst("France"); // добавляем элемент в самое начало
    states.push("Great Britain"); // добавляем элемент в самое начало
    states.addLast("Spain"); // добавляем элемент в конец коллекции
    states.add("Italy");

    // получаем первый элемент без удаления
    String sFirst = states.getFirst();
    System.out.println(sFirst);     // Great Britain
    // получаем последний элемент без удаления
    String sLast = states.getLast();
    System.out.println(sLast);      // Italy

    System.out.printf("Queue size: %d \n", states.size());  // 5
```