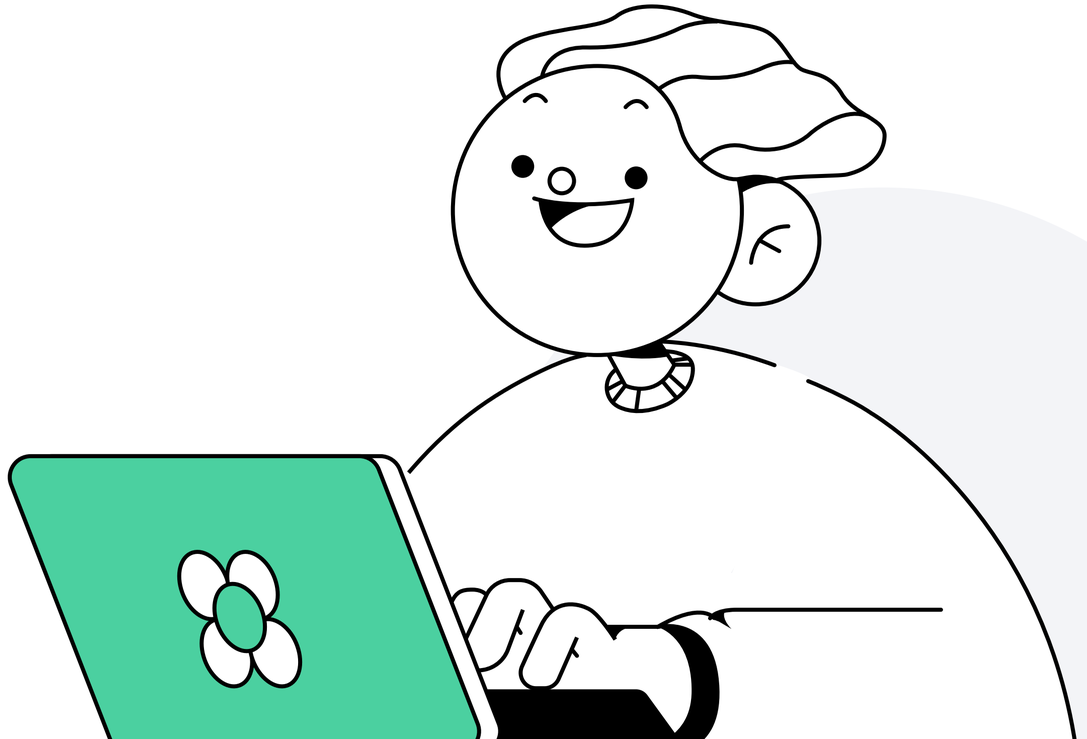
Коллекции Queue

План занятия

1. Очереди и PriorityQueue
2. Деки и стеки

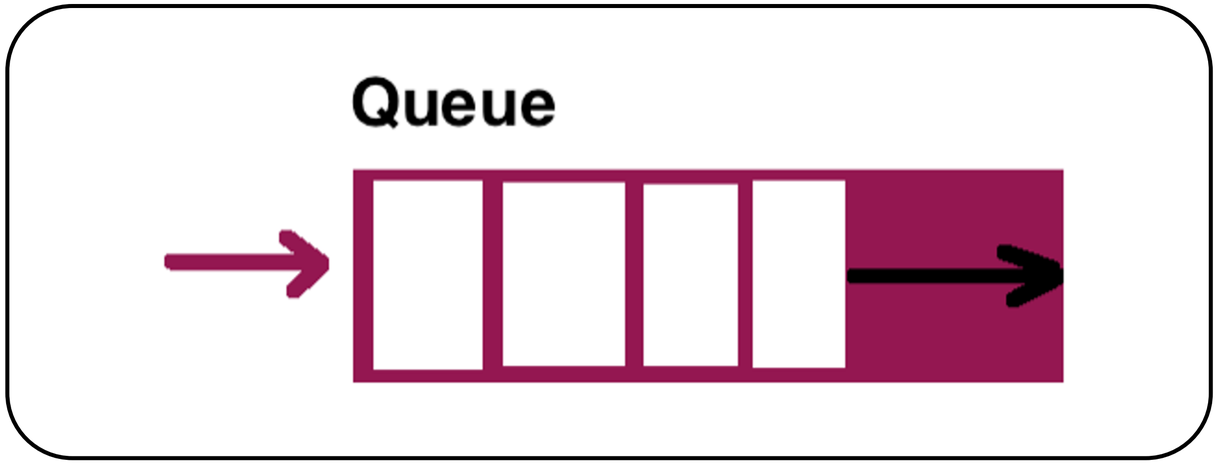


Очереди и PriorityQueue

Структура данных Queue — очередь

Структура, реализующая принцип **First In**, **First Out** (первый вошёл, первый вышел). Это значит, что тот элемент, который был первым добавлен в коллекцию, будет первым из неё извлечён, при этом очереди не могут содержать **null**.

