Прикладное программирование

Лекция №5.

1. Работа с коллекциями в Java

Коллекции как расширение возможностей массивов

В Java получили широкое использование коллекции (**collections**) – **«умные» массивы с динамически изменяемой длиной**, поддерживающие ряд важных дополнительных операций по сравнению с массивами.

Коллекции — это хранилища или контейнеры, поддерживающие различные способы накопления и упорядочения объектов с целью обеспечения возможностей эффективного доступа к ним.

Они представляют собой реализацию абстрактных структур данных, поддерживающих **три основные операции**:

- добавление нового элемента в коллекцию;
- удаление элемента из коллекции;
- изменение элемента в коллекции.

(+ заменить, просмотреть элементы, подсчитать их количество)

Зачем нужны коллекции?

Применение коллекций обусловливается **возросшими объемами обрабатываемой информации**.

Когда счет используемых объектов идет на сотни тысяч, **массивы** не обеспечивают **ни должной скорости**, **ни экономии ресурсов**.

Современные информационные системы тестируются на примере электронного магазина, содержащего около **40 тысяч товаров** и **125 миллионов клиентов**, сделавших **400 миллионов заказов**.

STACK / QUEUE

Наглядным примером коллекции является **стек (Stack)** (структура **LIFO — Last In First Out**), в котором всегда удаляется объект, вставленный последним.

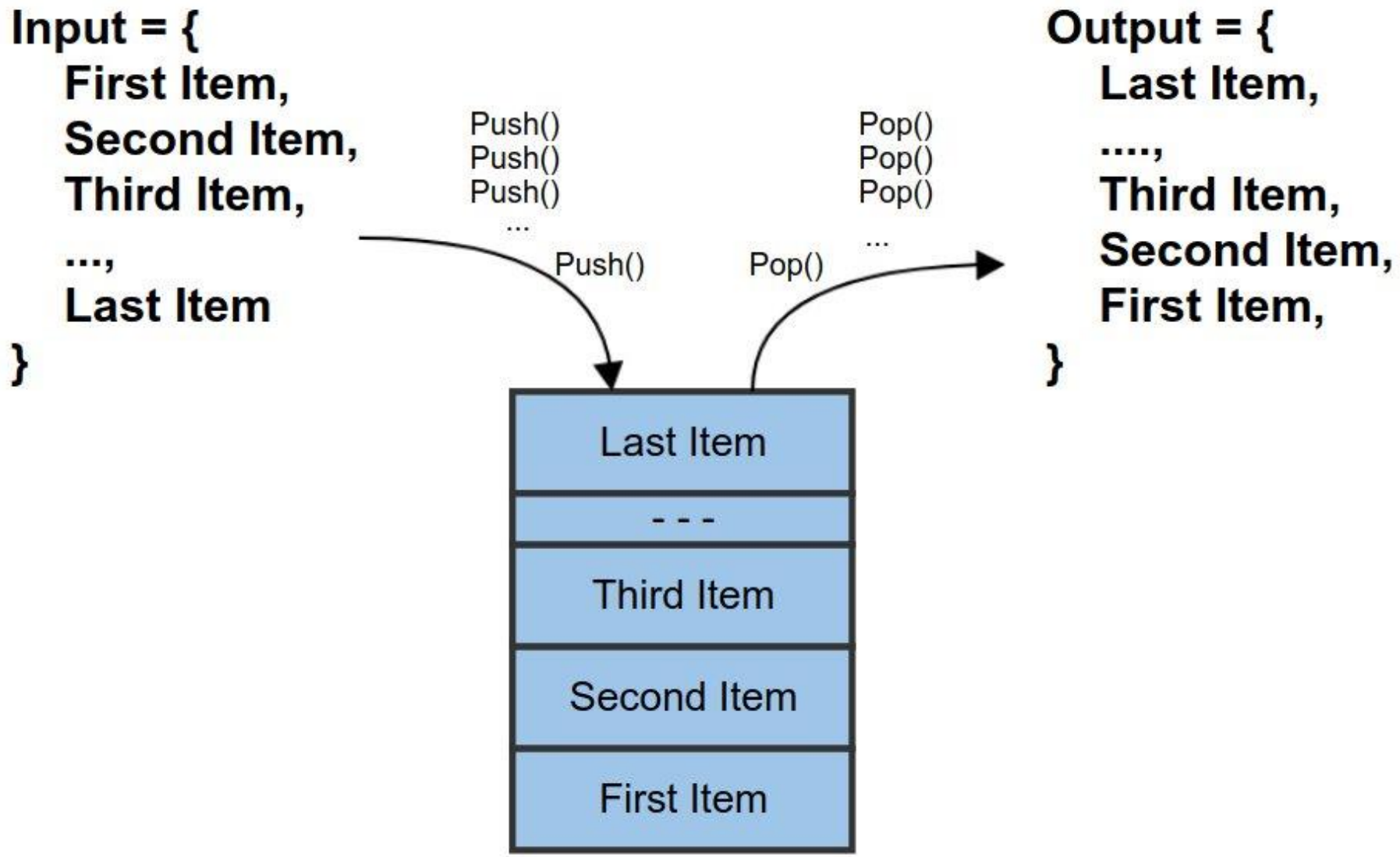
Для **очереди (Queue)** (структура **FIFO — First In First Out**) используется другое правило удаления: всегда удаляется элемент, вставляемый первым.

