Модуль 4. Алгоритмы и структуры данных

Тема 4.2. Списки

4 часа

Оглавление

4.2. Списки

4.2.1 Понятие списка

Список, как базовая структура данных

Ранее , когда у нас имелись однотипные данные, которые несут информацию одинакового характера, мы использовали массивы. Но у массивов есть недостаток: в Java размер массива после его создания нельзя изменить. Если заранее неизвестно какого размера понадобится массив, то придется выделять память “с запасом” и при этом нет гарантии того, что ее хватит. В итоге главное негативное последствие - неэффективное использование памяти.

Как раз для таких случаев и придуманы списки и другие структуры данных, о которых пойдет речь в Модуле 4.

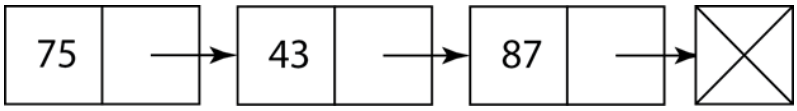
Списки и массивы сходны в том, что могут использоваться для хранения аналогичных по типу элементов. Главное же отличие между ними - в способе их хранения, и как следствие, их обработки.

Мы помним, что когда вы создаете массив, то найти следующий элемент просто: память выделяется последовательно под каждый последующий элемент массива (с точки зрения программиста память будет выделена сразу в нужном объеме). Поэтому, зная адрес начального элемента и тип элементов массива (что эквивалентно знанию о размере в битах на каждый элемент), программа может с легкостью найти любой элемент массива.

Со списками такое не пройдет, так как память под элемент выделяется динамически и только после этого программа “узнает” адрес его размещения и при добавлении в список запоминает, ссылку на него как минимум в одном из элементов. Таким образом список - это цепочка разрозненных кусочков памяти. Причем во время выполнения программы элементы могут не только добавляться, но и удаляться, переставляться поэтому эта цепочка постоянно меняется.

Итак, в случае с массивами память выделяется сразу под весь массив, а в случае списков по мере добавления/удаления элементов.

Следует обратить внимание, что здесь под списком мы понимаем не общее понятие списка из информатики, а понятие **связного списка**. Это следует учитывать при поиске дополнительной информации в литературе и интернете.



Список – это динамическая структура, состоящая из элементов, каждый из которых содержит в себе данные и ссылки (одну или две) на последующий или (и) предыдущий элемент. Один элемент при этом может содержать информацию о том, что он последний элемент в списке и один элемент - что он первый в списке.

Представьте, если каждый элемент структуры данных сможет содержать ссылки на несколько других элементов, тогда можно сформировать нелинейную структуру данных (не список): дерево или граф. Как видите, такой подход позволяет создавать огромное разнообразие структур данных, которые будут отличаться правилами, по которым строятся связи между элементами и способом их обработки.

Java известен своей богатой библиотекой, в частности есть множество классов и интерфейсов для работы с структурами данных. В этой теме мы начнем рассмотрение некоторых из них.

4.2.2. Классификация структур данных, основанных на списках

Списки бывают нескольких видов, так что их можно классифицировать по нескольким признакам. Рассмотрим различные классификации, которые встречаются в литературе.

Классификация по способу связи элементов списка. Элемент связного списка всегда содержит информацию об адресе последующего элемента. При этом, если список - это незамкнутая цепочка, т.е. последний элемент списка в качестве адреса следующего элемента содержит пустую ссылку (null) Такой список называют **линейным**.

Если же последний элемент списка в поле ссылки на следующий элемент содержит адрес головы списка (первого элемента), то такой список называют **циклическим**. Далее мы будем вести речь только о линейных списках.

Списки можно делить на основе разных принципов классификации:

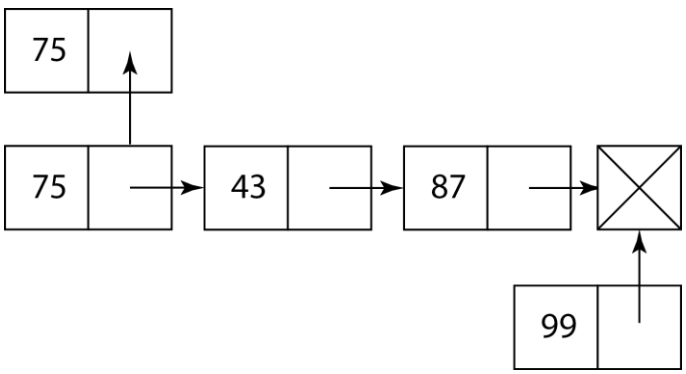
- **Классификация по дисциплине обслуживания**, то есть по способу добавления и прочтения элементов. По данной классификации списки как правило делят на очереди и стеки.
- **Классификация по количеству ссылок в элементе** списка делит их на односвязные и двусвязные списки.

Эти классификации не влияют друг на друга, т.е. существуют одновременно, к примеру, бывают двусвязные очереди и односвязные стеки.

Очередь

Очередь – это список, в котором добавление элементов происходит в хвост (конец списка), а считывание элементов происходит с головы. Такая дисциплина обслуживания «**первый пришел - первым ушел**» носит название **FIFO** (First In — First Out).

Это полностью соответствует обычной очереди людей, например, в кассу супермаркета. Когда люди подходят, то они встают в конец очереди, а кассиры обслуживают тех, кто первыми стоит в данной очереди. И только попробуйте нарушить дисциплину и пройти без очереди!



Стек

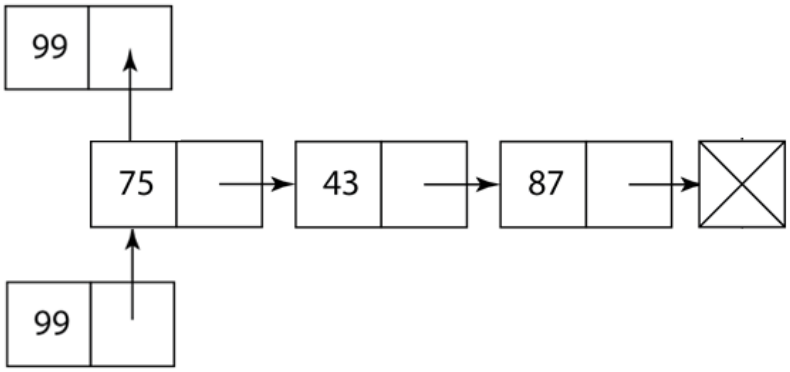
Стек – это список, в котором добавление и считывание реализуется в соответствии с обратной очереди дисциплиной обслуживания «**последним пришел - первым ушел**». Этот принцип носит название **LIFO** (Last in — first out).

Чтобы понять работу стека, представьте стеклянную колбу, в которую закладываются цветные шайбы (см. рисунок). Если положить первой желтую шайбу, а потом синюю, то понятно, что желтую мы можем достать только после синей.

Получается стопка шайб, в которой сверху лежит последняя положенная шайба. В стеке принято называть ее **вершиной стека**. При реализации стека с помощью списков удобно в качестве указателя на голову списка использовать указатель на вершину. И тогда добавление и удаление элементов в стек легко реализуется со стороны “головы” и нет необходимости пробегать весь список.

Другой пример стека – это магазин автомата с патронами. Последним сработает патрон, который в магазин попал самым первым.

В программировании общеизвестно понятие стека вызовов функций. Его работа, как следует из названия, также основана на структуре данных “стек”.

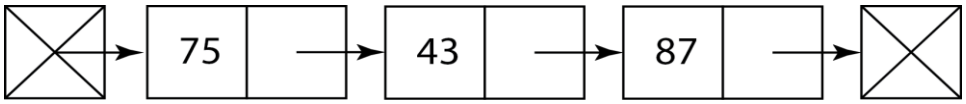


Нельзя сказать, что очередь лучше стека или наоборот - для решения каждой конкретной задачи выбирается та структура данных, которая подходит лучше.

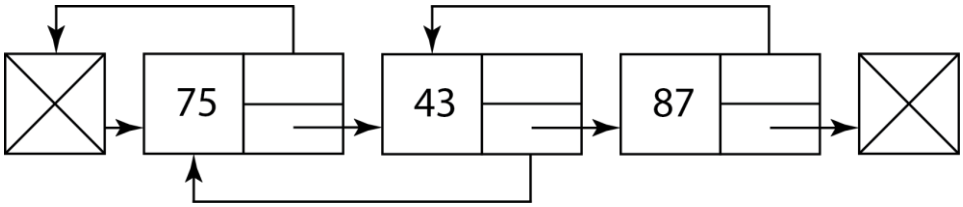
Двусвязные и односвязные списки

Со второй классификацией списков всё просто.

