Восьмая лекция

Коллекции в Java

Мощный и полезный механизм хранения объектов в Java. Простейшей коллекцией является массив. Но массив имеет ряд недостатков. Один из самых существенных это размер массива который фиксируется до начала его использования вторым существенным недостатком является то , что элементы массива имеют жестко заданное размещение в его ячейках, поэтому, например, удаление элемента из массива не является простой операцией.

Коллекции обладают одним важным свойством — их размер не ограничен а также в коллекциях эффективно реализованы операциями добавления, извлечения и поиска элементов.

Чтобы выбрать коллекцию, которая лучше всего подходит условию задачи, необходимо знать особенности каждой из них.

Основные принципы хранения данных

Список (**List**) - представляет собой упорядоченный набор элементов и может содержать повторяющиеся элементы. Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу. Список представляет собой динамический массив. Список является одним из наиболее используемых типов коллекций.

Множества - в множествах **Set** каждый элемент хранится только в одном экземпляре, а разные реализации Set используют разный порядок хранения элементов.

Queue — коллекция, предназначенная для хранения элементов в порядке, нужном для их обработки. В дополнение к базовым операциям интерфейса Collection, очередь предоставляет дополнительные операции вставки, получения и контроля.

Очереди обычно, но не обязательно, упорядочивают элементы в FIFO (first-in-first-out, «первым вошел — первым вышел») порядке.

Ассоциативные массивы **Map**. В Map'е хранится ключ и связанное с ним значение. Ключом, может быть любой объект или строка.

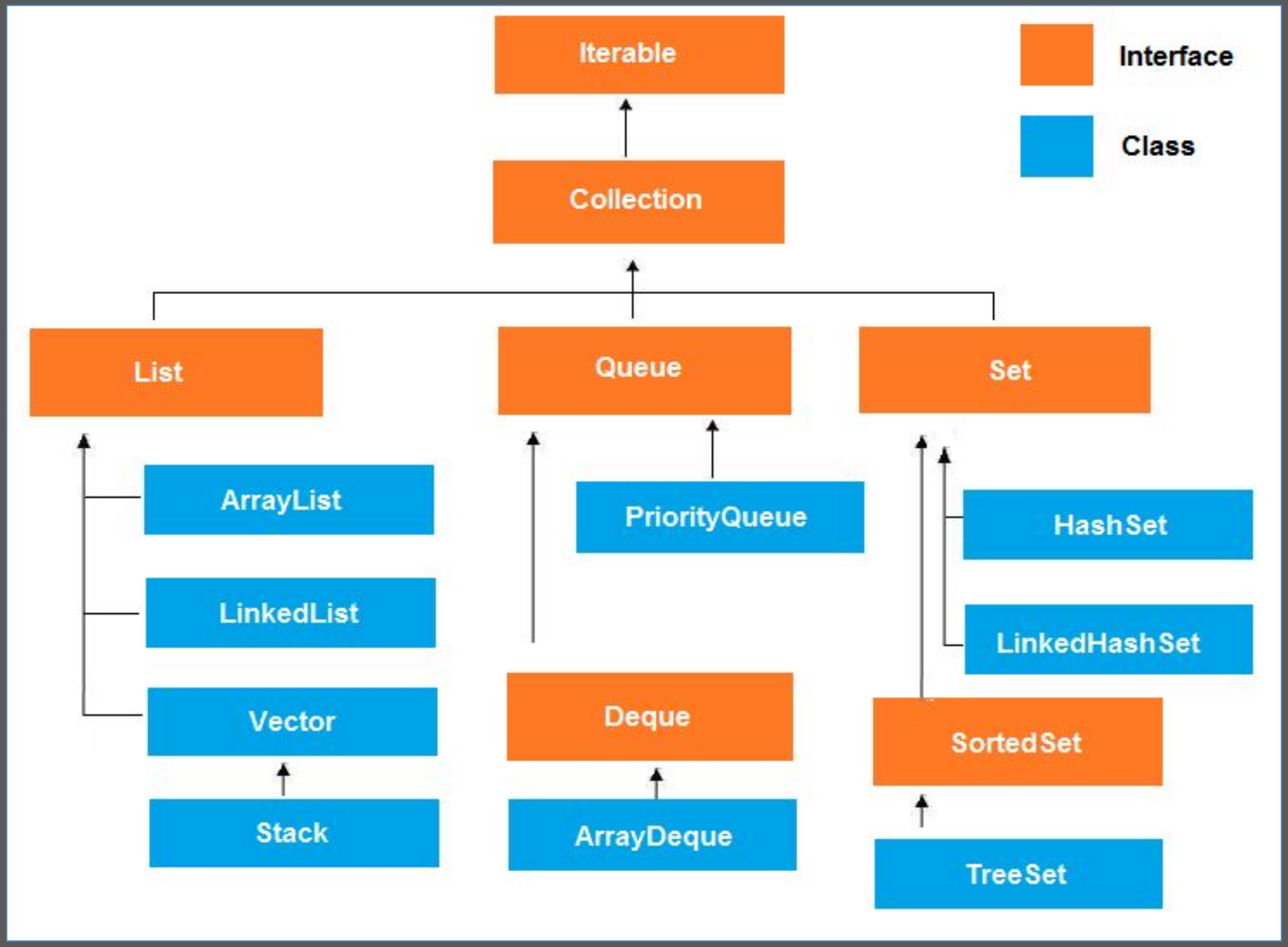
Java Collections Framework

Язык Java предоставляет нам стандартную библиотеку коллекций, которые находятся в пакете java.util.

Collection и **Map** являются базовыми интерфейсами. Их расширяют другие интерфейсы, добавляющие дополнительный функционал.

Каждый класс реализует некоторую коллекцию со специфичным для нее набором операций доступа к элементам. Чтобы использовать коллекцию в своей программе, нужно создать объект соответствующего класса.

На вершине иерархии в Java Collection Framework располагаются 2 интерфейса: Collection и Map. Эти интерфейсы разделяют все коллекции, входящие во фреймворк на две части по типу хранения данных: простые последовательные наборы элементов и наборы пар «ключ — значение».