В абстрактных типах данных существует несколько видов очередей:

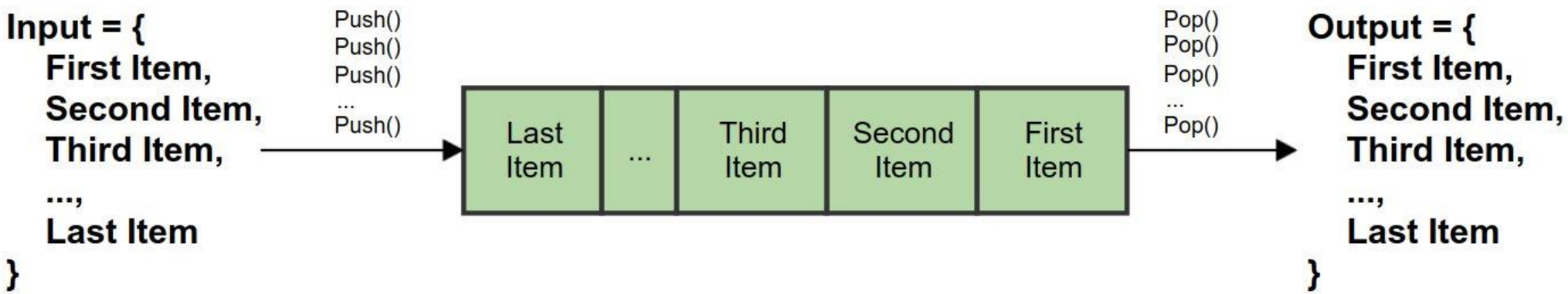
- двусторонние очереди,
- кольцевые очереди,
- обобщенные очереди, в которых запрещены повторяющиеся элементы.

Стеки и очереди могут быть реализованы как на базе массива, так и на базе связного списка.

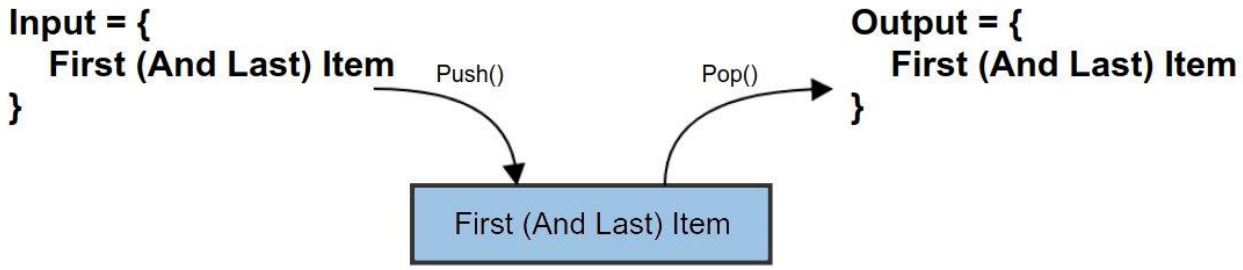
STACK (LIFO)