Односвязные списки – это списки, в которых каждый элемент имеет только одну ссылку, как правило, на следующий элемент в данной структуре данных.



Двусвязные списки – это списки, в которых каждый элемент имеет две ссылки: на предыдущий и следующий за ним элементы списка.



Таким образом каждый элемент односвязного списка “знает” только элемент который стоит после него, а каждый элемент двусвязного списка еще дополнительно “знает” элемент, который стоит перед ним. Следовательно в первом случае мы можем двигаться по списку только в одном направлении от головы списка к хвосту, тогда как во втором - в обоих направлениях.

Кроме указанных разновидностей списков выделают также некоторые другие структуры данных, которые относят к спискам:

- *Список с пропусками – связи между элементами строятся в сложном порядке с элементами случайности.*
- *Развернутый связный список – в списке вместо элементов находятся массивы элементов.*

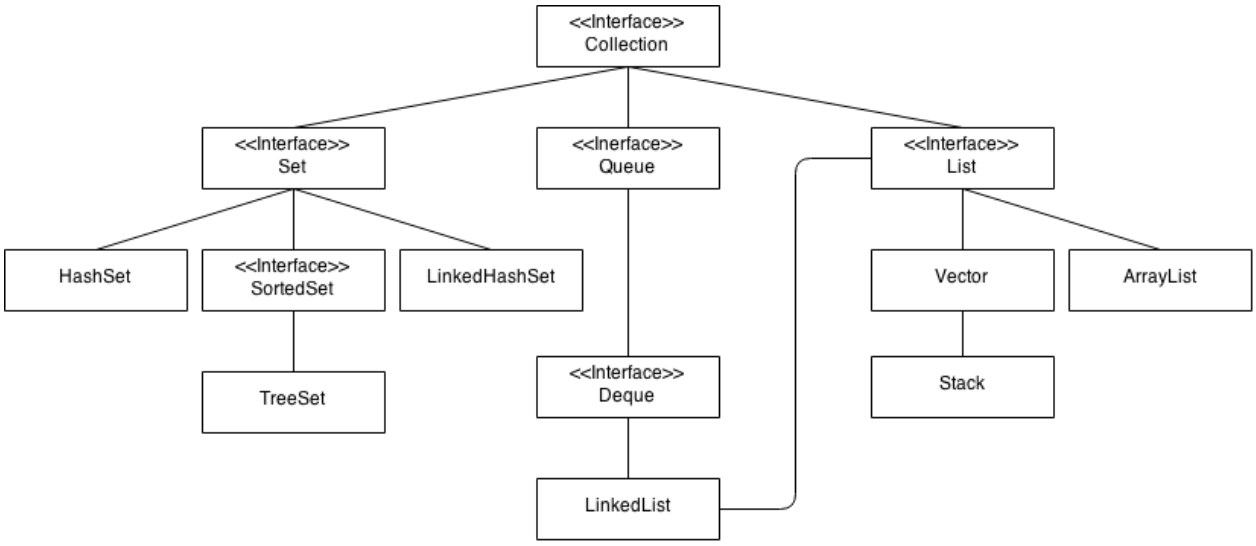
4.2.3 Библиотечный класс LinkedList, Queue, Stack

Коллекции Java

В Java существует большое разнообразие классов, реализующих структуры данных, поэтому для универсальности их представления введено понятие коллекции. Коллекция — абстрактная структура данных, которая оперирует группой объектов.

Стандартные коллекции расположены в библиотеке java.util. Использование коллекций дает широкие возможности при программировании, так как каждая коллекция эффективно справляется со своей задачей, поэтому чтобы не “изобретать велосипед” каждый программист должен знать основные коллекции языка, который он изучает.

Ниже на рисунке приведена схема некоторых классов и интерфейсов по структурам данных (данная схема является упрощенной и неполной).



Если мы посмотрим на схему, то увидим, что главным классом[1] является интерфейс **Collection**, который описывает коллекцию объектов без какой-то конкретики по реализации данного набора объектов.

На самом деле Collection не является корневым классом. Collection наследуется от класса Iterable, который определяет наличие итератора. Итератор по своим функциям похож на указатель. Он позволяет обеспечивать доступ к любым элементам некоторого контейнера (а нашем случае в Collection) с сокрытием внутренней реализации считывания элементов в данном контейнере.

В итераторе можно переходить от элемента к элементу списка через метод next().

Далее у нас идет разделение на три основных интерфейса:

- Queue - структура данных очередь, в дополнение к базовым операциям интерфейса Collection предоставляет дополнительные операции вставки, получения элемента и контроля;
- **List** - упорядоченная коллекция (элементы отсортированы) иногда называемая списком или последовательностью. Список может содержать повторяющиеся элементы, предусматривает возможность получения доступа к элементам списка по индексу;
- **Set** - не рассмотренная нами структура данных “множество”, которая будет разбираться в последующих темах;
- Интерфейс **Deque** реализует двунаправленную (двусвязную) очередь, которая позволяет добавлять и удалять объекты с двух концов списка.
- **LinkedList** - класс, реализующий упорядоченную двунаправленную очередь.

Обратите внимание, что если в теоретической части мы делили списки на очереди и стеки, то в реализации Java стек и очередь находятся немного на разных иерархических уровнях. Позже мы разберемся, в чем тут дело.

Класс LinkedList

Перейдем к рассмотрению функционала данного класса. Вначале рассмотрим список методов данного класса.

Методы класса LinkedList

`LinkedList()`	Конструктор, создающий пустой список.
`LinkedList(Collection<? extends E> c)`	Конструктор, который в качестве параметра берет какую-то коллекцию, содержимым которой и заполняет список.
`boolean add(E object)`	Добавляет элемент в конец списка.
`void add(int location, E object)`	Добавляет элемент в указанное место в списке.
`boolean addAll(Collection<? extends E> collection)`	Добавляет в конец списка все элементы из коллекции. То есть из какой-то структуры данных вставляем все элементы. Структура данных не обязательно должна быть

[1]Здесь и далее понятия класс и интерфейс не разделяются: предполагается, что разница между ними, а также их общие свойства известны

	списком LinkedList.
`boolean addAll(int location, Collection<? extends E> collection)`	Добавляет все элементы из коллекции в указанное место.
`void addFirst(E object)`	Добавляет элемент в начало списка.
`void addLast(E object)`	Добавляет элемент в конец списка. Данный метод эквивалентен методу add.
`void clear()`	Очищает список от всех элементов.
`Object clone()`	Возвращает клонированный список типа LinkedList.
`boolean contains(Object object)`	Проверяет наличие элемента в параметрах в данном списке.
`Iterator<E> descendingIterator()`	Возвращает итератор, который позволяет пройтись по элементам в обратном порядке.
`E element ()`	Получает, но не удаляет первый элемент из очереди. Отличается от метода peek тем, что в случае пустого списка кидает исключение.
`E get(int location)`	Возвращает элемент по соответствующему номеру
`E getFirst()`	Возвращает первый элемент из списка.
`E getLast()`	Возвращает последний элемент из списка.
`int indexOf(Object object)`	Проверяет наличие элемента в параметрах в данном списке и возвращает номер первого вхождения в списке.
`int lastIndexOf(Object object)`	Аналогичен предыдущему методу, но возвращает номер последнего вхождения элемента в списке. Если элемент встречается в списке только один раз, то оба метода вернут одно и то же число.
`ListIterator<E> listIterator(int location)`	Возвращает список-итератор данного списка с указанного номера.
`abstract ListIterator<E> listIterator()`	Аналогично предающему методу, но тут начинается обход с первого элемента.
`boolean offer(E o)`	Добавляет в конец списка элемент. Отличается от метода add тем, что метод add при ошибке выдает исключение, а метод offer возвращает false.
`boolean offerFirst (E e)`	Добавляет элемент в начало списка.
`boolean offerLast (E e)`	Добавляет элемент в конец списка.
`E peek ()`	Аналогичен методу element, но в случае пустого списка не кинет исключение, а возвратит null.