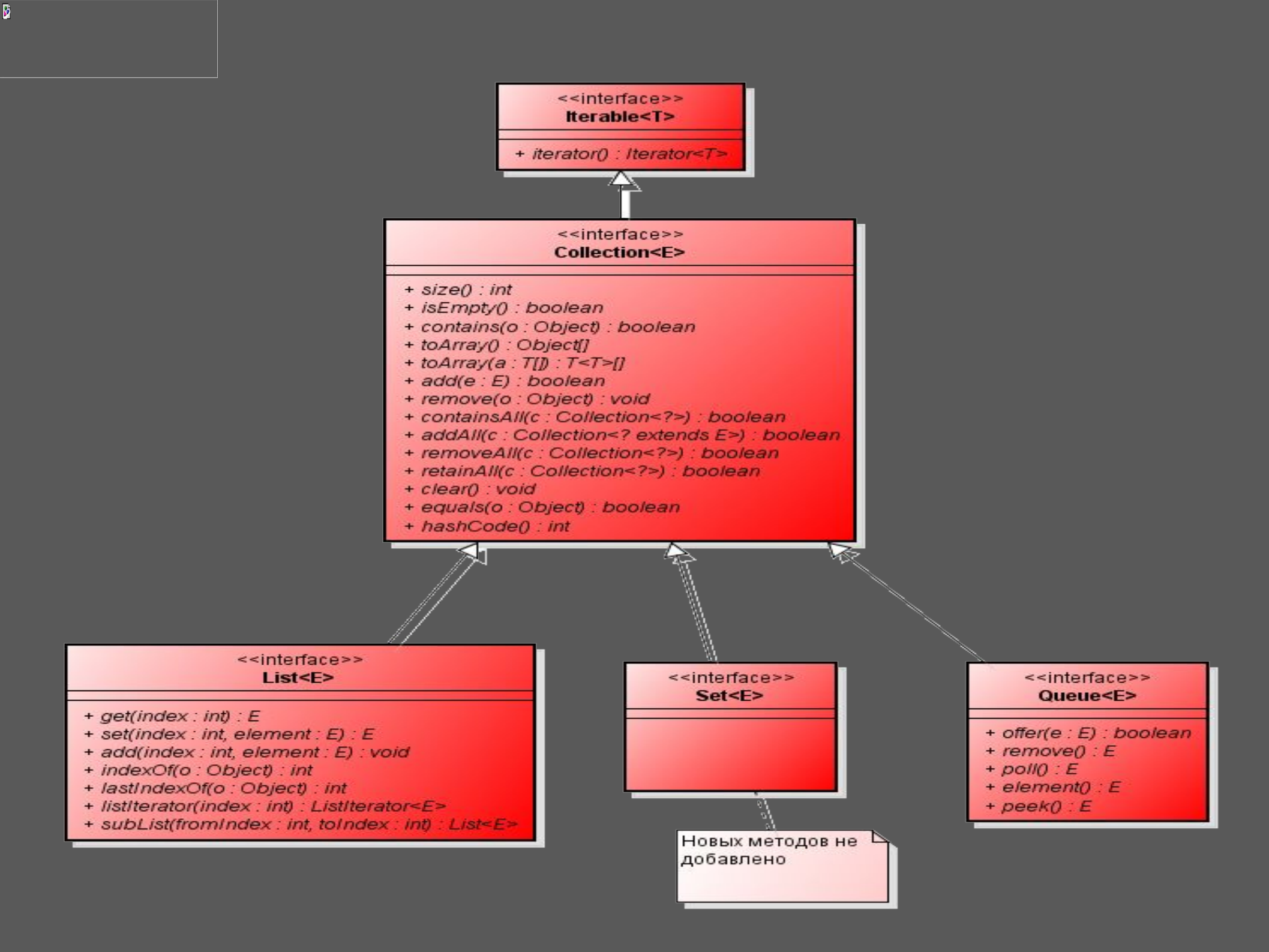

Iterable
Interface
Class
Collection
List
Queue
Set
ArrayList
PriorityQueue
HashSet
LinkedList
LinkedHashSet
Vector
Deque
SortedSet
Stack
ArrayDeque
TreeSet

Интерфейс Collection

Интерфейс Collection расширяется за счет двух основных интерфейсов.

List — пронумерованный список, содержит все методы интерфейса Collection, а также свои. Как и в обычном массиве все элементы списка пронумерованы начиная с нуля и к любому элементу можно обратиться по индексу:

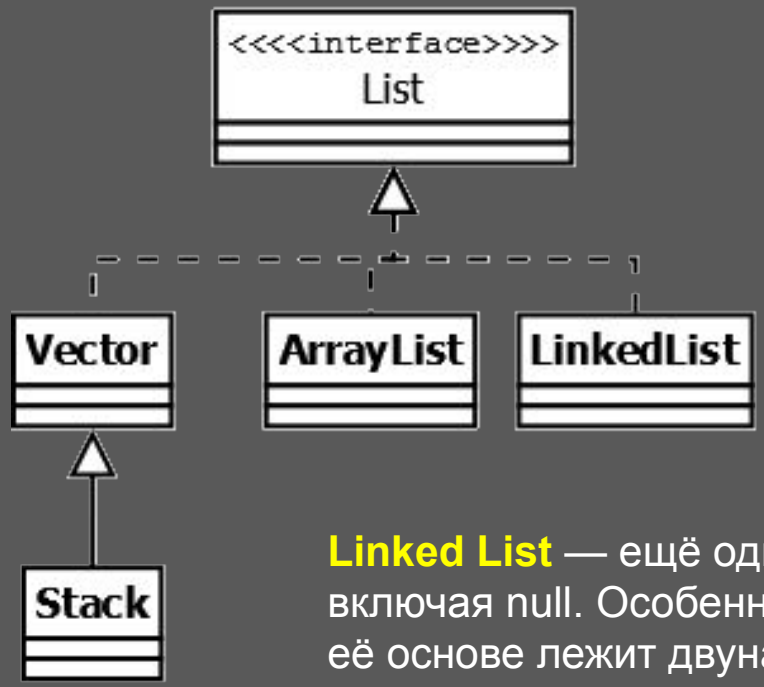
Set — множество. Содержит все операции интерфейса Collection, но некоторые из них имеют другой смысл. например операция add добавляет только уникальные элементы, зато операции поиска элемента происходят быстрее чем в списке.


<<interface>>
Iterable<T>
+ iterator() : Iterator<T>
<<interface>>
Collection<E>
+ size() : int
+ isEmpty() : boolean
+ contains(o : Object) : boolean
+ toArray() : Object[]
+ toArray(a : T[]) : T<T>[]
+ add(e : E) : boolean
+ remove(o : Object) : void
+ containsAll(c : Collection<?>) : boolean
+ addAll(c : Collection<? extends E>) : boolean
+ removeAll(c : Collection<?>) : boolean
+ retainAll(c : Collection<?>) : boolean
+ clear() : void
+ equals(o : Object) : boolean
+ hashCode() : int
<<interface>>
List<E>
+ get(index : int) : E
+ set(index : int, element : E) : E
+ add(index : int, element : E) : void
+ indexOf(o : Object) : int
+ lastIndexOf(o : Object) : int
+ listIterator(index : int) : ListIterator<E>
+ subList(fromIndex : int, toIndex : int) : List<E>
<<interface>>
Set<E>
Новых методов не добавлено
<<interface>>
Queue<E>
+ offer(e : E) : boolean
+ remove() : E
+ poll() : E
+ element() : E
+ peek() : E

Collection — самый общий интерфейс:
add(T e) — добавить элемент e
clear() — очистить
addAll(Collection col) — добавить все элементы другой коллекции, с схожим типом данных
contains(Object o) — содержит ли коллекция элемент
isEmpty() — возвращает пуста ли коллекция
remove(Object o) — удаляет элемент
removeAll(Collection col) — удаляет все элементы, которые есть в коллекции col.
size() — возвращает количество элементов в коллекции
containsAll(Collection col) — возвращает, содержатся ли все элементы col в коллекции.
toArray(T[] a) — возвращает массив, который содержит все элементы коллекции, на вход принимает массив, который будет заполнен ими.
retainAll(Collection col) — удаляет все элементы, не принадлежащие col
toArray() — возвращает массив Object-ов, который содержит все элементы коллекции.

List
get(int index) — получаем элемент по индексу
add(int index, T e) — вставляет элемент в позицию
indexOf(Object obj) — возвращает первое вхождение элемента в список
lastIndexOf(Object obj) — возвращает последнее вхождение
set(int index, T e) — заменяет элемент в позиции index
subList(int from, int to) — возвращает новый список, представляющий собой часть главного