Input = {
First Item,
Second Item,
Third Item,
...,
Last Item
}
Push()
Push()
Push()
...
Push()
Pop()
Pop()
Pop()
...
Pop()
Output = {
Last Item,
....,
Third Item,
Second Item,
First Item,
}
Last Item
- - -
Third Item
Second Item
First Item

QUEUE (FIFO)

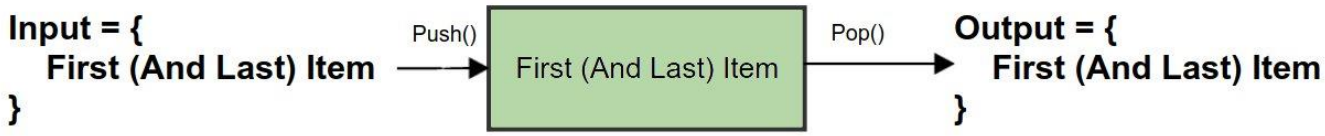

Input = {
First Item,
Second Item,
Third Item,
...,
Last Item
}
Push()
Push()
Push()
...
Push()
Last Item
...
Third Item
Second Item
First Item
Pop()
Pop()
Pop()
...
Pop()
Output = {
First Item,
Second Item,
Third Item,
...,
Last Item
}

Stack (LIFO and FIFO) / Queue (FIFO and LIFO)



Stack (LIFO and FIFO)

Queue (FIFO and LIFO)



Прочие факты о коллекциях

В коллекциях при практическом программировании хранится набор ссылок на объекты одного типа, поэтому следует обезопасить коллекцию от появления ссылок на другие, не разрешенные логикой приложения типы. Такие ошибки при использовании нетипизированных коллекций выявляются на стадии выполнения, что повышает трудозатраты на исправление и верификацию кода.

Начиная с версии Java SE 5, коллекции стали **типизированными**.

Более удобным стал механизм работы с коллекциями, а именно:

- предварительное сообщение компилятору о типе ссылок, которые будут храниться в коллекции, при этом проверка осуществляется на этапе компиляции;
- отсутствие необходимости постоянно преобразовывать возвращаемые по ссылке объекты **(тип Object)** к требуемому типу.

Структура коллекций

Структура коллекций характеризует способ, с помощью которого программы Java обрабатывают группы объектов. Так как **Object** — суперкласс для всех классов, то в коллекции можно хранить **объекты любого типа**, **кроме базовых.**

Коллекции — это динамические **массивы**, **связные списки**, **деревья**, **множества**, **хэш-таблицы**, **стеки**, **очереди**.

Map<K, V>	карта отображения вида «ключ-значение»;
Collection<E>	вершина иерархии коллекций List, Set;
List<E>	специализирует коллекции для обработки списков;
Set<E>	специализирует коллекции для обработки множеств, содержащих уникальные элементы.

Все классы коллекций реализуют также интерфейсы **Serializable**, **Cloneable** (**кроме WeakHashMap)**. Кроме того, классы, реализующие интерфейсы **List<E>** и **Set<E>**, реализуют также интерфейс **Iterable<E>**.

Интерфейс Collection<E>

В интерфейсе Collection<E> определены методы, которые работают на всех коллекциях:

boolean add(E obj)	**добавляет obj** к вызывающей коллекции и возвращает **true**, если объект добавлен, и **false**, если **obj** уже элемент коллекции
boolean remove(Object obj)	**удаляет obj** из коллекции
boolean addAll(Collection<? extends E> c)	**добавляет все элементы** коллекции к вызывающей коллекции
void clear()	**удаляет все элементы** из коллекции;
boolean contains(Object obj)	возвращает **true**, если вызывающая коллекция содержит элемент **obj**

Интерфейс Collection<E>

В интерфейсе Collection<E> определены методы, которые работают на всех коллекциях:

boolean equals(Object obj)	возвращает **true**, если коллекции эквивалентны
boolean isEmpty()	возвращает **true**, если коллекция пуста
Iterator<E> iterator()	извлекает **итератор***
int size()	**возвращает количество элементов** в коллекции
Object[] toArray()	**копирует элементы** коллекции в массив объектов
<T> T[] toArray(T a[])	**копирует элементы** коллекции в массив объектов **определенного типа**

Прочие интерфейсы **коллекций**

При работе с элементами коллекции применяются следующие интерфейсы:

Comparator<T> **Comparable<T>**	для сравнения объектов
Iterator<E> **ListIterator<E>** **Map.Entry<K, V>**	для перечисления и доступа к объектам коллекции

Интерфейс **Map.Entry** предназначен для извлечения ключей и значений карты с помощью методов **K getKey()** и **V getValue()** соответственно.