`E` **`peekFirst`** `()`	Эквивалентен предыдущему методу
`E` **`peekLast`** `()`	Получает, но не удаляет последний элемент списка.
`E` **`poll`** `()`	Получает и удаляет первый элемент из списка. Если ничего нет, то возвращает null.
`E` **`pollFirst`** `()`	Аналогичен предыдущему методу.
`E` **`pollLast`** `()`	Получает и удаляет последний элемент из списка. Если ничего нет, то возвращает null.
`E` **`pop`** `()`	«Выталкивает» элемент и стека, который представлен нашей очередью. Иными словами, удаляет первый элемент из списка. Эквивалентен методу removeFirst.
`void` **`push`** `(E e)`	«Вталкивает» элемент в стек, представленный нашей очередью. Иными словам вставляет элемент в начало очереди. Эквивалентен методу addFirst.
`E` **`remove`** `()`	Удаляет первый элемент в списке. Эквивалентен методу removeFirst.
`E` **`remove`** `(int location)`	Удаляет элемент из списка из указанного места.
`boolean` **`remove`** `(Object object)`	Удаляет один экземпляр элемента object в списке. Эквивалентен методу removeFirstOccurrence.
`E` **`removeFirst`** `()`	Удаляет первый элемент в списке.
`boolean` **`removeFirstOccurrence`** `(Object o)`	Удаляет первое вхождение элемента в списке.
`E` **`removeLast`** `()`	Удаляет последний элемент в списке.
`boolean` **`removeLastOccurrence`** `(Object o)`	Удаляет последнее вхождение элемента в список
`E` **`set`** `(int location, E object)`	Заменяет содержимое элемента списка под соответствующим номером.
`int` **`size`** `()`	Возвращает число элементов в списке.
`T[]` **`toArray`** `(T[] contents)`	Возвращает массив элементов из списка. Используется, когда нам нужно использовать данные не в виде списка, а в виде массива. Если массив, указанный в параметрах способен вместить все элементы списка, то будет он использоваться, иначе создается новый массив.
`Object[]` **`toArray`** `()`	Аналогичен предыдущему методу, но тут сразу массив создается без массива от пользователя.

Обратите внимание, что есть некоторые методы (например, **add** и **offer**), которые по смыслу очень схожи, но отличаются незначительными деталями. А некоторые методы вообще эквиваленты друг другу (например, **pop** и **removeFirst**).

В таблице ниже показано два семейства таких методов, которые отличаются по поведению в случае неудачи при выполнении метода.

	Выдает исключение при неудаче	**Возвращает булевскую переменную об исходе операций**
Вставить элемент	`add(e)`	`offer(e)`
Удалить элемент	`remove()`	`poll()`
Проверить список (получить первый элемент)	`element()`	`peek()`

Кроме вышеуказанных методов, в классе есть методы, унаследованные от родителей без изменений. Данные методы присутствуют почти во всех классах на структурах данных.

`Iterator<E> iterator()`	Возвращает итератор для обхода элементов в списке.
`boolean equals(Object object)`	Сравнивает два списка на равенство
`int hashCode()`	Возвращает хэш-код списка. Подробнее описано в теме 4.7. Пока достаточно вам знать, что это число характеризует списки. Равные списки будут иметь одинаковые хэш-коды. Если у двух списков хэш коды равны, то с большой вероятностью, что они равны, но не обязательно. Для знающих. По смыслу это напоминает на хэш-суммы у торрент файлов.
`List<E> subList(int start, int end)`	Возвращает часть списка от номера **start** и до номера **end (**не включая элемент с номером **end)**.
`boolean isEmpty()`	Проверяет пустой ли список или нет.
`String toString()`	Переводит список в строку. Полезно для вывода списка.
`boolean retainAll (Collection<?> collection)`	Удаляет все элементы, которых нет в **collection**.
`boolean removeAll(Collection<?> collection)`	Удаляет все элементы, которые есть в коллекции **collection**.
`boolean containsAll(Collection<?> collection)`	Проверяет наличие всех элементов из указанной коллекции.
`Class<?> getClass()`	Возвращает класс данного класса. Несмотря на кажущуюся

	ненужность, функция очень полезна. В первую очередь полезно для узнавания названия классов для потомков других классов. Особенно полезен метод **toString**().
`void sort(Comparator c)`	Сортирует список согласно какому-то компоратору. **Появился в Java8.**
`boolean removeIf (Predicate<? super E> filter)`	Удаляет элементы из списка по какому-то условию. **Появился в Java8.**
`void forEach(Consumer<? super T> action)`	Позволяет пробежаться по элементам списка без непосредственного использования цикла, а через метод. **Появился в Java8.**
`void replaceAll(UnaryOperator< E> operator)`	Позволяет произвести какие-то операции над всеми элементами списка. Не путать с методом replaceAll из класса Collections. Делают они разные вещи! Тут метод совершает какие-то операции над всеми элементами, а там заменяет одни элементы на другие. **Появился в Java8.**

Есть еще несколько методов для работы со списком в потоках. Примеры по данным методам не предусмотрены в рамках данной темы.

`void notify()`	Позволяет запустить работу объекта в другом потоке, который находился в ожидании из-за вызова метода **wait**.
`void notifyAll()`	Аналогичен предыдущему методу, но позволяет запустить всех ждущих.
`void wait()`	Заставляет вызывающий поток ожидать вызова из другого потока методом **notify**() или **notifyAll**().
`void wait(long millis, int nanos)`	Аналогичен методу **wait**(), но тут можно поставить ограничение по времени: сколько максимально ждать.
`void wait(long millis)`	Аналогичен методу **wait**(), но тут можно поставить ограничение по времени: сколько максимально ждать, но только в миллисекундах.
`Stream<E> stream()`	Возвращает последовательный поток Stream данного списка. **Появился в Java8.**
`Stream<E> parallelStream()`	Возвращает параллельный поток Stream данного списка. **Появился в Java8.**
`Spliterator<E> spliterator()`	Возвращает специальный итератор типа **Spliterator**, который позволяет запускать несколько итераторов параллельно на одном списке. То есть может считывать данные из одного списка параллельно в разных местах. **Появился в Java8.**

Почти все методы присутствуют и в других структурах данных, например, в ArrayList. Так что информация о методах и их функционале пригодится и для использования других классов.

*Обратите еще раз внимание, что методы **forEach**, **parallelStream**, **removeIf**, **replaceAll**, **sort**, **spliterator**, **stream** появились только в Java 8. Например, установщик Android Tools, который ставился в 2014 году во многих классах Samsung IT School содержит Java 7. Поэтому там эти методы работать не будут. Java 8 пока вообще не работает в Android.*

Кроме методов данного класса существуют разнообразные методы других классов, которые могут обрабатывать экземпляры класса **LinkedList** (как частный случай: так можно использовать другие структуры данных). В первую очередь это касается статических функций класса **Collections** (не путать с интерфейсом **Collection**).

Ниже в таблице представлены некоторые полезные функции, которые могут быть вам полезны. Ниже в примерах показаны примеры их использования.

`static <T> void fill(List<? super T> list, T obj)`	Заполняет список какими-нибудь значениями
`static <T> boolean addAll(Collection<? super T> c, T... elements)`	Добавляет в список перечисленные элементы. Элементов может быть сколько угодно.
`static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)`	Поиск элемента в списке бинарным поиском.
`static boolean disjoint(Collection<?> c1, Collection<?> c2)`	Если два списка не содержат одинаковых элементов, то возвращается true.
`static int frequency(Collection<?> c, Object o)`	Сколько раз в списке встречается проверяемый элемент.
`static int indexOfSubList(List<?> source, List<?> target)`	Находит номер первого вхождения подсписка в списке.
`static int lastIndexOfSubList(List<?> source, List<?> target)`	Находит номер последнего вхождения подсписка в списке.
`static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> coll)`	Находит максимальный элемент в списке.

`static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> coll)`	Находит минимальный элемент в списке.
`static <T> boolean replaceAll(List<T> list, T oldVal, T newVal)`	Замещает в списке все элементы одного значения на элементы другого значения.
`static void reverse(List<?> list)`	Разворачивает список.
`static void shuffle(List<?> list)`	Перемешивает массив в случайном порядке.
`static void swap(List<?> list, int i, int j)`	Переставляет местами два элемента списка.
`static <T extends Comparable<? super T>> void sort(List<T> list)`	Сортировка списка.
`static <T> void sort(List<T> list, Comparator<? super T> c)`	Сортировка списка с каким-то компоратором.

Теперь перейдем к рассмотрению примеров использования данного класса и интерфейса **Collections** по работе с ним.