Документация Interface Queue<E>



Методы Queue

Queue в Java — это интерфейс однонаправленной очереди, наследуемый от общего интерфейса **Collection** и предоставляющий для реализации следующие методы:

- **add(e)** — добавляет элемент типа **T** в конец очереди; при успешном добавлении возвращает **true**, при неудачном — выбрасывает соответствующее исключение
- **T element()** — возвращает первый элемент из очереди типа **T**; если в очереди нет элементов, выбрасывает исключение **NoSuchElementException**. При этом элемент остаётся в очереди
- **T remove()** — возвращает первый элемент типа **T** из очереди, при этом удаляет из неё этот элемент; если элементов в очереди нет, выбрасывает исключение **NoSuchElementException**

Методы Queue

- **boolean offer(T object)** — добавляет элемент типа **T** в конец очереди; при успешном добавлении возвращает **true**, при неудачном — **false**
- **T peek()** — возвращает первый элемент типа **T** из очереди без последующего удаления элемента из неё; если в очереди нет элементов, метод возвращает **null**
- **T poll()** — возвращает первый элемент типа **T** из очереди, при этом удаляет из неё элемент; если в очереди нет элементов, метод возвращает **null**

T — тип данных, который будет храниться в очереди, может быть любым. Его также можно опустить при объявлении очереди, тогда она будет хранить все объекты, приводя их к типу Object, и при извлечении любого элемента нужно будет приводить его к ожидаемому типу

Важно отметить

- Методы **add**, **element** и **remove** требуют обработки исключений
- Методы **remove** и **poll** одновременно возвращают и удаляют элементы из очереди
- Методы **peek** и **element** возвращают элементы из очереди, оставляя их на том же месте

Методы, наследуемые от интерфейса Collection

- addAll
- clear
- contains
- containsAll
- equals
- hashCode
- isEmpty
- iterator
- removeIf
- retainAll
- size
- toArray
- remove
- removeAll
- spliterator
- stream
- parallelStream

Скорость работы

Вставка элемента в конец очереди
и извлечение элемента из её начала
константно **O(1)**

Класс AbstractQueue

Как уже говорилось ранее, **Queue** — это интерфейс, описывающий контракт однонаправленной очереди без какой-либо реализации.

Стандартная библиотека Java предоставляет разработчикам реализацию этих методов по умолчанию с помощью класса **AbstractQueue**, который, в свою очередь, реализует все методы интерфейса **Queue**.

Документация Class AbstractQueue<E>

Примеры реализации AbstractQueue в JDK

На основе класса **AbstractQueue** в стандартной библиотеке Java реализованы следующие типы очередей:

1. ArrayBlockingQueue
2. ConcurrentLinkedQueue
3. DelayQueue
4. LinkedBlockingDeque
5. LinkedBlockingQueue
9. LinkedTransferQueue
10. PriorityBlockingQueue
11. PriorityQueue
12. SynchronousQueue

Класс PriorityQueue

Класс **PriorityQueue** — единственный класс, наследуемый только от интерфейса **Queue** и реализующий его.

Особенность этой очереди — возможность задавать порядок элементов в ней при вставке на основе сортировки.

По умолчанию элементы сортируются с использованием «natural ordering». Если нужно изменить порядок элементов, необходимо передать при создании очереди объект **Comparator**, который будет сравнивать объекты при добавлении и выставлять их в порядке сортировки. Эта коллекция не поддерживает хранение **null** в качестве элемента.

Документация **Class PriorityQueue<E>**

Пример использования PriorityQueue

Давайте рассмотрим пример отправки по электронной почте новогодних поздравлений для клиентов.

Если клиентов много (100 тысяч), наш сервис может не справиться с нагрузкой. В таком случае нам поможет очередь **PriorityQueue**

Пример использования PriorityQueue

Не нужно вручную следить за размером очереди и удалять из неё элементы, метод **poll** сделает это за нас.

Ничто не мешает использовать список (например **ArrayList**), но это будет неудобно в работе: нам придётся вручную добавлять и удалять элементы из него.