Вызов метода **V setValue(V value)** заменяет значение, ассоциированное с текущим ключом.

Итератор*

Доступ к элементам коллекции в общем случае не может осуществляться по индексу, так как не все коллекции поддерживают индексацию элементов.

Эту функцию реализуют с помощью специального объекта – **итератора (Iterator)**. У каждой коллекции **collection** имеется **свой итератор,** который умеет с ней работать, поэтому итератор получают следующим образом:

Iterator iter **=** collection**.iterator();**

Итератор позволяет просматривать содержимое коллекции последовательно, элемент за элементом.

Позиции итератора располагаются в коллекции между элементами.

В коллекции, состоящей из N элементов, существует N+1 позиция итератора.

Методы интерфейса Iterator<E>

В интерфейсе Collection<E> определены методы, которые работают на всех коллекциях:

boolean hasNext()	проверяет **наличие следующего элемента**, а в случае его отсутствия (т.е. завершения коллекции) – возвращает **false**. Итератор при этом остается неизменным
Object next()	возвращает **ссылку на объект**, на который указывает итератор, и передвигает текущий указатель на следующий, предоставляя доступ к следующему элементу. Если следующий элемент коллекции отсутствует, то метод **next()** генерирует исключение **NoSuchElementException**
void remove()	удаляет объект, возвращенный последним вызовом метода **next()**. Если метод **next()** до вызова **remove()** не вызывался, то будет сгенерировано исключение **IllegalStateException**

Работа с линейными массивами ArrayList

Класс **ArrayList<E>** — динамический массив объектных ссылок. Реализует интерфейсы **List<E>, Collection<E>, Iterable<E>.**

В классе объявлены конструкторы:

- **ArrayList()**
- **ArrayList(Collection <? extends E> c)**
- **ArrayList(int capacity)**

Удаление и добавление элементов для такой коллекции представляет собой ресурсоемкую задачу, поэтому объект **ArrayList<E>** лучше всего подходит **для хранения списков с малым числом подобных действий**.

С другой стороны, **навигация по списку** осуществляется **очень быстро**, поэтому **операции поиска** производятся **за более короткое время.**

Работа с линейными массивами ArrayList

Методы интерфейса **List<E>** позволяют вставлять и удалять элементы из позиций, указываемых через отсчитываемый от нуля индекс:

void add(int index, E element)	вставляет **element** в позицию, указанную в **index**
Object remove(int index)	удаляет объект из позиции **index**
Object set(int index, E element)	заменяет объект в позиции **index**, возвращает при этом удаляемый элемент
void addAll(int index, Collection<? extends E> c)	вставляет в вызывающий список все элементы коллекции **c**, начиная с позиции **index**
Object get(int index)	возвращает элемент в виде объекта из позиции **index**
int indexOf(Object ob)	возвращает индекс указанного объекта
List<E> subList(int fromIndex, int toIndex)	извлекает часть коллекции в указанных границах

Интерфейс ListIterator

Для доступа к элементам списка может использоваться интерфейс **ListIterator<E>.**

Он позволяет получить доступ сразу в необходимую позицию списка вызовом метода **listIterator(int index).**

Интерфейс **ListIterator<E>** расширяет интерфейс **Iterator<E>** и предназначен для обработки списков и их вариаций.

Object previous(), int previousIndex(), boolean hasPrevious() – обеспечивают обратную навигацию по списку.

- **int nextIndex()** возвращает номер следующего итератора.
- **void add(E obj)** позволяет вставлять элемент в список текущей позиции.
- **void set(E obj)** производит замену текущего элемента списка на объект, передаваемый методу в качестве параметра.

При внесении изменений в коллекцию после извлечения итератора гарантирует генерацию исключения

java.util.ConcurrentModificationException при попытке использовать итератор позднее.

Интерфейс Comparator

При реализации интерфейса **java.util.Comparator<T>** появляется возможность сортировки списка объектов конкретного типа по правилам, определенным для этого типа.