Пример1. Простейшее использование класса LinkedList

В примере создаем список, добавляем два элемента и выводим список.

```
import java.util.LinkedList;

public class MainClass {
	public static void main(String[] args) {

		LinkedList list = new LinkedList();//Создаем список

		list.add("11");//Добавляем новый элемент
		list.add("22");//Добавляем новый элемент

		System.out.println(list);//Выводим список
	}
}
```

На выводе получим:

```
[11, 22]
```

Пример 2. Список с разными по типу элементами и жестко фиксированными

Как и другие структуры данных, списки на основе LinkedList могут быть как без определения типа элементов, так и с жестким заданием типа элементов. В первом случае мы можем в список добавлять произвольные объекты (то есть всё, что разрешит себя добавить и скомпилироваться программе), а во втором мы, соответственно, этой свободы лишены (исключения составляют потомки от того класса, который мы указали).

```
import java.util.LinkedList;

public class MainClass {
     public static void main(String[] args) {
           //Создаем новый список без определенния типа
           LinkedList list = new LinkedList();

           //Добавляем новые элементы
           list.add("11");
           list.add(22);
           list.add(2.56);

           System.out.println("Список:" + list);//Выводим список

           //Создаем новый список определенного типа
           LinkedList <String> list2 = new LinkedList();

           //Добавляем новые элементы
           list2.add("11");
           list2.add("22");
           list2.add("33");

           System.out.println("Список:" + list2);//Выводим список
     }
}
```

На выводе получим:

```
Список:[11, 22, 2.56]
Список:[11, 22, 33]
```

Пример 3. Использование основных методов класса

```
import java.util.LinkedList;

public class MainClass {
     public static void main(String[] args) {
           //Создаем новый список
           LinkedList <String> list = new LinkedList();

           //Добавляем новые элементы
           list.add("11");
           list.add("22");
           list.add("33");
           list.add("44");
           list.add("55");
           list.add("22");
           list.addLast("Z");//Добавим в конец очереди
           list.addFirst("A");//Добавим в начало очереди
           list.add(1, "B");//Добавим вторым элементом

           //Выводим список
           System.out.println("Список:" + list);

           list.remove("22");//Удалим первый элемент равный "22"
           list.remove(2);//Удалим третий элемент
           System.out.println("Список:" + list);//Выводим список

           list.removeFirst();//Удалим первый элемент
           list.removeLast();//Удалим последний элемент
           System.out.println("Список:" + list);//Выводим список

           String val = list.get(2);//Получим третий элемент из списка
           System.out.println("val = " + val);

           list.set(2, "Привет!");//Изменим третий элемент
           System.out.println("Список:" + list);//Выводим список
     }
}
```

На выводе получим:

```
Список:[A, B, 11, 22, 33, 44, 55, 22, Z]
Список:[A, B, 33, 44, 55, 22, Z]
Список:[B, 33, 44, 55, 22]
val = 44
Список:[B, 33, Привет!, 55, 22]
```

Пример 4. Вывод информации о списке

```
import java.util.LinkedList;

public class MainClass {
     public static void main(String[] args) {
           //Создаем новый список
           LinkedList <String> list = new LinkedList();
           list.add("11");
           list.add("22");
           list.add("33");
           list.add("44");
           list.add("22");
           System.out.println("Список:" + list);

           //Получим количество элементов в списке
           int n = list.size();
           System.out.println("Количество элементов в списке: " + n);

           //Проверим пустой ли список
           boolean empty = list.isEmpty();
           System.out.println("Пустой ли список: " + empty);

           //Выведем хэш-код списка
           int hash = list.hashCode();
           System.out.println("Хэш-код списка: " + hash);

           //Проверим, что за класс используется в нашем списке
           String className = list.getClass().toString();
           System.out.println("Класс списка: " + className);

           //Переведем список в строку
           String listString = list.toString();
           System.out.println("Список в виде строки: " + listString);

     }
}
```

На выводе получим:

```
Список:[11, 22, 33, 44, 22]
Количество элементов в списке: 5
Пустой ли список: false
Хэш-код списка: 1525997215
Класс списка: class java.util.LinkedList
Список в виде строки: [11, 22, 33, 44, 22]
```

Пример 5. Поиск элементов в списке

```
import java.util.LinkedList;

public class MainClass {
     public static void main(String[] args) {
           //Создаем новый список
           LinkedList <String> list = new LinkedList();

           //Добавляем новые элементы
           list.add("11");
           list.add("22");
           list.add("33");
           list.add("44");
           list.add("55");
           list.add("22");

           System.out.println("Список:" + list);//Выводим список

           //Проверка наличия элементов
           boolean b;

           b = list.contains("22");
           System.out.println("Содержится ли элемент «22»: " + b);

           b = list.contains("34");
           System.out.println("Содержится ли элемент «34»: " + b);

           //Номера ищущихся элементов
           int m;

           m = list.indexOf("22");
           System.out.println("Номер первого вхождения «22»: " + m);

           m = list.lastIndexOf("22");
           System.out.println("Номер последнего вхождения «22»: " + m);

           m = list.indexOf("34");
           System.out.println("Номер первого вхождения «34»: " + m);
     }
}
```

На выводе получим:

```
Список:[11, 22, 33, 44, 55, 22]
Содержится ли элемент «22»: true
Содержится ли элемент «34»: false
Номер первого вхождения «22»: 1
Номер последнего вхождения «22»: 5
Номер первого вхождения «34»: -1
```

Пример 6. Использование итератора descendingIterator

Метод позволяет получить итератор для обратного обхода элементов.

```
import java.util.Iterator;
import java.util.LinkedList;

public class MainClass {
     public static void main(String[] args) {
            //Создаем новый список
            LinkedList list = new LinkedList();

            //Добавляем новые элементы
            list.add("Вова");
            list.add("Таня");
            list.add("Даша");
            list.add("Вася");

            //Выводим список
            System.out.println("Список:" + list);

            //Получим итератор для обратного порядка
            Iterator x = list.descendingIterator();

            //Выведем в обратном порядке
            while (x.hasNext()) {
                  System.out.println(x.next());
            }
     }
}
```

На выводе получим:

```
Список:[Вова, Таня, Даша, Вася]
Вася
Даша
Таня
Вова
```

Пример 7. Использование метода listIterator

Метод позволяет получить итератор для обхода списка с указанного места.

```
import java.util.Iterator;
import java.util.LinkedList;

public class MainClass {
	public static void main(String[] args) {
		//Создаем новый список
		LinkedList list = new LinkedList();

		//Добавляем новые элементы
		list.add("11");
		list.add("22");
		list.add("33");
		list.add("44");
		list.add("55");

		//Выводим список
		System.out.println("Список:" + list);

		//Выберем итератор со второго элемента
		Iterator x = list.listIterator(1);

		//Выведем в обратном порядке
		while (x.hasNext()) {
			System.out.println(x.next());
		}
	}
}
```

На выводе получим:

```
Список:[11, 22, 33, 44, 55]
22
33
44
55
```

Пример 8. Использование метода iterator

Метод позволяет получить итератор для обхода списка с начала списка.

```
import java.util.Iterator;
import java.util.LinkedList;

public class MainClass {
	public static void main(String[] args) {
		//Создаем новый список
		LinkedList list = new LinkedList();
```

```
            //Добавляем новые элементы
            list.add("11");
            list.add("22");
            list.add("33");
            list.add("44");
            list.add("55");

            //Выводим список
            System.out.println("Список:" + list);

            //Выберем итератор со второго элемента
            Iterator x = list.iterator();

            //Выведем в обратном порядке
            while (x.hasNext()) {
                  System.out.println(x.next());
            }
      }
}
```

На выводе получим:

```
Список:[11, 22, 33, 44, 55]
11
22
33
44
55
```

Пример 9. Получение первого элемента списка разными способами

В примере используются методы, не удаляющие элемент из списка.

```
import java.util.Iterator;
import java.util.LinkedList;

public class MainClass {
      public static void main(String[] args) {
            //Создаем новый список
            LinkedList <String> list = new LinkedList();

            //Добавляем новые элементы
            list.add("11");
            list.add("22");
            list.add("33");
            list.add("44");
            list.add("55");

            //Выводим список
```

```
            System.out.println("Список:" + list);

            //Получим первый элемент списка разными способами
            String x;
            System.out.println("Первый элемент:");

            x = list.getFirst();
            System.out.println("Через getFirst: "+x);

            x = list.get(0);
            System.out.println("Через get(0): "+x);

            x = list.element();
            System.out.println("Через element(): "+x);

            x = list.peek();
            System.out.println("Через peek(): "+x);

            x = list.peekFirst();
            System.out.println("Через peekFirst(): "+x);

            x = list.listIterator(0).next();
            System.out.println("Через listIterator(0).next(): "+x);
      }
}
```