Интерфейс List

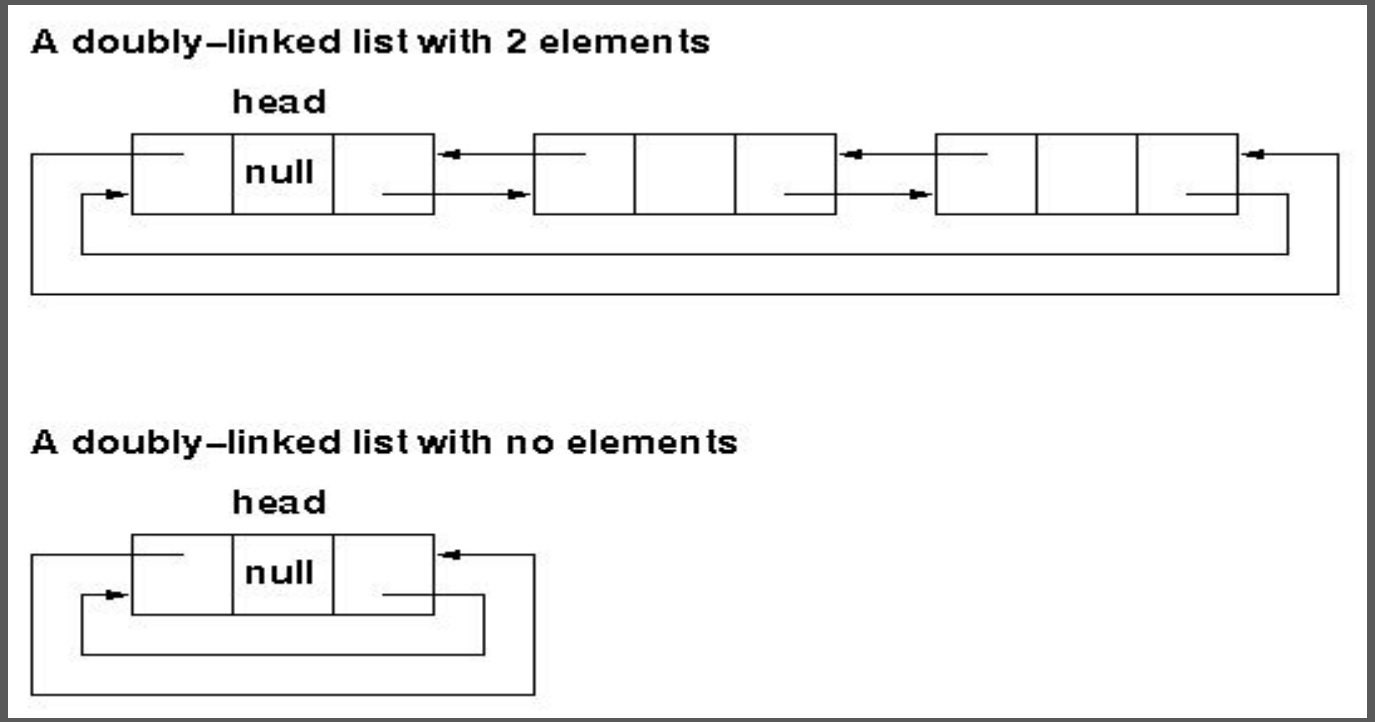


ArrayList — Позволяет хранить любые данные, включая null в качестве элемента. Как можно догадаться из названия, его реализация основана на обычном массиве. Поэтому при вставке элемента в середину, приходится сначала сдвигать на один все элементы после него, а уже затем в освободившееся место вставлять новый элемент. Зато в нем быстро реализованы взятие и изменение элемента – операции get, set, так как в них мы просто обращаемся к соответствующему элементу массива.

Linked List — ещё одина реализация List. Позволяет хранить любые данные, включая null. Особенностью реализации данной коллекции является то, что в её основе лежит двунаправленный связный список (каждый элемент имеет ссылку на предыдущий и следующий). Поиск элемента в LinkedList начинается в цикле от края списка. Если искомый элемент находится в первой половине списка, то поиск идёт с начала, в обратном случае — с конца.

Vector — реализация динамического массива объектов. Позволяет хранить любые данные, включая null в качестве элемента. Vector появился в JDK версии Java 1.0. В качестве альтернативы применяется аналог — ArrayList, оставлен для совместимости с предыдущими версиями.

Stack — данная коллекция является расширением коллекции Vector. Была добавлена в Java 1.0 как реализация стека LIFO (last-in-first-out). После добавления в Java 1.6 интерфейса Deque, рекомендуется использовать именно реализации этого интерфейса, например ArrayDeque.



Если необходимо вставлять (или удалять) в середину коллекции много элементов, то лучше использовать LinkedList. Во всех остальных случаях – ArrayList.

LinkedList требует больше памяти для хранения такого же количества элементов, потому что кроме самого элемента хранятся еще указатели на следующий и предыдущий элементы списка, тогда как в ArrayList элементы просто идут по порядку.

Класс ArrayList

ArrayList — реализует интерфейс List. Как известно, в Java массивы имеют фиксированную длину, и после того как массив создан, он не может расти или уменьшаться. ArrayList может менять свой размер во время исполнения программы, при этом не обязательно указывать размерность при создании объекта. Элементы ArrayList могут быть абсолютно любых типов в том числе и null.

По умолчанию при создании нового объекта **ArrayList** создается внутренний массив длиной 10 элементов, но так же можно также создать **ArrayList**, задав его начальную длину

```
List ar = new ArrayList(100);
```

Если при добавлении элемента, оказывается, что массив полностью заполнен, будет создан новый массив размером (n * 3) / 2 + 1, в него будут помещены все элементы из старого массива + новый, добавляемый элемент.

Работать с ArrayList просто: создайте нужный объект, вставляйте созданные объекты методом **add()**, обращайтесь к ним методом **get()**, используйте индексирование так же, как для массивов, но без квадратных скобок. ArrayList также содержит метод **size()**, который возвращает текущее количество элементов (напомню, что в обычном массиве используется свойство lenght). Удалять элементы можно двумя способами: по индексу **remove(index)** и по значению **remove(value).** Удален будет лишь первый найденный элемент.

Пример:

```
ArrayList<String> catNamesList = new ArrayList<String>();

catNamesList.add("Васька");

String name = catNamesList.get(0);
```

Класс LinkedList

Существует также класс **LinkedList** имеющий больше операций, чем ArrayList , а значит более сложный и требующий больше памяти.

LinkedList — реализует интерфейс List. Является представителем двунаправленного списка, где каждый элемент структуры содержит указатели на предыдущий и следующий элементы.

Реализует методы получения, удаления и вставки в начало, середину и конец списка. Позволяет добавлять любые элементы в том числе и null.

LinkedList<String> list = new LinkedList<String>(); или
List<String> list = new LinkedList<String>(); используя интерфейс List

Добавление элемента в конец списка с помощью методом add(value);

Для того чтобы добавить элемент на определенную позицию в списке, необходимо вызвать метод **add(index, value).**

Удалять элементы из списка по индексу **remove(index);**

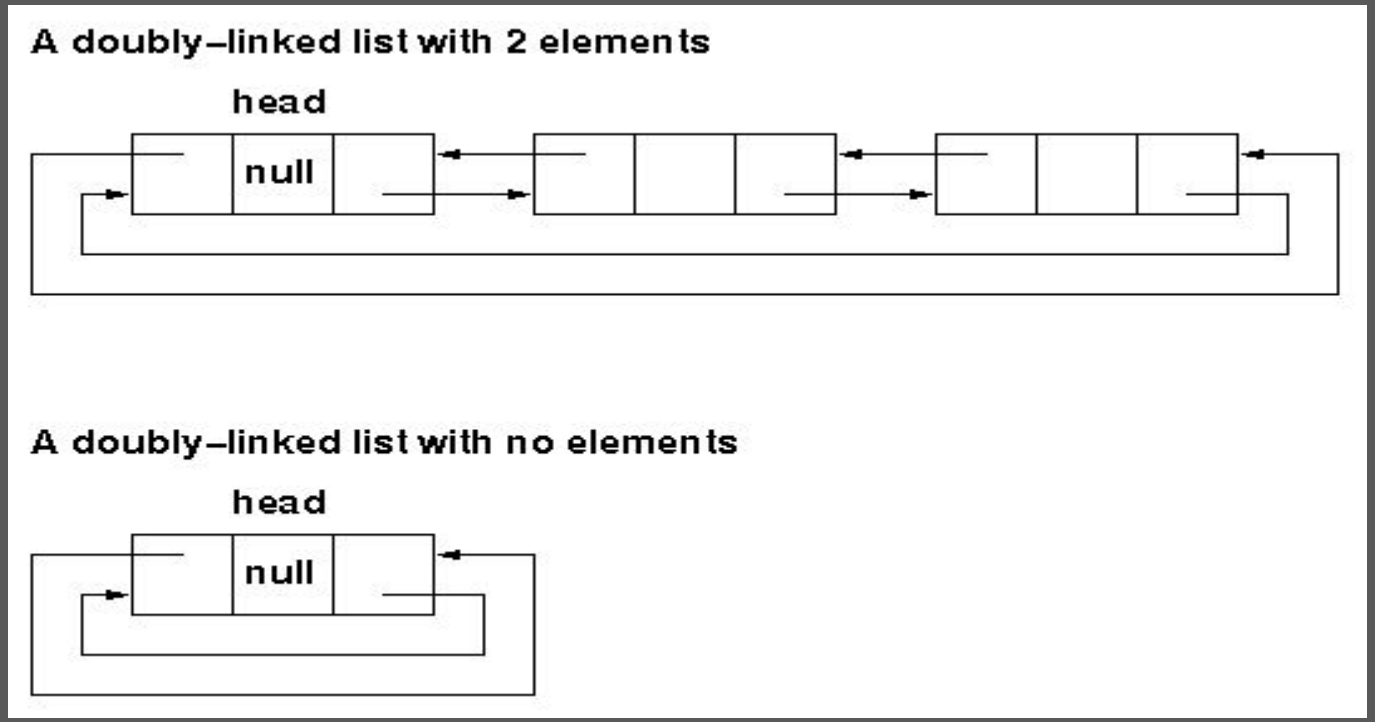
Для очистки массива используется метод **clear();**