Контракт интерфейса подразумевает реализацию метода **int compare(T ob1, T ob2),** принимающего в качестве параметров **два объекта**, для которых должно быть определено возвращаемое **целое значение**, знак которого и определяет правило сортировки.

Метод **public abstract boolean equals(Object obj)**, также объявленный в интерфейсе **Comparator<T>**, очень рекомендуется переопределять, если экземпляр класса будет использоваться **для хранения информации**.

*Это необходимо для исключения противоречивой ситуации, когда для двух объектов метод **compare()** возвращает **0**, т.о. сообщая об их эквивалентности, в то же время метод **equals()** для этих же объектов возвращает **false**, т.к. данный метод не был никем определен и была использована его версия из класса **Object**.*

Интерфейс Comparator

Кроме того, наличие метода **equals()** обеспечивает корректную работу метода семантического поиска и проверки на идентичность **contains(Object o),** определенного в интерфейсе **java.util.Collection**, а следовательно, реализованного в любой коллекции.

Метод **compare()** автоматически вызывается при сортировке списка методом **static <T> void sort(List<T> list, Comparator<? super T> c)** класса **Collections**, в качестве первого параметра принимающий коллекцию, в качестве второго — **объект-comparator**, из которого извлекается и применяется правило сортировки.

С помощью анонимного типа можно привести простейшую реализацию компаратора.

Если в процессе использования необходимы сортировки по различным полям класса, то реализацию компаратора следует вынести в отдельный класс.

```
public class SortItemRunner {
    public static void main(String[ ] args) {
        ArrayList<Item> p = new ArrayList<Item>() {
            {
                add(new Item(52201, 9.75f, "T-Shirt"));
                add(new Item(52127, 13.99f, "Dress"));
                add(new Item(47063, 45.95f, "Jeans"));
                add(new Item(90428, 60.9f, "Gloves"));
                add(new Item(53295, 31f, "Shirt"));
                add(new Item(63220, 14.9f, "Tie"));
                }
            };
            // создание компаратора
            Comparator<Item> comp = new Comparator<Item>() {
                // сравнение для сортировки по убыванию цены товара
                public int compare(Item one, Item two) {
                    return Double.compare(two.getPrice(), one.getPrice());
                }
                // public boolean equals(Object ob) { /* реализация */ }
            };
            // сортировка списка объектов
            Collections.sort(p, comp);
            System.out.println(p);
        }
    }
}
```

Класс LinkedList и интерфейс Queue

Коллекция **LinkedList<E>** реализует связанный список.

Кроме интерфейсов, указанных при описании **ArrayList**, **LinkedList<E>** также реализует интерфейсы **Queue<E>** и **Deque<E>**.

Связанный список хранит ссылки на объекты отдельно вместе со ссылками на следующее и предыдущее звенья последовательности, поэтому часто называется **двунаправленным списком**.

Операции добавления и удаления выполняются достаточно быстро, в отличие от операций поиска и навигации.

В этом классе объявлены методы, позволяющие манипулировать им как очередью, двунаправленной очередью и т. д.

Двунаправленный список кроме обычного имеет особый **«нисходящий» итератор**, позволяющий двигаться от конца списка к началу, и извлекается методом **descendingIterator().**

Класс LinkedList и интерфейс Queue

Для манипуляций с первым и последним элементами списка в **LinkedList<Object>** реализованы методы:

void addFirst(Object ob)	добавляет элементы в начало списка
void addLast(Object ob)	добавляет элементы в конец списка
Object getFirst(), Object getLast()	извлекают элементы
Object removeFirst(), Object removeLast()	удаляют и извлекают элементы
Object removeLastOccurrence(Object elem), Object removeFirstOccurrence(Object elem)	удаляют и извлекают элемент, первый или последний раз встречаемый в списке

Класс LinkedList и интерфейс Queue

Класс LinkedList<E> реализует интерфейс Queue<E>, что позволяет списку придать свойства очереди. В компьютерных науках очередь — структура данных, в основе которой лежит принцип FIFO (first in, first out). Элементы добавляются в конец и вынимаются из начала очереди.

Но существует возможность не только добавлять и удалять элементы, также можно просмотреть, что находится в очереди. К тому же специализированные методы интерфейса Queue<E> по манипуляции первым и последним элементами такого списка E element(), boolean offer(E o), E peek(), E poll(), E remove() работают немного быстрее, чем соответствующие методы класса LinkedList<E>.