На выводе получим:

```
Список:[11, 22, 33, 44, 55]
Первый элемент:
Через getFirst: 11
Через get(0): 11
Через element(): 11
Через peek(): 11
Через peekFirst(): 11
Через listIterator(0).next(): 11
```

Пример 10. Реализация принципа очереди FIFO (первый пришел - первый ушел)

```
import java.util.LinkedList;

public class MainClass {
      public static void main(String[] args) {
            //Создаем новый список
            LinkedList <String> list = new LinkedList();

            //Добавляем новые элементы
            list.add("11");
            list.add("22");
```

```
            list.add("33");
            list.add("44");
            list.add("55");

            //Выводим список
            System.out.println("Список:" + list);

            //Получим первый элемент списка, удаляя его при этом из очереди
            String x = list.poll();
            System.out.println("x = " + x);

            //Выводим список повторно
            System.out.println("Список:" + list);
      }
}
```

На выводе получим:

```
Список:[11, 22, 33, 44, 55]
x = 11
Список:[22, 33, 44, 55]
```

Пример 11. Перевод списка в массив

```
import java.util.Arrays;
import java.util.LinkedList;

public class MainClass {
      public static void main(String[] args) {
            //Создаем новый список
            LinkedList <String> list = new LinkedList();

            //Добавляем новые элементы
            list.add("11");
            list.add("22");
            list.add("33");
            list.add("44");
            list.add("55");

            //Выводим список
            System.out.println("Список:" + list);

            //Создаем массив из элементов списка
            Object[] x = list.toArray();

            for (int i = 0; i < x.length; i++)
                        System.out.println("x[" + i +"] = "+ x[i]);

            //Если нам нужен массив не просто объектов, а массива нужного
типа
            String[] y = Arrays.asList(x).toArray(new String[x.length]);
```

```
            for (int i = 0; i < y.length; i++)
                        System.out.println("y[" + i +"] = "+ y[i]);

            //Или так
            String[] z = list.toArray(new String[list.size()]);

            for (int i = 0; i < z.length; i++)
                        System.out.println("z[" + i +"] = "+ z[i]);
      }
}
```

На выводе получим:

```
Список:[11, 22, 33, 44, 55]
x[0] = 11
x[1] = 22
x[2] = 33
x[3] = 44
x[4] = 55
y[0] = 11
y[1] = 22
y[2] = 33
y[3] = 44
y[4] = 55
```

Обратите внимание, что приведение массива объектов к массиву определенного типа, производится не совсем простым способом:

```
ТИП_НОВЫЙ[] ПРИВЕДЕННЫЙ_МАССИВ = Arrays.asList(МАССИВ_ОБЪЕКТОВ).toArray(new
ТИП_НОВЫЙ[МАССИВ_ОБЪЕКТОВ.length]);
```

Стандартное же приведение типов скомпилируется, но при выполнении выкинет исключение ClassCastException, т.к. приведение массивов в Java не работает. Так что пользуйтесь таким длинным и не красивым вариантом.

Пример 12. Различия между присваиванием списков и клонированием их

Если мы присвоим один список к другому, то при изменении одного списка будет меняться и другой. При клонировании это не будет происходить.

```
import java.util.LinkedList;
```

```
public class MainClass {
	public static void main(String[] args) {
		//Создаем новый список
		LinkedList list = new LinkedList();
		list.add("11");
		list.add("22");
		list.add("33");

		//Один список просто приравниваем к другому
		LinkedList list2 = list;
		//А второй клонируем
		LinkedList list3=(LinkedList)list.clone();

		list.set(0, "00");//у первого списка поменяем первый элемент

		System.out.println("Список 1:" + list);
		System.out.println("Список 2:" + list2);
		System.out.println("Список 3:" + list3);
	}
}
```

На выводе получим:

```
Список 1:[00, 22, 33]
Список 2:[00, 22, 33]
Список 3:[11, 22, 33]
```

Пример 13. Сортировка списка

Непосредственно в самом классе до Java версии 8 метода сортировки не было. Но он присутствует у класса **Collections**, чем мы и воспользуемся. В первом примере реализована простая сортировка числового списка.

```
import java.util.Collections;
import java.util.LinkedList;

public class MainClass {
	public static void main(String[] args) {

		LinkedList list = new LinkedList();//Создаем список

		list.add(6);
		list.add(2);
		list.add(1);
		list.add(11);
		list.add(8);

		System.out.println("Список:" + list);//Выведем список

		Collections.sort(list);//Сортировка массива
```

```
            System.out.println("Отсортированный  список:"  +  list);//Выведем
список
      }
}
```

На выводе получим:

```
Список:[6, 2, 1, 11, 8]
Отсортированный список:[1, 2, 6, 8, 11]
```

Со строками обычная сортировка будет не совсем корректной, так как будет учитываться регистр и строка «**Ba**» будет идти впереди строки «**aB**». Поэтому сортировку делаем с переопределением метода сравнения строк. Аналогично можно поступать с другими классами.

```
import java.text.Collator;
import java.util.Collections;
import java.util.Comparator;
import java.util.LinkedList;

public class MainClass {
      public static void main(String[] args) {

            LinkedList list = new LinkedList();//Создаем список

            list.add("abc");
            list.add("Bcd");
            list.add("aAb");

            System.out.println("Список:" + list);//Выведем список

            Collections.sort(list);
            System.out.println("Обычный     отсортированный     список:"     +
list);//Выведем список

            //Сортировка массива «правильно», то есть без учета регистра
            Collections.sort(list, new Comparator<String>() {
                  @Override
                  public int compare(String o1, String o2) {
                        return Collator.getInstance().compare(o1, o2);
                  }
            });

            System.out.println("Правильно    отсортированный    список:"    +
list);//Выведем список
      }
}
```

На выводе получим:

```
Список:[abc, Bcd, aAb]
Обычный отсортированный список:[Bcd, aAb, abc]
Правильно отсортированный список:[aAb, abc, Bcd]
```

Если же вы используете Java версии 8 и выше, то у класса List появляется метод **sort**, которым можно отсортировать список (так как LinkedList является потомком). Но при этом обязательно требуется **Comparator**.

```
import java.text.Collator;
import java.util.Comparator;
import java.util.LinkedList;

public class MainClass {
	public static void main(String[] args) {

		LinkedList list = new LinkedList();//Создаем список

		list.add("abc");
		list.add("Bcd");
		list.add("aAb");

		System.out.println("Список:" + list);//Выведем список

		list.sort(new Comparator<String>() {
			@Override
			public int compare(String o1, String o2) {
				return Collator.getInstance().compare(o1, o2);
			}
		});
		System.out.println("Отсортированный список:" + list);//Выведем
список
	}
}
```

На выводе получим:

```
Список:[abc, Bcd, aAb]
Отсортированный список:[aAb, abc, Bcd]
```

Пример 14. Удаление элементов из списка от какого-то элемента до другого элемента

```
import java.util.Iterator;
import java.util.LinkedList;

public class MainClass {
	public static void main(String[] args) {
		//Создаем новый список
```

```
            LinkedList <String> list = new LinkedList();

            //Добавляем новые элементы
            list.add("11");
            list.add("22");
            list.add("33");
            list.add("44");
            list.add("55");

            //Выводим список
            System.out.println("Список:" + list);

            //Удалим элементы со второго по четвертый, не включая его.
            list.subList(1, 3).clear();

            //Выводим список
            System.out.println("Список:" + list);
      }
}
```

На выводе получим:

```
Список:[11, 22, 33, 44, 55]
Список:[11, 44, 55]
```

Пример 15. Взаимодействие нескольких списков

В примере показывается как из списка удалить все элементы, которые есть в некоторой другой коллекции элементов (например, другой список) или, наоборот, отсутствуют в коллекции. Показывается, как вытащить подсписок из списка, как проверить равенства списков? как проверить наличие всех элементов из другой коллекции элементов.