У **LinkedList** представлен ряд методов, не входящих в интерфейс **List**:

addFirst() и **addLast**() - добавить в начало и в конец списка

removeFirst() и **removeLast**() - удалить первый и последний элементы

getFirst() и **getLast**() - получить первый и последний элементы



Если необходимо вставлять (или удалять) в середину коллекции много элементов, то лучше использовать LinkedList. Во всех остальных случаях – ArrayList.

LinkedList требует больше памяти для хранения такого же количества элементов, потому что кроме самого элемента хранятся еще указатели на следующий и предыдущий элементы списка, тогда как в ArrayList элементы просто идут по порядку.

Итератор

Итератор — это объект, который позволяет программисту пробежать по элементам коллекции.

Этот интерфейс позволяет пройти по очереди коллекцию объектов. При этом код, выполняющий итерирование, может ничего не знать об используемой коллекции.

Интерфейс Collection не является базовым. Интерфейс Collection расширяет интерфейс Iterable, у которого есть только один метод iterator(). Это значит что любая коллекция, которая есть наследником Iterable должна возвращать итератор.

Итератор это паттерн позволяющий получить доступ к элементам любой коллекции без вникания в суть ее реализации.

Реализация интерфейса предполагает, что с помощью вызова метода **next()** можно получить следующий элемент.

С помощью метода **hasNext()** можно узнать, есть ли следующий элемент, и не достигнут ли конец коллекции.

И если элементы еще имеются, то **hasNext()** вернет значение true.

Метод **hasNext()** следует вызывать перед методом **next()**, так как при достижении конца коллекции метод **next()** выбрасывает исключение NoSuchElementException.
И метод **remove()** удаляет текущий элемент, который был получен последним вызовом next().

Пример использования Итератора

```
public class CollectionApp {
    public static void main(String[] args) {
    ArrayList<String> name = new ArrayList<String>();
        name.add("Вася");
        name.add("Коля");
        name.add("Саша");
        name.add("Дима");

    Iterator<String> iter = name.iterator();
        while(iter.hasNext()){
            System.out.println(iter.next());
        }
    }
}
```

ListIterator

Интерфейс Iterator предоставляет ограниченный функционал. Гораздо больший набор методов предоставляет другой итератор - интерфейс ListIterator. Данный итератор используется классами, реализующими интерфейс List, то есть классами LinkedList, ArrayList и др.

Интерфейс ListIterator расширяет интерфейс Iterator и определяет ряд дополнительных методов:

void add(E obj): вставляет объект obj перед элементом, который должен быть возвращен следующим вызовом next()
boolean hasNext(): возвращает true, если в коллекции имеется следующий элемент, иначе возвращает false
boolean hasPrevious(): возвращает true, если в коллекции имеется предыдущий элемент, иначе возвращает false
E next(): возвращает следующий элемент, если такого нет, то генерируется исключение NoSuchElementException
E previous(): возвращает предыдущий элемент, если такого нет, то генерируется исключение NoSuchElementException
int nextIndex(): возвращает индекс следующего элемента. Если такого нет, то возвращается размер списка
int previousIndex(): возвращает индекс предыдущего элемента. Если такого нет, то возвращается число -1
void remove(): удаляет текущий элемент из списка. Таким образом, этот метод должен быть вызван после методов next() или previous(), иначе будет сгенерировано исключение IlligalStateException
void set(E obj): присваивает текущему элементу, выбранному вызовом методов next() или previous(), ссылку на объект obj

Во многих языках существует более компактная форма for для перебора элементов массивов - foreach. Конструкция foreach не требует ручного изменения переменной-шага для перебора - цикл автоматически выполняет эту работу.

В Java решили не добавлять новое ключевое слово, а просто сделали усовершенствованный вид цикла for, который имеет вид:

for(тип итер_пер : коллекция) блок_операторов
Для сравнения напишем цикл для вычисления суммы значений элементов массива традиционным способом:

```
int[] nums = { 1, 2, 3, 4, 5 };
int sum = 0;

for(int i = 0; i < 5; i++) sum += nums[i];
```

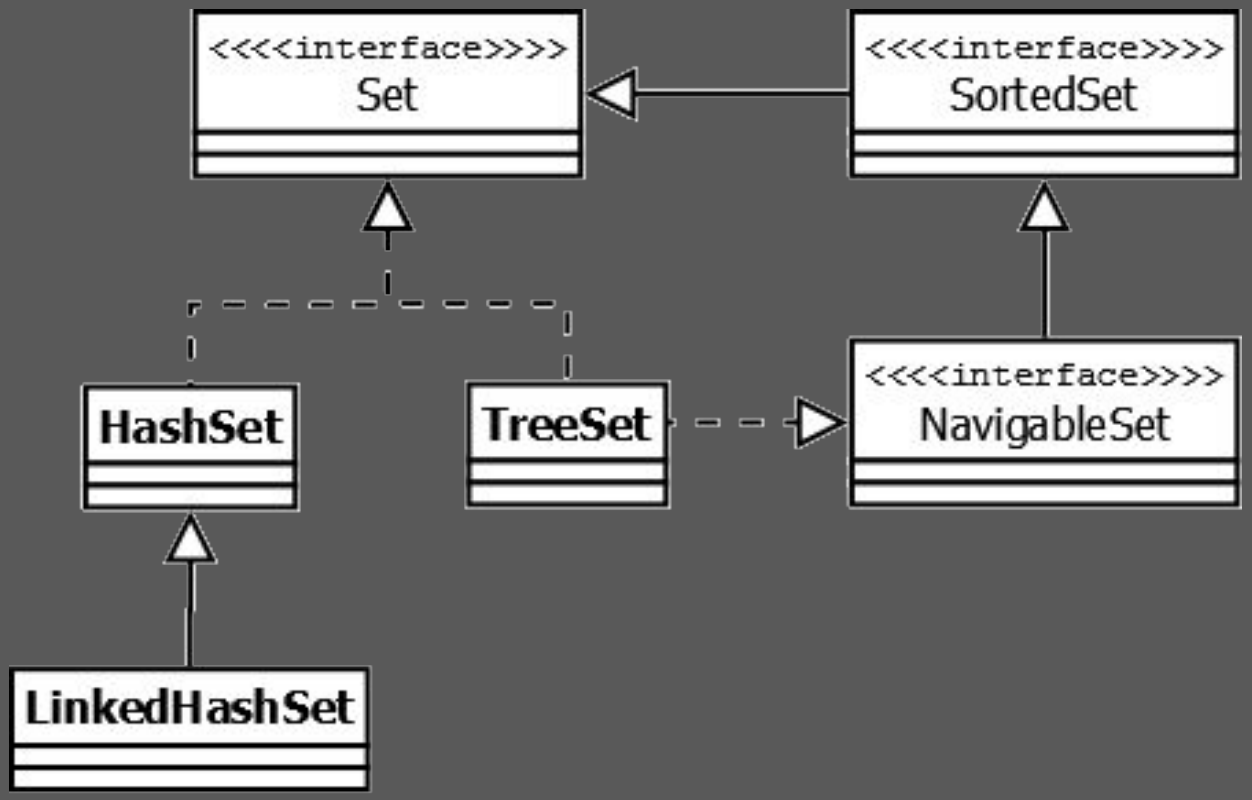
Этот код можно переписать следующим образом:

```
int[] nums = { 1, 2, 3, 4, 5 };
int sum = 0;

for(int i : nums) sum += i;
```