Плюсы очередей в том, что мы можем в один конец добавлять элементы, а из другого читать, тем самым балансируя нагрузку между источником и потребителем (использовать как буфер между ними)

Пример использования PriorityQueue

Положим все письма в эту коллекцию, а наш сервис будет забирать из неё сообщения и постепенно отправлять их:

```
PriorityQueue<PostMessage> queue = new PriorityQueue<>();
queue.offer(message1);                    //добавляем сообщение в очередь
queue.offer(message2);                    //добавляем ещё одно сообщение в очередь
...
PostMessage message = queue.poll();      //извлекаем message1 из очереди
System.out.println(message.toString()); //выводим сообщение
```

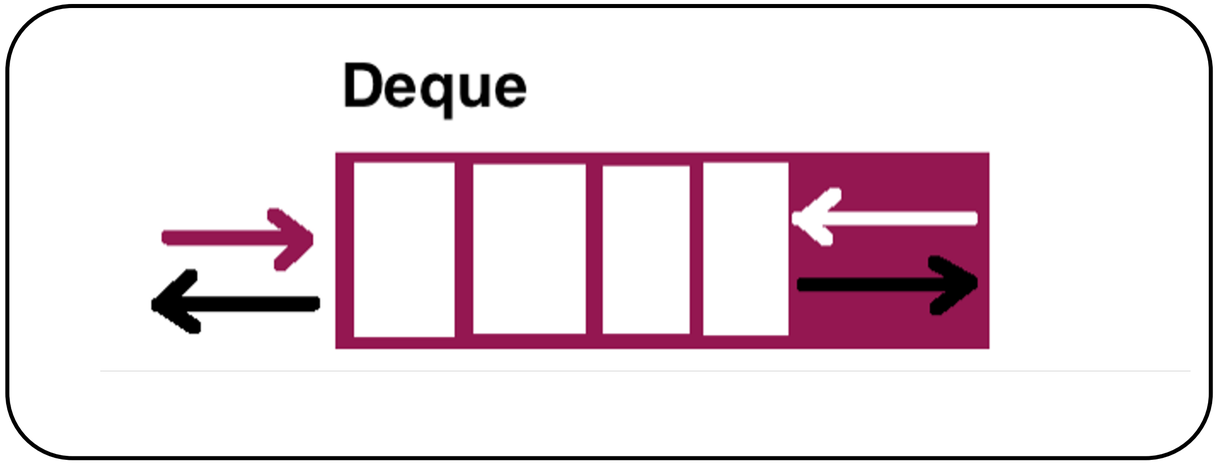
Деки и стеки

Интерфейс Deque

Ещё один важный интерфейс, наследуемый от **Queue**, — **Deque**.

Этот тип расширяет однонаправленную очередь до двунаправленной и позволяет работать в режиме не только **First In**, **First Out**, но и **Last In**, **First Out** (последний вошёл, первый вышел).

Документация Interface Deque<E>



Собственные методы интерфейса Deque

- **addFirst(T e), addLast(T e)** — методы добавляют новый элемент в начало или конец соответственно. Если добавить элемент не удалось, выбрасывают исключение **IllegalStateException**
- **T getFirst(), T getLast()** — методы возвращают элемент очереди с начала или с конца соответственно. Элемент при этом из очереди не удаляется. Если элемента нет (очередь пуста), выбрасывают исключение **NoSuchElementException**
- **removeFirst(), removeLast()** — методы удаляют элемент с конца или начала соответственно. Если удалить не удалось, выбрасывают исключение **NoSuchElementException**
- **offerFirst(T e), offerLast(T e)** — методы добавляют новый элемент в начало или в конец соответственно. Если добавить элемент удалось, возвращают **true**, иначе — **false**

Собственные методы интерфейса Deque

- **T peekFirst(), T peekLast()** — методы возвращают элемент с начала или с конца очереди соответственно. Элемент при этом из очереди не удаляется. Если очередь пуста, возвращают **null**
- **T pollFirst(), T pollLast()** — методы возвращают элемент с начала или с конца очереди соответственно. Элемент после возвращения удаляется из очереди. Если очередь пуста, возвращают **null**

Примеры реализации Deque в JDK

На основе интерфейса **Deque** в стандартной библиотеке Java есть коллекции:

1. ArrayDeque
2. ConcurrentLinkedDeque
3. LinkedBlockingDeque
4. LinkedList

Наиболее часто используемые коллекции из этого списка: **ArrayDeque, LinkedList**

Пример использования ArrayDeque

Рассмотрим пример.

У нас есть список сообщений, которые мы хотим обработать. В какой-то момент при обработке одного из сообщений возникает ошибка, и мы хотим вернуть сообщение в список (очередь) и обработать позже. В этом случае нам поможет Deque: мы сможем вернуть сообщение в начало, и оно по-прежнему будет первым в очереди на обработку

Пример использования ArrayDeque

Давайте рассмотрим преимущество двунаправленной очереди (Deque) перед списком (ArrayList) и однонаправленной очередью (Queue).

Можно воспользоваться любой из трёх коллекций. Минус списка (**ArrayList**), как и в предыдущей задаче, — ручное добавление и удаление элементов. Минус однонаправленной очереди (**Queue**) — возможность вернуть элемент только в конец очереди.