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class MainClass {
      public static void main(String[] args) {
            //Создаем новый список
            LinkedList <String> list = new LinkedList();
            list.add("11");
            list.add("22");
            list.add("33");
            list.add("44");
            list.add("22");
            System.out.println("Список:" + list);

            //Создадим некоторую коллекцию элементов
            ArrayList <String> collection = new ArrayList();
            collection.add("22");
            collection.add("33");
            System.out.println("Коллекция:" + collection);
```

```
            //Продублируем списки во временные переменные
            LinkedList <String> list2=(LinkedList<String>) list.clone();
            LinkedList <String> list3=(LinkedList<String>) list.clone();
            LinkedList <String> list4=(LinkedList<String>) list.clone();

            //Удалим из list2 все элементы, которые есть в коллекции
            list2.removeAll(collection);
            System.out.println("Список 2 после удаления:" + list2);

            //Удалим из list3 все элементы, которых нет в коллекции
            list3.retainAll(collection);
            System.out.println("Список 3 после удаления:" + list3);

            //Выделим подсписок из двух элементов с 1 по 3 (не включая
последний)
            List <String> sublist=list.subList(1, 3);
            System.out.println("Подсписок:" + sublist);

            //Равны ли список list и list2
            boolean b = list.equals(list2);
            System.out.println("b = " + b);

            //Равны ли список list и list4
            b = list.equals(list4);
            System.out.println("b = " + b);

            //Содержит ли список list все элементы из коллекции
            b = list.containsAll(collection);
            System.out.println("b = " + b);
      }
}
```

На выводе получим:

```
Список:[11, 22, 33, 44, 22]
Коллекция:[22, 33]
Список 2 после удаления:[11, 44]
Список 3 после удаления:[22, 33, 22]
Подсписок:[22, 33]
b = false
b = true
b = true
```

Пример 16. Использование метода spliterator

Пример без показа преимущества с параллельным запуском несколько итераторов. Будет работать только в Java 8 и выше.

```
import java.util.LinkedList;
import java.util.Spliterator;

public class MainClass {
	public static void main(String[] args) {
		//Создаем новый список
		LinkedList list = new LinkedList();

		//Добавляем новые элементы
		list.add("11");
		list.add("22");
		list.add("33");
		list.add("44");
		list.add("55");

		//Выводим список
		System.out.println("Список:" + list);

		//Выберем итератор со второго элемента
		 Spliterator<String> spliterator = list.spliterator();

		//Выведем через итератор
		spliterator.forEachRemaining(val->System.out.println(val));
	}
}
```

На выводе получим:

```
Список:[11, 22, 33, 44, 55]
11
22
33
44
55
```

Пример 17. Удаление элементов по условию через метод removeIf

Удаляем все четные элементы из списка. Будет работать только в Java 8 и выше.

```
import java.util.LinkedList;
import java.util.function.Predicate;

public class MainClass {
	public static void main(String[] args) {
		//Создаем новый список
		LinkedList <Integer> list = new LinkedList();

		//Добавляем новые элементы
		list.add(11);
		list.add(22);
```

```
            list.add(33);
            list.add(44);
            list.add(55);

            //Выводим список
            System.out.println("Список:" + list);

            //Удалим элементы по условию, что они четные
            list.removeIf(new Predicate<Integer>() {
                        @Override
                        public boolean test(Integer arg0) {
                                    return (arg0 % 2) == 0;
                        }
            });

            //Выводим список
            System.out.println("Список:" + list);
      }
}
```

На выводе получим:

```
Список:[11, 22, 33, 44, 55]
Список:[11, 33, 55]
```

Пример 18. Использование метода forEach

Работать будет только в Java 8 и выше.

```
import java.util.LinkedList;

public class MainClass {
      public static void main(String[] args) {
            //Создаем новый список
            LinkedList <Integer> list = new LinkedList();

            //Добавляем новые элементы
            list.add(11);
            list.add(22);
            list.add(33);
            list.add(44);
            list.add(55);

            //Выводим список
            System.out.println("Список:" + list);

            //Выведем элементы списка через метод forEach
            list.forEach((val) -> {
                  System.out.println(val);
            });
      }
}
```

На выводе получим:

```
Список:[11, 22, 33, 44, 55]
11
22
33
44
55
```

Пример 19. Использование метода replaceAll

```
import java.util.LinkedList;

public class MainClass {
     public static void main(String[] args) {
           //Создаем новый список
           LinkedList <Integer> list = new LinkedList();

           //Добавляем новые элементы
           list.add(11);
           list.add(22);
           list.add(33);
           list.add(44);
           list.add(55);

           //Выводим список
           System.out.println("Список:" + list);

           //Добавим ко всем элементам 1
           list.replaceAll((x)-> x+1);
           System.out.println("Список:" + list);

           //Произведем более сложные действия
           list.replaceAll((x)-> 2*x+5);
           System.out.println("Список:" + list);
     }
}
```

На выводе получим:

```
Список:[11, 22, 33, 44, 55]
Список:[12, 23, 34, 45, 56]
Список:[29, 51, 73, 95, 117]
```

В следующем примере мы переведем все элементы списка к верхнему регистру.

```
import java.util.LinkedList;

public class MainClass {
	public static void main(String[] args) {
		//Создаем новый список
		LinkedList <String> list = new LinkedList();

		//Добавляем новые элементы
		list.add("Hello");
		list.add("world");

		//Выводим список
		System.out.println("Список:" + list);

		//Добавим ко всем элементам 1
		list.replaceAll((x)-> x.toUpperCase());
		System.out.println("Список:" + list);
	}
}
```

На выводе получим:

```
Список:[Hello, world]
Список:[HELLO, WORLD]
```

Пример 20. Использование статического метода fill из Collections

```
import java.util.Collections;
import java.util.LinkedList;

public class MainClass {
	public static void main(String[] args) {
		//Создаем новый список
		LinkedList <Integer> list = new LinkedList();
		list.add(11);
		list.add(22);
		list.add(33);
		System.out.println("Список:" + list);

		//Заполним его единицами
		Collections.fill(list, 1);

		System.out.println("Список:" + list);

	}
}
```

На выводе получим:

```
Список:[11, 22, 33]
Список:[1, 1, 1]
```

Пример 21. Использование статических методов addAll, binarySearch, disjoint и frequency из Collections

```
import java.util.Collections;
import java.util.LinkedList;

public class MainClass {
     public static void main(String[] args) {
           //Создаем новый список
           LinkedList <Integer> list = new LinkedList();
           list.add(44);
           list.add(22);
           list.add(33);
           System.out.println("Список:" + list);

           //Добавим еще два элемента
           Collections.addAll(list, 77, 55);

           System.out.println("Список:" + list);//Выведем список

           //Найдем номер элемента 33 бинарным поиском
           int index = Collections.binarySearch(list, 33);
           System.out.println("index = " + index);

           //Создадим еще один список
           LinkedList <Integer> list2 = new LinkedList();
           Collections.addAll(list2, 2, 2, 4);
           System.out.println("Список 2:" + list2);//Выведем список

           //Абсолютно ли разные списки или нет?
           boolean b = Collections.disjoint(list,list2);
           System.out.println("b:" + b);

           //А если будет хотя бы один одинаковый элемент?
           Collections.addAll(list2, 22);
           b = Collections.disjoint(list,list2);
           System.out.println("b:" + b);

           //Сколько раз в списке встречается число 2
           int F = Collections.frequency(list2, 2);
           System.out.println("F:" + F);
     }
}
```

На выводе получим:

```
Список:[44, 22, 33]
```

```
Список:[44, 22, 33, 77, 55]
index = 2
Список 2:[2, 2, 4]
b:true
b:false
F:2
```

Пример 22. Нахождение минимального и максимального элемента в списке

```
import java.util.Collections;
import java.util.LinkedList;

public class MainClass {
	public static void main(String[] args) {
		//Создаем новый список
		LinkedList <Integer> list = new LinkedList();
		Collections.addAll(list, 1, -2, 33, 4, 5);

		System.out.println("Список:" + list);//Выведем список

		//Найдем максимальный элемент
		int max  = Collections.max(list);
		System.out.println("max: " + max);

		//Найдем минимальный элемент
		int min  = Collections.min(list);
		System.out.println("min: " + min);
	}
}
```

На выводе получим:

```
Список:[1, -2, 33, 4, 5]
max: 33
min: -2
```

Пример 23. Поиск подсписка в списке

```
import java.util.Collections;
import java.util.LinkedList;

public class MainClass {
	public static void main(String[] args) {
		//Создаем новый список
		LinkedList <Integer> list = new LinkedList();
		Collections.addAll(list, 1, 2, 3, 4, 5);
```