Пример использования Итератора

```
public class CollectionApp {
	public static void main(String[] args) {
	ArrayList<String> name = new ArrayList<String>();
		name.add("Вася");
		name.add("Коля");
		name.add("Саша");
		name.add("Дима");

	 ListIterator<String> listIter = name.listIterator();

	 while(listIter.hasNext()){

	    System.out.println(listIter.next());
	 }  // сейчас текущий элемент - Дима изменим значение этого элемента

	listIter.set("Миша");  // пройдемся по элементам в обратном порядке
	 while(listIter.hasPrevious()){

	    System.out.println(listIter.previous());
	 }
    }
}
```

Интерфейс Set



Интерфейс Set, унаследовавшись от Collection, не содержит никаких новых методов. В множествах Set каждый элемент хранится только в одном экземпляре при этом добавление повторяющихся элементов в **Set** не вызывает исключений, при добавлении они не попадают в множество.

Классы **HashSet, TreeSet и LinkedHashSet** реализуют интерфейс **Set.** В зависимости от реализации меняется порядок сортировки элементов.

В **HashSet** порядок элементов определяется по сложному алгоритму.

В **TreeSet**, объекты хранятся отсортированными по возрастанию.

В **LinkedHashSet** элементы хранятся в порядке добавления.

Механизм хеширования

Хэш-код — это целое число, которое является уникальным идентификатором содержимого объекта. От сюда вывод — у каждого объекта, с разными данными, свое уникальное число, с помощью которого мы можем судить о равенстве или неравенстве. За вычисление этого кода отвечает метод hashCode(), который переопределенный в наследниках Object.

Хеширование в простейшем представлении, это – способ преобразования любой переменной или объекта в уникальный код после применения любой формулы или алгоритма к их свойствам.

Хеш-функция должна возвращать одинаковый хеш-код всякий раз, когда она применена к одинаковым или равным объектам. Другими словами, два одинаковых объекта должны возвращать одинаковые хеш-коды по очереди.

Одинаковые объекты — это объекты одного класса с одинаковым содержимым полей.

1. для одного и того-же объекта, хеш-код всегда будет одинаковым;

2. если объекты одинаковые, то и хеш-коды одинаковые (но не наоборот, см. правило 3).

3. если хеш-коды равны, то входные объекты не всегда равны (коллизия);

4. если хеш-коды разные, то и объекты гарантированно разные;

Итог:

Если хеш-коды разные, то и входные объекты гарантированно разные.

Если хеш-коды равны, то входные объекты не всегда равны.

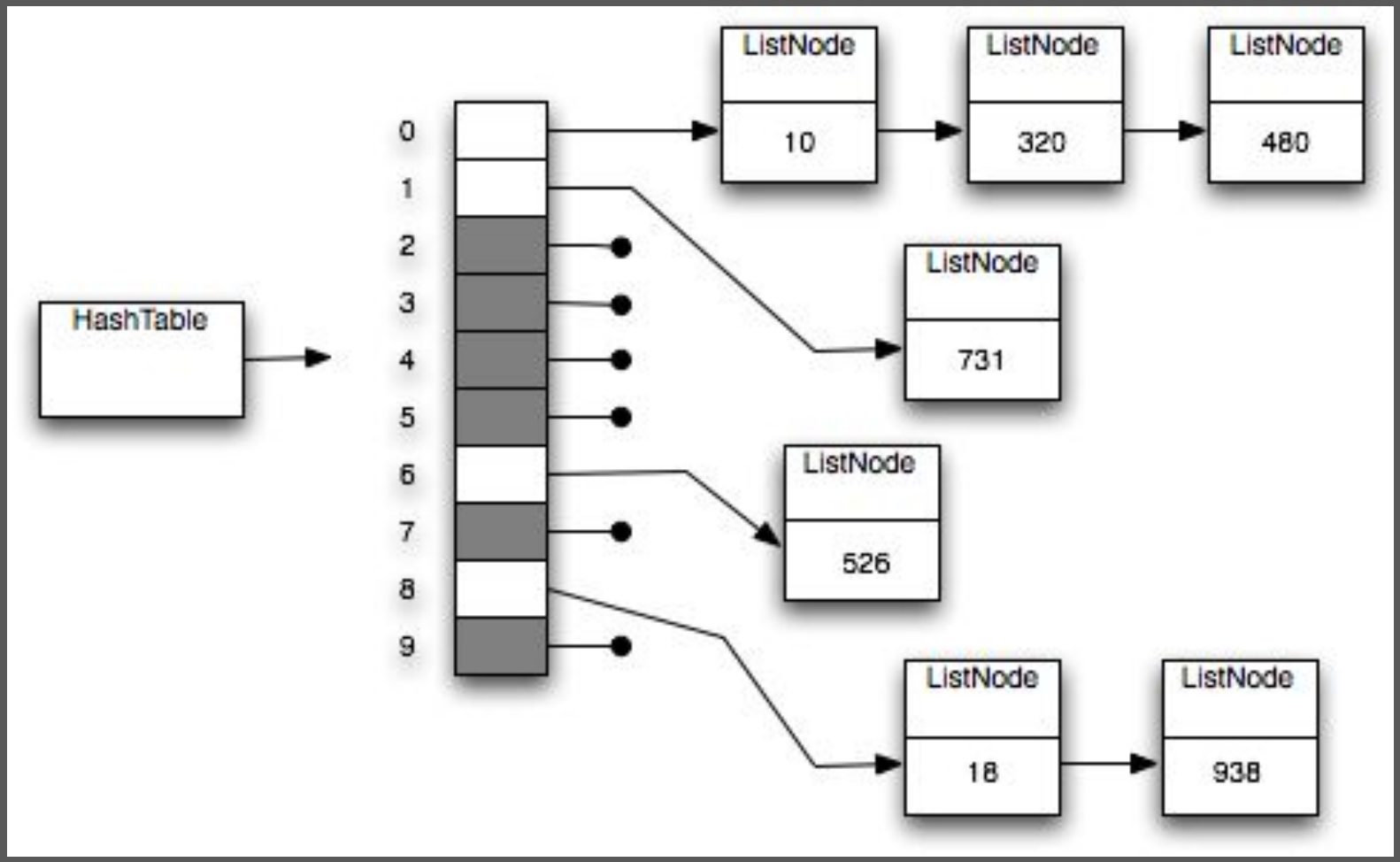
Ситуация, когда у разных объектов одинаковые хеш-коды называется — коллизией. Вероятность возникновения коллизии зависит от используемого алгоритма генерации хеш-кода.

Хэш Таблица

Таблица представляет собой обобщение обычного массива. Ключом массива может быть любой объект, для которого можно вычислить хеш-код. Интерфейс хеш-таблицы предоставляет нам следующие операции:

- Добавление новой пары ключ-значение
- Поиск значения по ключу
- Удаление пары ключ-значение по ключу

Ячейки массива называют корзинами они в свою очередь содержат связанный список объектов реализованный по принципу LinkedList. Список содержит объекты «ключь - значение» для Map, для Set ключом является сам объект с нулевым значением.


HashTable
0
1
2
3
4
5
6
7
8
9
ListNode
10
ListNode
320
ListNode
480
ListNode
731
ListNode
526
ListNode
18
ListNode
938

Добавление объектов в таблицу

Когда приходит новый объект, вычисляется хэш-код ключа, функция hashCode() может вернуть слишком высокое или слишком низкое значение хеш-кода. Для решения этой проблемы, существует функция hash() она получает значение хеш-кода объекта и приводит хеш-значение в соответствие с размером массива.

Затем вызывается функция indexFor(hash, table.length), для вычисления индекса массива (корзины), для элемента.

Зная индекс в массиве, мы получаем список (цепочку) элементов, привязанных к этой ячейке.