Класс LinkedList и интерфейс Queue

Методы интерфейса Queue<E>:

boolean add(Object o)	вставляет элемент в очередь, если же очередь полностью заполнена, то генерирует исключение **IllegalStateException**
boolean offer(Object o)	вставляет элемент в очередь, если возможно
Object element()	возвращает, но не удаляет головной элемент очереди
Object peek()	возвращает, **но не удаляет** головной элемент очереди, возвращает **null**, если очередь пуста
Object poll()	возвращает **и удаляет** головной элемент очереди, возвращает **null**, если очередь пуста
Object remove()	возвращает и удаляет головной элемент очереди

Методы **element()** и **remove()** отличаются от методов **peek()** и **poll()** тем, что генерируют исключение **NoSuchElementException**, если очередь пуста.

Отличия ArrayList и LinkedList

При всей схожести списков **ArrayList** и **LinkedList** существуют серьезные отличия, которые необходимо учитывать при использовании коллекций в конкретных задачах.

Если необходимо осуществлять **быструю навигацию по списку**, то следует применять **ArrayList**, так как перебор элементов в **LinkedList** осуществляется на порядок медленнее.

С другой стороны, если требуется часто добавлять и удалять элементы из списка, то уже класс **LinkedList** обеспечивает значительно более **высокую скорость переиндексации**.

Если коллекция формируется в начале процесса и в дальнейшем используется только для доступа к информации, то применяется **ArrayList**, если же коллекция подвергается изменениям на всем протяжении функционирования приложения, то выгоднее **LinkedList**.

Интерфейс Deque и класс ArrayDeque

Интерфейс **Deque** определяет **«двунаправленную» очередь** и методы доступа к первому и последнему элементам двусторонней очереди.

Реализацию этого интерфейса можно использовать для моделирования стека.

Методы обеспечивают **удаление**, **вставку** и **обработку элементов**. Каждый из этих методов существует в двух формах.

Одни методы создают исключительную ситуацию в случае неудачного завершения, другие возвращают **null** или **false** (в зависимости от типа операции).

Вторая форма добавления элементов в очередь сделана специально для реализаций **Deque**, имеющих ограничение по размеру.

Методы **addFirst(), addLast()** (аналогичен методу **add()**) вставляют элементы в начало и в конец очереди соответственно.

Объявить двуконечную очередь на основе связанного списка можно:

Deque<String> dq = new **LinkedList<>**();

!!! Класс **ArrayDeque** быстрее, чем **Stack**, если **используется как стек**, и быстрее, чем **LinkedList<E>**, если используется **в качестве очереди**.

Множества

Интерфейс **Set<E>** объявляет поведение коллекции, не допускающей дублирования элементов.

Интерфейс **SortedSet<E>** наследует **Set<E>** и объявляет поведение набора, отсортированного в возрастающем порядке, заранее определенном для класса.

Интерфейс **NavigableSet<E>** существенно облегчает поиск элементов, расположенных рядом с заданным.

Класс **HashSet<E>** наследуется от абстрактного суперкласса **AbstractSet<E>** и реализует интерфейс **Set<E>**, используя **хэш-таблицу** для хранения коллекции.

Ключ (хэш-код) используется в качестве **индекса хэш-таблицы** для доступа к объектам множества, что значительно ускоряет процессы поиска, добавления и извлечения элемента.

Множества

Скорость указанных процессов становится заметной для коллекций с большим количеством элементов.

Множество **HashSet** не является сортированным. В таком множестве могут храниться элементы с одинаковыми хэш-кодами в случае, если эти элементы не эквивалентны при сравнении.

Для грамотной организации **HashSet** следует следить, чтобы реализации методов **hashCode()** и **equals()** соответствовали контракту.

Конструкторы класса:

- **HashSet() HashSet(Collection <? extends E> c)**
- **HashSet(int capacity)**
- **HashSet(int capacity, float loadFactor),**

где **capacity** — число ячеек для хранения **хэш-кодов**.

Множества

Класс **TreeSet<E>** для хранения объектов использует бинарное дерево.

При добавлении объекта в дерево он сразу же размещается в необходимую позицию с учетом сортировки.