```
            System.out.println("Список:" + list);//Выведем список

            //Создадим еще один список
            LinkedList <Integer> list2 = new LinkedList();
            Collections.addAll(list2, 3, 4);

            int index  = Collections.indexOfSubList(list, list2);
            System.out.println("index:" + index);
      }
}
```

На выводе получим:

```
Список:[1, 2, 3, 4, 5]
index:2
```

Пример 24. Использование статических методов из Collections для управления порядком элементов и замены элементов

```
import java.util.Collections;
import java.util.LinkedList;

public class MainClass {
      public static void main(String[] args) {
            //Создаем новый список
            LinkedList <Integer> list = new LinkedList();
            Collections.addAll(list, 1, -3, 33, -3, 5);

            System.out.println("Список:" + list);//Выведем список

            //Заменим все элементы -3 на 77
            Collections.replaceAll(list, -3, 77);
            System.out.println("Список:" + list);//Выведем список

            //Развернем список
            Collections.reverse(list);
            System.out.println("Список:" + list);//Выведем список

            //Случайно перемешаем список
            Collections.shuffle(list);
            System.out.println("Список:" + list);//Выведем список

            //Переставим местами первый и второй элемент списка
            Collections.swap(list,0,1);
            System.out.println("Список:" + list);//Выведем список

            //Отсортируем массив
            Collections.sort(list);
            System.out.println("Список:" + list);//Выведем список
      }
}
```

На выводе получим:

```
Список:[1, -3, 33, -3, 5]
Список:[1, 77, 33, 77, 5]
Список:[5, 77, 33, 77, 1]
Список:[77, 77, 33, 1, 5]
Список:[77, 77, 33, 1, 5]
Список:[1, 5, 33, 77, 77]
```

Различия между классами LinkedList и ArrayList

С точки зрения функционала, данные классы очень похожи. В чем же разница между ними, и зачем разработчики языка наплодили кучу классов?

К тому же есть странности. Например, в теоретической части говорилось, что очередь реализует принцип «первый пришел - первый ушел». Но почему-то в рассматриваемом классе присутствует метод **pollLast**, когда считывается и удаляется из списка последний пришедший элемент. Также почему-то есть методы, которые позволяют добавлять элементы в середину очереди. Разве это не противоречие тому, что было написано в теоретической части?

Дело в том, что разница с точки зрения «теории» находится во внутренней реализации классов, а вот обычные пользователи не должны видеть разницы в основных методах с точки зрения их использования.

Но это не снимает всех наших вопросов. Ну, захотели программисты по разному реализовать «внутренности» классов, а нам-то зачем всё это? Берем ArrayList и никаких проблем.

Важно понимать, что хотя многие методы выглядят одинаково, но из-за разницы во внутренней реализации, скорость их работы будет сильно различаться. При больших размерах списков и объемах памяти для каждого элемента, разница будет очень существенной. Особенно это заметно при работе с визуальными компонентами для ресурсоограниченных мобильных приложений. Грамотный выбор реализаци списка основывается на том, какие операции будут чаще всего вызываться и какие из них для нас критичнее.

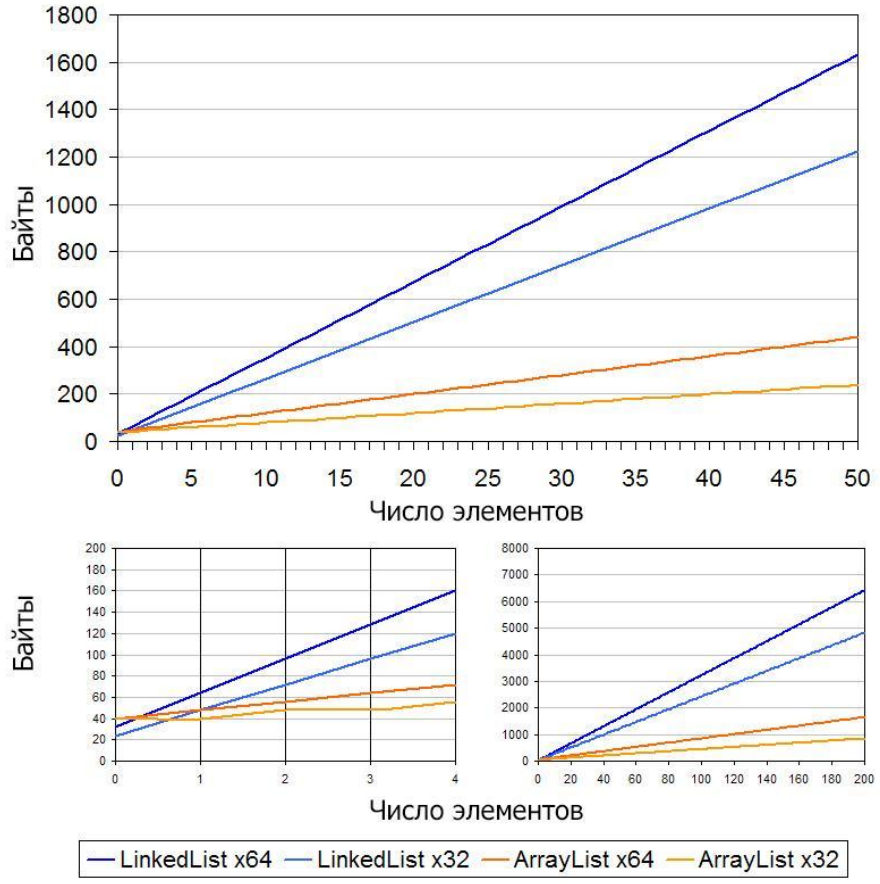
Сложность выполнения операций в ArrayList и LinkedList

	ArrayList	**LinkedList**
get(int index)	O(1)	O(n)
add(E element)	O(n) - в худшем случае	O(1)
add(int index, E element)	O(n-i) (i - номер элемента)	O(n)
remove(int index)	O(n-i) (i - номер элемента)	O(n)
Iterator.remove()	O(n-i) (i - номер элемента)	O(1)
ListIterator.add(E element)	O(n-i) (i - номер элемента)	O(1)

Итак, **ArrayList** хорошо подходит в качестве замены массива, и в данном классе обеспечивается очень быстрый доступ к чтению элементов. Тогда как в **LinkedList** для этого требуется гораздо больше времени.

Но вот добавление элементов в **ArrayList** медленное, а когда число добавляемых элементов превышает количество элементов в списке, то происходит создание нового списка и копирование элементов из старой копии в новую (пользователь этого не видит). Добавление элементов и использование итераторов является ильной стороной **LinkedList**.

Кроме сложности операций данные классы можно сравнить по объему памяти, которое тратится на хранение элементов. На рисунке ниже приведены графики требуемого объема памяти от количества элементов.



Как видим, **LinkedList** занимает гораздо больше места в памяти.

На практике класс **LinkedList** используют не часто и среди программистов бытует мнение, что его выгодно применять только в определенных ситуациях, в первую очередь, где требуется именно двусвязный список.

Класс Stack

Мы рассмотрели класс **LinkedList** и его использование. Но понимание темы списков будет не полным, если не рассказать об очередях и стеках.

Класс Stack реализует рассмотренную нами структуру данных стек. Опишем только основные методы.

Методы класса Stack

`boolean` **`empty`**`()`	Пустой ли список. Обратите внимание, что в LinkedList аналогичный метод назывался isEmpty.
`E` **`peek`**`()`	Возвращает **последний** элемент из списка, но не удаляет.
`E` **`pop`**`()`	Возвращает **последний** элемент из списка и удаляет.
`E` **`push`**`(E object)`	Вставляет элемент в конец списка.
`int` **`search`**`(Object o)`	Возвращает номер первого вхождения в списке искомого элемента. Обратите внимание, что в LinkedList метод аналогичный назывался indexOf.

Если вы попробуйте на практике использовать стек, то увидите, что в стеке есть, например, метод **indexOf**, хотя в таблице написано, что вместо него используется метод **search**. Дело в том, что метод **indexOf** и много других методов наследуются от класса **Vector** (фактически это ArrayList с синхронизацией операций).

Пример 25. Стек

```
import java.util.Stack;

public class MainClass {
	public static void main(String[] args) {
		//Создадим новый список-стек
		Stack<String> stack = new Stack<String>();

		//Заполним список данными
		stack.push("Java");
		stack.push("C#");
		stack.push("Python");

		//Считаем последний элемент списка
		String peek = stack.peek();
		System.out.println("peek : " + peek);