Далее проходим по списку и сравниваем ключи элементов списка с нашим ключом для поиска. Сравнение производим по функции equals(). Вернула функция true - значит нашли и перезаписали, не нашли добавили новый.

Интерфейс Set предоставляет нам следующие методы

add(E e) — добавляем элемент в коллекцию, если такого там ещё нет. Возвращает true, если элемент добавлен
addAll(Collection c) — добавляет все элементы коллекции c (если их ещё нет)
clear() — удаляет все элементы коллекции
contains(Object o) — возвращает true, если элемент есть в коллекции
containsAll(Collection c) — возвращает true, если все элементы содержатся в коллекции
equals(Object o) — проверяет, одинаковы ли коллекции
hashCode() — возвращает hashCode
isEmpty() — возвращает true если в коллекции нет ни одного элемента
iterator() — возвращает итератор по коллекции
remove(Object o) — удаляет элемент
removeAll(Collection c) — удаляет элементы, принадлежащие переданной коллекции
retainAll(Collection c) — удаляет элементы, не принадлежащие переданной коллекции
size() — количество элементов коллекции
toArray() — возвращает массив, содержащий элементы коллекции
toArray(T[] a) — также возвращает массив, но (в отличии от предыдущего метода, который возвращает массив объектов типа Object) возвращает массив объектов типа, переданного в параметре.

HashSet и LinkedHashSet

Название класса HashSet происходит от понятия хэш-функция. Класс Object имеет метод hashCode(), который используется классом HashSet для эффективного размещения объектов, заносимых в коллекцию. В классах объектов, заносимых в HashSet, этот метод должен быть переопределен (override).

Имеет два основных конструктора:

public HashSet() // Строит пустое множество
public HashSet(Collection c) // Строит множество из элементов коллекции

Методы аналогичны методам ArrayList за исключением того, что метод **add(Object o)** добавляет объект в множество только в том случае, если его там нет. Возвращаемое методом значение — **true**, если объект добавлен, и **false**, если нет

Класс **LinkedHashSet** расширяет класс HashSet, не добавляя никаких новых методов. Класс поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Поиск по этой коллекции происходит также по hashCode, но порядок будет всегда совпадать с очерёдностью добавления.

Упорядоченные множества (SortedSet)

Класс TreeSet<E> представляет структуру данных в виде бинарного дерева, в котором все объекты хранятся в ортсортированом порядке , порядок задается либо естественным следованием элементов, либо объектом, реализующим интерфейс сравнения Comparator.

В классе TreeSet четыре конструктора:

TreeSet () — создает пустой объект с естественным порядком элементов;

TreeSet (Comparator c) — создает пустой объект, в котором порядок задается объектом сравнения c;

TreeSet (Collection coll) — создает объект, содержащий все элементы коллекции coll , с естественным порядком ее элементов;

```
Set sorted = new TreeSet();
sorted.add(new Integer(2));
sorted.add(new Integer(3));
sorted.add(new Integer(1));
System.out.println(sorted); // Распечатает [1, 2, 3]
```

Сортировка и упорядочивание. Интерфейсы Comparable и Comparator

При добавлении новых элементов объект **TreeSet** автоматически проводит сортировку, помещая новый объект на правильное для него место.

Помещая объекты в **TreeSet** мы должны дать понять TreeSet как их сравнивать.

Для этого класс должен реализовывать интерфейс **Comparable**.

В интерфейсе Comparable объявлен всего один метод **compareTo(**Object obj), предназначенный для реализации упорядочивания объектов класса. Его удобно использовать при сортировке упорядоченных списков или массивов объектов.

Данный метод сравнивает вызываемый объект с obj. В отличие от метода equals, который возвращает true или false, compareTo возвращает:

0, если значения равны;

Отрицательное значение, если вызываемый объект меньше параметра;

Положительное значение , если вызываемый объект больше параметра.

Если вы хотите обеспечить способность сортировки ваших собственных классов, тогда вы должны реализовать(implement) Comparable и перегружать compareTo(Object obj) самостоятельно.

Пример использования Comparable

```
public class Student implements Comparable {
       public int student_id;
       public String name;

public Student(int student_id, String name)  {
    this.student_id = student_id;
    this.name = name;
  }

public int compareTo(Object obj)  {
      Student tmp = (Student) obj;
      if( this.student_id < tmp.student_id ) {
            return -1; /* текущее меньше полученного */
    }
      else if(this.student_id > tmp.student_id)  {
       return 1;  /* текущее больше полученного */
    }
            return 0;   /* текущее равно полученному */
    }
```

Теперь, когда в классе Student перегружен метод compareTo(Object obj), мы легко можем сортировать массив объектов типа Student.

```
public static void main(String[] args)
{
  /* Создание массива объектов Student */

  Student[] students = new Student[3];
  students[0] = new Student(52645, "Bob");
  students[1] = new Student(98765, "Will");
  students[2] = new Student(1354,   "Matt");

       Arrays.sort(students);
       System.out.println(students);

 TreeSet<Student> st = new TreeSet<Student>();
            st.add(new Student(52645,"Bob"));
            st.add(new Student(98765,"Will"));
            st.add(new Student(1354,"Matt"));

            System.out.println(st);

}
```

Интерфейс **Comparable** определяет логику сравнеиния объекта внутри своей реализации.
Интерфейс **Comparator** позволает определить логику сравнения объектов вне реализации этого типа и эту логику можно в любой момент подменить. В этом случае создается отдельный вспомогательный класс, реализующий интерфейс **Comparator**, и уже на основании объекта этого класса будет производиться сортировка.

В этом классе нужно реализовать метод **compare**(Object o1, Object o2).

Правила работы этого метода такие же, как и у compareTo(Object o)

```
Set<ObjectName> sortedSet = new TreeSet<ObjectName>(new
Comparator<ObjectName>() {
        public int compare(ObjectName o1, ObjectName o2) {
                return o1.toString().compareTo(o2.toString());
        }
});

sortedSet.addAll(unsortedSet);
```