Сортировка происходит благодаря тому, что класс реализует интерфейс **SortedSet**, где правило сортировки добавляемых элементов определяется в самом классе, сохраняемом в множестве, который в большинстве случаев реализует интерфейс **Comparable**.

Обработка операций удаления и вставки объектов происходит медленнее, чем в хэш-множествах, но быстрее, чем в списках.

Конструкторы класса:

- **TreeSet()**
- **TreeSet(Collection <? extends E> c)**
- **TreeSet(Comparator <? super E> c)**
- **TreeSet(SortedSet <E> s)**

Класс **TreeSet<E>** содержит методы по извлечению первого и последнего (наименьшего и наибольшего) элементов **E first()** и **E last().**

Методы **subSet(E from, E to), tailSet(E from)** **и** **headSet(E to)** предназначены для извлечения определенной части множества.

Метод **Comparator <? super E> comparator()** возвращает объект **Comparator**, используемый для сортировки объектов множества или **null**, если выполняется обычная сортировка.

Множества

Множество **инициализируется списком** и **сортируется сразу же в процессе создания**.

С помощью итератора элемент может быть **найден** и **удален** из множества.

Для множества, состоящего из обычных строк, используется **лексикографическая сортировка**, задаваемая реализацией интерфейса **Comparable**, поэтому метод **comparator()** возвращает **null**.

Абстрактный класс **EnumSet<E extends Enum<E>>** наследуется от абстрактного класса **AbstractSet.**

Класс специально реализован для работы с типами **enum**.

Все элементы такой коллекции должны принадлежать единственному типу **enum**, определенному явно или неявно.

Скорость выполнения операций над таким множеством очень высока, даже если в ней участвует большое количество элементов.

Карты отображений

Карта отображений — это объект, который хранит пару **«ключ–значение».** Поиск объекта (значения) облегчается по сравнению с множествами за счет того, что его можно найти по его уникальному ключу. Уникальность объектов ключей должна обеспечиваться переопределением методов **hashCode()** и **equals()** или реализацией интерфейсов **Comparable**, **Comparator** пользовательским классом.

Классы карт отображений:

AbstractMap<K, V>	реализует интерфейс **Map<K, V>,** является суперклассом для всех перечисленных карт отображений
HashMap<K, V>	использует хэш-таблицу для работы с ключами
TreeMap<K, V>	использует дерево, где ключи расположены в виде дерева поиска в определенном порядке
WeakHashMap<K, V>	позволяет механизму сборки мусора удалять из карты значения по ключу, ссылка на который вышла из области видимости приложения
LinkedHashMap<K, V>	образует дважды связанный список ключей. Этот механизм эффективен, только если превышен коэффициент загруженности карты при работе с кэш-памятью и др.

Карты отображений

Для класса **IdentityHashMap<K, V>** хэш-коды объектов-ключей вычисляются методом **System.identityHashCode()** по адресу объекта в памяти в отличие от обычного значения **hashCode(),** вычисляемого сугубо по содержимому самого объекта.

Интерфейсы карт:

Map<K, V>	отображает уникальные ключи и значения
Map<K, V>	описывает пару «ключ–значение»
SortedMap<K, V>	содержит отсортированные ключи и значения
NavigableMap<K, V>	добавляет новые возможности навигации и поиска по ключу

Карты отображений

Интерфейс **Map<K, V>** содержит следующие методы:

V get(Object obj)	возвращает значение, связанное с ключом **obj**. Если элемент с указанным ключом отсутствует в карте, то возвращается значение **null**
V put(K key, V value)	помещает ключ key и значение value в вызывающую карту. При добавлении в карту элемента с существующим ключом произойдет замена текущего элемента новым. При этом метод возвратит заменяемый элемент
void putAll(Map <? extends K,? extends V> t)	помещает коллекцию **t** в вызывающую карту
V remove(Object key)	удаляет пару **«ключ–значение»** по ключу **key**
void clear()	удаляет все пары из вызываемой карты

Карты отображений

Интерфейс **Map<K, V>** содержит следующие методы:

boolean containsKey(Object key)	возвращает **true**, если вызывающая карта содержит **key** как ключ
boolean containsValue(Object value)	возвращает **true**, если вызывающая карта содержит **value** как значение
Set<K> keySet()	возвращает множество ключей
Set<Map.Entry<K, V>> entrySet()	возвращает множество, содержащее значения карты в виде пар **«ключ–значение»**
Collection<V> values()	возвращает коллекцию, содержащую значения карты