		//Выведем список через цикл
		for (Object lang : stack.toArray())
			System.out.println("lang : " + lang);

		//Выведем список обычным способом
		System.out.println("Список : " + stack);
	}
}
```

На выводе получим:

```
peek : Python
lang : Java
```

```
lang : C#
lang : Python
Список : [Java, C#, Python]
```

Интерфейс Queue

Вторым интерфейсом, который нас интересует является очередь **Queue**. Класс, который от него наследуется, мы уже рассмотрели **LinkedList**.

Как ни странно, интерфейс **Queue** реализует список в виде очереди, которая описывается в теоретической части. При этом во внутренней реализации данного интерфейса реализуется принцип «первый пришел-первый ушел».

Так как данный интерфейс наследуется от **Collection**, то все методы **Collection** также присутствуют в данном интерфейсе. Но появляются специфические методы, которые присущи именно очереди.

Методы интерфейса Queue

`boolean add(E object)`	Добавляет элемент в конец списка. Если не получится, то выдаст исключение.
`E element ()`	Получает, но не удаляет первый элемент из очереди. Если не получится, то выдаст исключение.
`boolean offer(E o)`	Добавляет в конец списка элемент. Если не получится, то возвратит **false**.
`E peek ()`	Получает, но не удаляет первый элемент из очереди. Если не получится, то возвратит **null**.
`E poll ()`	Получает и удаляет первый элемент из списка. Если не получится, то возвратит **null**.
`E remove ()`	Удаляет первый элемент в списке. Если не получится, то выдаст исключение.

Рассматривать дополнительно данные методы не имеет смысла, так как они подробно рассмотрены выше в классе **LinkedList**.

Отметим, что мы можем создавать наши списки, объявляя их как очередь. При этом будут доступны только методы, которые есть у класса **Queue** и его родителей.

Пример 26. Использование очереди

```
Queue<Integer> queue = new LinkedList<Integer>(); // Queue это интерфейс, а LinkedList одна из реализаций
queue.add(16);
queue.add(18);
queue.add(19);
```

Также отметим какие методы идеологически отличают интерфейс **Queue** и класс **Stack**.

В **Queue** метод **poll** возвращает и удаляет **первый** элемент в списке.

В **Stack** метод **pop** возвращает и удаляет **последний** элемент в списке.

В **Queue** метод **offer**, а в **Stack** метод **push** вставляют элемент в **конец** списка.

В **Queue** метод **peek** возвращает первый элемент в списке, а в **Stack** метод с таким же именем возвращает последний элемент списка.

Упражнение 4.2.1

В данном упражнении рассмотрим применение классов LinkedList, Queue и Stack. Во многом данное упражнение повторяет рассмотренные выше примеры использования классов.

Итак, вначале создадим «болванку» обычного java приложения:

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class MainClass {
     public static void main(String[] args) {


     }
}
```

Создадим список типа **LinkedList** с фиксированным типом **String** для элементов.

```
LinkedList<String> list = new LinkedList<String>(); //Создание пустого списка
```

Добавим в список новые элементы:

```
//Добавление элементов в конец списка
list.add(", ");
list.addLast("world");
//Добавление элемента в начало списка
list.addFirst("Hello");
```

Выводим список через цикл (выше использовался способ без использования цикла):

```
for (String s : list) {
     System.out.print(s);
}
System.out.println();
```

Удалим последний элемент и добавим в конец списка другой элемент. Потом выведем полученный список.

```
//Удаление последнего элемент
list.removeLast();
list.addLast("user");

for (String s : list) {
     System.out.print(s);
}
System.out.println();
```

На выводе получим следующее:

```
Hello, world
Hello, user
```

Получим и выведем первый и последний элемент списка:

```
//Получение первого и последнего элемента списка
System.out.println(list.getFirst() + " " + list.getLast());
```

На выводе (без учета предыдущего вывода) получим:

```
Hello user
```

Теперь рассмотрим пример использования стека. Создадим стек, добавим в него элементы:

```
//Пример использования стэка
Stack<Integer> stack = new Stack<Integer>();
stack.add(16);
stack.add(18);
stack.add(19);
```

Теперь вытащим из стека элементы стандартным методом **pop**. При этом вытаскиваться будут элементы в обратном порядке, в котором они добавлялись.

```
System.out.println("===== Stack =====");
System.out.println(stack.pop());
System.out.println(stack.pop());
System.out.println(stack.pop());
```

На выводе получим:

```
===== Stack =====
19
18
16
```

Теперь создадим очередь, и добавим в нее такие же элементы, что и в стеке.

```
//Пример использования очереди
Queue<Integer> queue = new LinkedList<Integer>(); // Queue это интерфейс, а
LinkedList одна из реализаций
queue.add(16);
queue.add(18);
queue.add(19);
```

А из очереди вытаскивать элементы будем методом **poll**.

```
System.out.println("===== Queue =====");
System.out.println(queue.poll());
System.out.println(queue.poll());
System.out.println(queue.poll());
```

На выводе получим:

```
===== Queue =====
16
18
19
```

На данном примере видим разницу в процессе вытаскивания элементов из стека и очереди.

Полный код упражнения:

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class MainClass {
     public static void main(String[] args) {

           LinkedList<String> list = new LinkedList<String>(); //Создание
пустого списка

           //Добавление элементов в конец списка
           list.add(", ");
           list.addLast("world");
           //Добавление элемента в начало списка
           list.addFirst("Hello");

           for (String s : list) {
                 System.out.print(s);
           }
           System.out.println();

           //Удаление последнего элемент
           list.removeLast();
           list.addLast("user");
```

```
            for (String s : list) {
                  System.out.print(s);
            }
            System.out.println();

            //Получение первого и последнего элемента списка
            System.out.println(list.getFirst() + " " + list.getLast());

            //Пример использования стэка
            Stack<Integer> stack = new Stack<Integer>();
            stack.add(16);
            stack.add(18);
            stack.add(19);
            System.out.println("===== Stack =====");
            System.out.println(stack.pop());
            System.out.println(stack.pop());
            System.out.println(stack.pop());

            //Пример использования очереди
            Queue<Integer> queue = new LinkedList<Integer>(); // Queue это
интерфейс, a LinkedList одна из реализаций
            queue.add(16);
            queue.add(18);
            queue.add(19);
            System.out.println("===== Queue =====");
            System.out.println(queue.poll());
            System.out.println(queue.poll());
            System.out.println(queue.poll());
      }
}
```

Полный вывод программы из упражнения:

```
Hello, world
Hello, user
Hello user
===== Stack =====
19
18
16
===== Queue =====
16
18
19
```

Задание 4.2.1

Реализовать интерфейс очереди с ограничением на количество. Очередь должна игнорировать добавление элемента, если ее размер достиг максимума. Требуется реализовать методы **poll** (извлечь первый элемент из очереди), **peek** (считать первый элемент из очереди) и **add** (добавить элемент в конец очереди).

Решите задачи на informatics:

- Задача №54. Простой стек
- *Задача №52. Постфиксная запись

Задание 4.2.2

Создайте класс «Книга» и создайте список книг. В Android приложении предусмотрите функционал для добавления книг, удаления. Реализуйте отображение списка книг.

Задание 4.2.3

Проверьте работу методов класса Collections для строкого списка и для списка со смешанными данными. Обратите внимание на методы **max** и **min**. Добейтесь корректной работы.

Задание 4.2.4

Реализуйте список LinkedList с сортировкой чисел в порядке возрастания суммы их цифр (задача №112319 на informatics).

Дополнительные источники для самостоятельного изучения

1. http://habrahabr.ru/post/127864/ - внутреннее устройство класса **LinkedList**, что происходит при добавлении элементов, как представлен каждый элемент.
2. http://habrahabr.ru/post/237043/ - дополнительная информация по другим структурам данных в Java.
3. http://developer.android.com/reference/java/util/LinkedList.html - документация по классам в Android.
4. http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html - документация по классам в Java8.
5. http://developer.android.com/reference/java/util/Queue.html - документация по интерфейсу очереди.
6. http://developer.android.com/reference/java/util/Stack.html - документация по классу стека.
7. http://developer.android.com/reference/java/util/Collection.html - документация по классу Collection.
8. http://developer.android.com/reference/java/util/Vector.html - документация по классу Vector.

Благодарности

Компания Samsung Electronics выражает благодарность за участие в подготовке данного материала преподавателю IT ШКОЛЫ SAMSUNG Сергиенко Антону Борисовичу.