Пример работы с Comparator

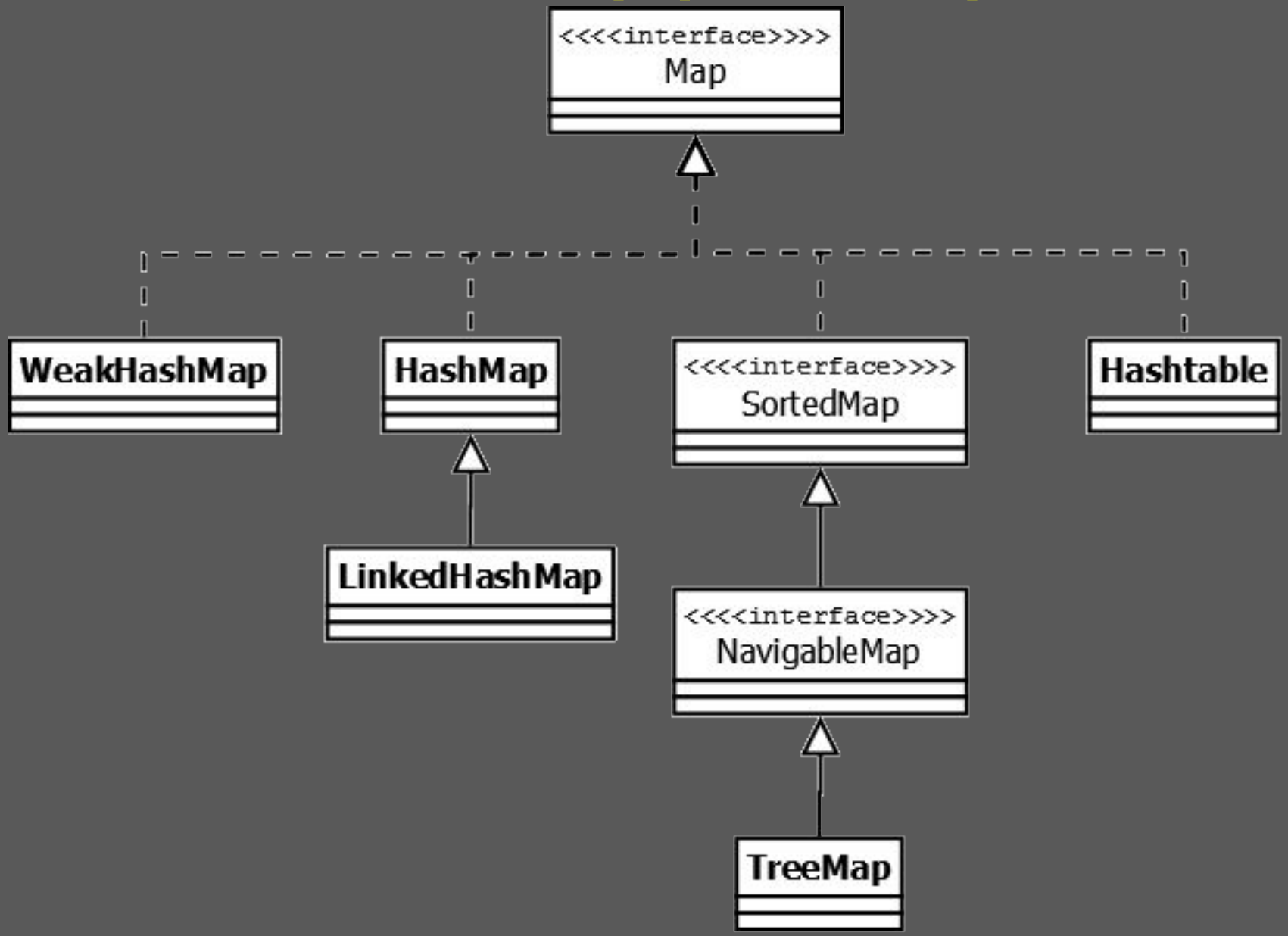
```
public class MyComparator implements Comparator<ObjectName>{

    public int compare(  ObjectName a, ObjectName  b){

        return a.getName().compareTo(b.getName());
    }
}

MyComparator mcomp = new MyComparator();

Set<ObjectName> sortedSet = new TreeSet<ObjectName>(mcomp);

sortedSet.addAll(unsortedSet);
```

Для создания TreeSet здесь используется одна из версий конструктора, которая в качестве параметра принимает компаратор. Теперь вне зависимости от того, реализован ли в классе интерфейс Comparable, логика сравнения и сортировки будет использоваться та, которая определена в классе компаратора.

Основные методы класса Collections

Все методы класса collections статические, ими можно пользоваться, не создавая экземпляры классу Collections.
static void sort (List coll) — сортирует в естественном порядке возрастания коллекцию coll, реализующую интерфейс List;
static void sort (List coll, Comparator c) — сортирует коллекцию coll
в порядке, заданном объектом c.
static int binarySearch(List coll, Object element) — отыскивает элемент element в отсортированной в естественном порядке возрастания коллекции coll и возвращает индекс элемента или отрицательное число, если элемент не найден; отрицательное число показывает индекс, с которым элемент element был бы вставлен в коллекцию;
static int binarySearchfList coll, Object element, Comparator c) — То же, но коллекция отсортирована в порядке , определенном объектом c .
static object max (Collection coll) — возвращает наибольший в естественном порядке элемент коллекции coll;
static Object max (Collection coll, Comparator c) — ТО же В порядке , заданном объектом c ;
static object mm (Collection coll) — возвращает наименьший в естественном порядке элемент коллекции;
static Object min(Collection coll, Comparator c) — ТО же В порядке , заданном объектом c .
Два метода "перемешивают" элементы коллекции в случайном порядке:
static void shuffle (List coll) — случайные числа задаются по умолчанию;
static void shuffle (List coll, Random r) — случайные числа определяются объектом r .
Метод **reverse (List coll)** меняет порядок расположения элементов на обратный.
Метод **copy (List from, List to)** копирует коллекцию from в коллекцию to .
Метод **fill (List coll, object element)** заменяет все элементы существующей коллекции coll элементом element .

Интерфейс Map



Наиболее распространенным классом отображений является HashMap.

Map представляют такие коллекции, в которых каждый объект представляет пару "ключ-значение". Такие коллекции облегчают поиск элемента, если нам известен ключ - уникальный идентификатор объекта.

Чтобы положить объект в коллекцию, используется метод put, а чтобы получить по ключу - метод get. Реализация интерфейса Map также позволяет получить наборы как ключей, так и значений.

А метод entrySet() возвращает набор всех элементов в виде объектов Map.Entry<K, V>.

```
Map pets = new HashMap();
        pets.put("Мурзик", "кот");
        pets.put("Бобик", "собака");
        pets.put("Кеша", "попугай");

Set<Map.Entry> set = pets.entrySet();
Iterator<Map.Entry> iter = set.iterator();
while (iter.hasNext()) {
        Map.Entry pet = iter.next();
        System.out.println(pet.getKey() + " это " + pet.getValue());        }
```

Пример Использования HashMap

```
Map<Integer, String> states = new HashMap<Integer, String>();

        states.put(1, "Германия");
        states.put(2, "Испания");
        states.put(4, "Франция");
        states.put(3, "Италия");

String first = states.get(1);  // получим объект по ключу 1
    System.out.println(first);
Set<Integer> keys = states.keySet();    // получим весь набор ключей
Collection<String> values = states.values(); // получить набор всех значений
    states.replace(1, "Бельгия");  //заменить элемент
    states.remove(2);    // удаление элемента по ключу 2

for(Map.Entry<Integer, String> item : states.entrySet()){
System.out.printf("Ключ: %d  Значение: %s \n", item.getKey(), item.getValue());
}
```

TreeMap

Класс TreeMap<K, V> представляет отображение в виде дерева. В отличие от коллекции TreeHash в TreeMap все объекты автоматически сортируются по возрастанию их ключей.

Класс TreeMap имеет следующие конструкторы:

TreeMap(): создает пустое отображение в виде дерева

TreeSet(Map<K, V> map): создает дерево, в которое добавляет все элементы из отображения map

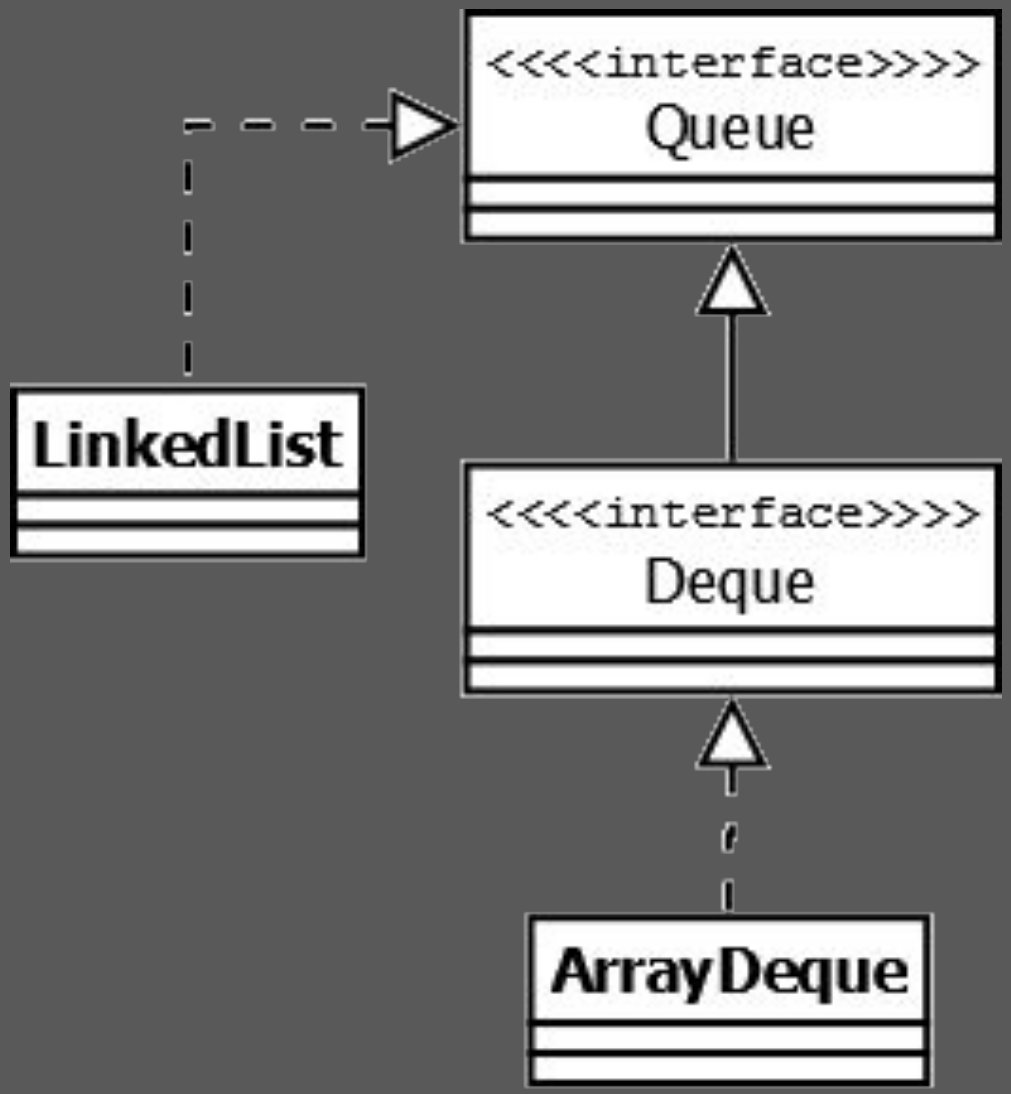
TreeSet(SortedMap<K, V> smap): создает дерево, в которое добавляет все элементы из отображения smap