Следовательно, если нам нужно сохранить очерёдность обработки элементов (последовательность), следует использовать двунаправленную очередь, чтобы иметь возможность вернуть элемент в начало очереди

Пример использования ArrayDeque

Для добавления элемента в начало очереди воспользуемся методом интерфейса **Deque.addFirst**

```
Deque<String> deque = new ArrayDeque<>();

deque.add("Message 1");
deque.add("Message 2");
deque.add("Message 3");
deque.add("Message 4");
deque.add("Message 5");

String message = deque.poll(); //poll Message 1

System.out.println(deque);
//Вывод: [Message 2, Message 3, Message 4, Message 5]

//Программа не может обработать наше сообщение message, и мы хотим вернуть его обратно в начало очереди
deque.addFirst(message);

System.out.println(deque);
//Вывод: [Message 1, Message 2, Message 3, Message 4, Message 5]
```

LinkedList vs ArrayDeque

В примере из предыдущего слайда мы могли бы использовать класс из стандартной библиотеки Java — **LinkedList**, если бы не знали следующих отличий между классами **LinkedList** (сделан на основе списка) и **ArrayDeque** (сделан на основе массива):

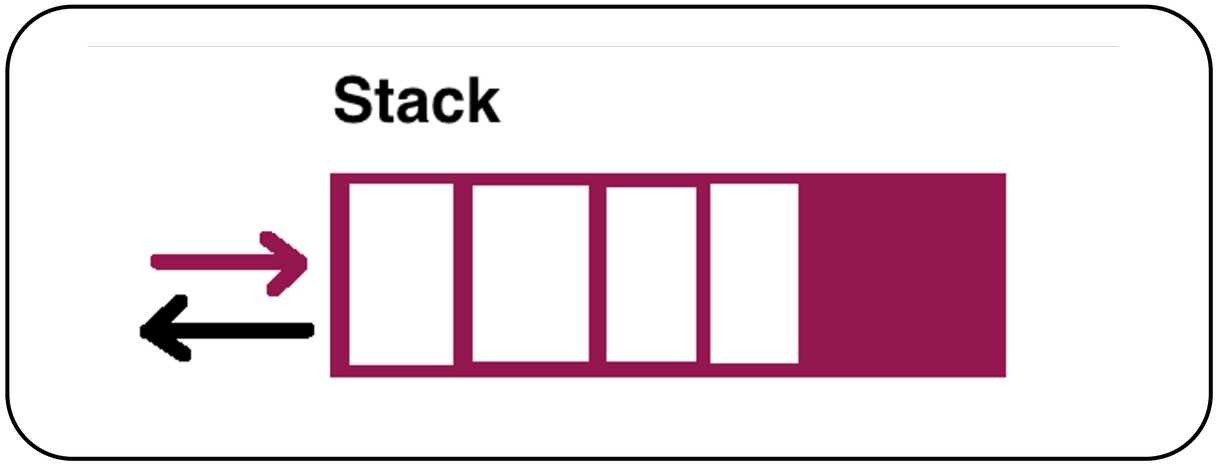
- производительность вставки и удаления элементов с любого конца у **ArrayDeque** амортизирована (на большом количестве рассматриваемых случаев) выше, чем у **LinkedList**
- для каждого элемента **LinkedList** выделяется больше памяти, следовательно, вся коллекция занимает больше памяти

Подробнее об имплементации деков

Класс Stack

Мы рассмотрели две очереди на основе интерфейсов **Queue** и **Deque**. В Java с ранних версий реализован класс **Stack**, который действует по концепции **Last In**, **First Out** (последний вошёл, первый вышел).

Класс **Stack** реализован на основе класса **Vector**. В этом его недостаток, так как все операции в классе **Vector** синхронизированные, а значит медленные



Методы Stack

- **boolean empty()** — возвращает **true** или **false**, проверяя, есть ли элементы в коллекции
- **T peek()** — возвращает последний элемент типа **T** из стека без последующего удаления элемента из него; если в стеке нет элементов, метод возвращает **null**
- **T pop()** — возвращает последний элемент типа **T** из стека и при этом удаляет из него элемент; если в стеке нет элементов, метод возвращает **null**
- **boolean push(T object)** — добавляет элемент типа **T** в конец стека; при успешном добавлении возвращает **true**, при неудачном — **false**
- **int search(Object o)** — возвращает позицию элемента в стеке; если элемента нет, метод возвращает **–1**

Документация Class Stack<E>

Важный момент

Документация языка Java не рекомендует использовать класс Stack. Вместо него предлагается применять двунаправленную очередь Deque. Пример объявления:

Deque<Integer> stack = new ArrayDeque<Integer>()