В коллекциях, возвращаемых тремя последними методами, **можно только удалять элементы**, **добавлять нельзя.**

Данное ограничение обуславливается **параметризацией возвращаемого методами значения.**

Карты отображений

Интерфейс **Map.Entry<K, V>** представляет пару **«ключ–значение»** и содержит следующие методы:

K getKey()	возвращает ключ текущего входа
V getValue()	возвращает значение текущего входа
V setValue(V obj)	устанавливает значение объекта **obj** в текущем входе

Класс **EnumMap<K extends Enum<K>, V>** в качестве ключа может принимать только объекты, принадлежащие одному типу **enum**, который должен быть определен при создании коллекции.

Специально организован для обеспечения максимальной скорости доступа к элементам коллекции.

Унаследованные коллекции

В ряде распределенных приложений, например с использованием ***сервлетов***, до сих пор применяются коллекции, более медленные в обработке, но при этом **потокобезопасные** (**thread-safety**), существовавшие в языке Java с момента его создания,

а именно карта **Hashtable<K, V>**, **список Vector<E>** и перечисление (аналог итератора) **Enumeration<E>**.

Все они также были параметризованы, но сохраняют возможность одновременного доступа из конкурирующих потоков.

Класс **Hashtable<K, V>** реализует интерфейс **Map**, а также обладает несколькими специфичными по сравнению с другими коллекциями методами:

Enumeration<V> elements() — возвращает перечисление для значений карты;

Enumeration<K> keys() — возвращает перечисление для ключей карты.

Алгоритмы класса Collections

Класс **java.util.Collections** содержит большое количество статических методов, предназначенных для манипулирования коллекциями.

С применением предыдущих версий языка было разработано множество коллекций, в которых нет проверок, поэтому нельзя гарантировать, что в коллекцию не будет помещен «посторонний» объект.

Для этого в класс **Collections** был добавлен новый метод:

static <E> Collection <E> checkedCollection(Collection<E> c, Class<E> type)

Этот метод создает коллекцию, проверяемую на этапе выполнения, т. е. в случае добавления «постороннего» объекта генерируется исключение **ClassCastException**.

В этот же класс добавлен целый ряд статических методов, специализированных для проверки конкретных типов коллекций, а именно:

- **checkedList(),**
- **checkedSortedMap(),**
- **checkedMap(),**
- **checkedSortedSet(),**
- **checkedCollection().**

Алгоритмы класса Collections

<T> void copy(List<? super T> dest, List<? extends T> src)	копирует все элементы из одного списка в другой
boolean disjoint(Collection<?> c1, Collection<?> c2)	возвращает **true**, если коллекции не содержат одинаковых элементов
<T> List <T> emptyList() **<K, V> Map <K, V> emptyMap()** **<T> Set <T> emptySet()**	возвращают пустой список, карту отображения и множество соответственно
<T> void fill(List<? super T> list, T obj)	заполняет список заданным элементом
int frequency(Collection<?> c, Object o)	возвращает количество вхождений в коллекцию заданного элемента
<T> boolean replaceAll(List<T> list, T oldVal, T newVal)	заменяет все заданные элементы новыми
void reverse(List<?> list)	«переворачивает» список
void rotate(List<?> list, int distance)	сдвигает список циклически на заданное число элементов
void shuffle(List<?> list)	перетасовывает элементы списка

Алгоритмы класса Collections

singleton(T o), singletonList(T o), singletonMap(K key, V value)	создают множество, список и карту отображения, позволяющие добавлять только один элемент
<T extends Comparable<? super T>> void sort(List<T> list), <T> void sort(List<T> list, Comparator<? super T> c)	сортировка списка естественным порядком и с использованием **Comparable** или **Comparator** соответственно
void swap(List<?> list, int i, int j)	меняет местами элементы списка, стоящие на заданных позициях
<T> List<T> unmodifiableList(List<? extends T> list)	возвращает ссылку на список с запрещением его модификации. Аналогичные методы есть и для других видов коллекций

На сегодня все!
Увидимся на следующей неделе!