TreeMap(Comparator<K> comparator): создает пустое дерево, где все добавляемые элементы впоследствии будут отсортированы компаратором.

ОСНОВНЫЕ МЕТОДЫ

void clear(): очищает коллекцию
boolean containsKey(Object k): возвращает true, если коллекция содержит ключ k
boolean containsValue(Object v): возвращает true, если коллекция содержит значение v
Set<Map.Entry<K, V>> entrySet(): возвращает набор элементов коллекции. Все элементы представляют объект Map.Entry
boolean equals(Object obj): возвращает true, если коллекция идентична коллекции, передаваемой через параметр obj
boolean isEmpty: возвращает true, если коллекция пуста
V get(Object k): возвращает значение объекта, ключ которого равен k. Если такого элемента не окажется, то возвращается значение null
V put(K k, V v): помещает в коллекцию новый объект с ключом k и значением v. Если в коллекции уже есть объект с подобным ключом, то он перезаписывается. После добавления возвращает предыдущее значение для ключа k, если он уже был в коллекции. Если же ключа еще не было в коллекции, то возвращается значение null
Set<K> keySet(): возвращает набор всех ключей отображения
Collection<V> values(): возвращает набор всех значений отображения
void putAll(Map<? extends K, ? extends V> map): добавляет в коллекцию все объекты из отображения map
V remove(Object k): удаляет объект с ключом k
int size(): возвращает количество элементов коллекции

Queue



Queue - коллекция, предназначенная для хранения элементов в порядке, нужном для их обработки. В дополнение к базовым операциям интерфейса Collection, очередь предоставляет дополнительные операции вставки, получения и контроля.

Очереди обычно, но не обязательно, упорядочивают элементы в FIFO (first-in-first-out, "первым вошел - первым вышел") порядке.

Методы Queue

Обобщенный интерфейс Queue<E> расширяет базовый интерфейс Collection и определяет поведение класса в качестве однонаправленной очереди. Свою функциональность он раскрывает через следующие методы:

Метод offer() вставляет элемент в очередь, если это не удалось - возвращает false. Методы remove() и poll() удаляют верхушку очереди и возвращают ее.

Какой элемент будет удален (первый или последний) зависит от реализации очереди. Методы remove() и poll() отличаются лишь поведением, когда очередь пустая: метод remove() генерирует исключение, а метод poll() возвращает null.

Методы element() и peek() возвращают (но не удаляют) верхушку очереди.
Также не разрешается добавлять элементы, которые не можно сравнить с помощью класса Comparator.

PriorityQueue - единственная прямая реализация интерфейса Queue (не считая LinkedList, который больше является списком, чем очередью).
Эта очередь упорядочивает элементы либо по их натуральному порядку (используя интерфейс Comparable), либо с помощью интерфейса Comparator, полученному в конструкторе.

Интерфейс Deque расширяет вышеописанный интерфейс Queue и определяет поведение двунаправленной очереди, которая работает как обычная однонаправленная очередь, либо как стек, действующий по принципу LIFO (последний вошел - первый вышел).
void addFirst(E obj): добавляет элемент в начало очереди
void addLast(E obj): добавляет элемент obj в конец очереди
E getFirst(): возвращает без удаления элемент из головы очереди. Если очередь пуста, генерирует исключение NoSuchElementException
E getLast(): возвращает без удаления последний элемент очереди. Если очередь пуста, генерирует исключение NoSuchElementException
boolean offerFirst(E obj): добавляет элемент obj в самое начало очереди. Если элемент удачно добавлен, возвращает true, иначе - false
boolean offerLast(E obj): добавляет элемент obj в конец очереди. Если элемент удачно добавлен, возвращает true, иначе - false
E peekFirst(): возвращает без удаления элемент из начала очереди. Если очередь пуста, возвращает значение null
E peekLast(): возвращает без удаления последний элемент очереди. Если очередь пуста, возвращает значение null
E pollFirst(): возвращает с удалением элемент из начала очереди. Если очередь пуста, возвращает значение null
E pollLast(): возвращает с удалением последний элемент очереди. Если очередь пуста, возвращает значение null
E pop(): возвращает с удалением элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException
void push(E element): добавляет элемент в самое начало очереди
E removeFirst(): возвращает с удалением элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException

Класс ArrayDeque

В Java очереди представлены рядом классов. Одни из низ - класс ArrayDeque<E>. Этот класс представляют обобщенную двунаправленную очередь, наследуя функционал от класса AbstractCollection и применяя интерфейс Deque.

В классе ArrayDeque определены следующие конструкторы:

ArrayDeque(): создает пустую очередь

ArrayDeque(Collection<? extends E> col): создает очередь, наполненную элементами из коллекции col

ArrayDeque(int capacity): создает очередь с начальной емкостью capacity. Если мы явно не указываем начальную емкость, то емкость по умолчанию будет равна 16

Пример использования ArrayDeque

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

    ArrayDeque<String> states = new ArrayDeque<String>();
    // стандартное добавление элементов
    states.add("Германия");
    states.add("Франция");
    // добавляем элемент в самое начало
    states.push("Великобритания");

    // получаем первый элемент без удаления
    String sFirst = states.getFirst();
    String sLast = states.getLast();

    while(states.peek()!=null){
        // извлечение с начала
        System.out.println(states.pop());
    }
    System.out.printf("Размер очереди: %d \n", states.size());
```

PriorityQueue

PriorityQueue - единственная прямая реализация интерфейса Queue (не считая LinkedList, который больше является списком, чем очередью).

Эта очередь упорядочивает элементы либо по их натуральному порядку (используя интерфейс Comparable), либо с помощью интерфейса Comparator, полученному в конструкторе. Например, следующим кодом упорядочиваются целочисленные элементы в обратном порядке:

Размер PriorityQueue ничем не ограничен, однако в этой очереди есть внутреннее значение capacity, которое регулирует размер очереди. Это значение должно быть как минимум больше, чем размер очереди. При добавлении элементов в PriorityQueue мощность (capacity) растет автоматически.

Также стоит отметить, что PriorityQueue является не синхронизированной очередью. Разные потоки не должны одновременно модифицировать экземпляр PriorityQueue. Для этого используйте потоко-безопасный класс PriorityBlockingQueue.