Java.SE.06

GENERIC & COLLECTIONS

Author: **Ihar Blinou**
Oracle Certified Java Instructor
ihar_blinou@epam.com

Содержание

ОПРЕДЕЛЕНИЕ КОЛЛЕКЦИЙ

Определение коллекций

Коллекции – это хранилища, поддерживающие различные способы **накопления** и **упорядочения** объектов с целью обеспечения возможностей **эффективного** доступа к ним. Применение **коллекций** обуславливается возросшими объемами обрабатываемой информации.

Коллекции в языке Java объединены в библиотеке классов **java.util** и представляют собой контейнеры, т.е объекты, которые группируют несколько элементов в отдельный модуль.

Коллекции используются для хранения, поиска, манипулирования и передачи данных.

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

Определение коллекций

Collections framework - это унифицированная архитектура для представления и манипулирования коллекциями.

Collections framework содержит:

- Интерфейсы
- Реализации (Implementations)
- Алгоритмы

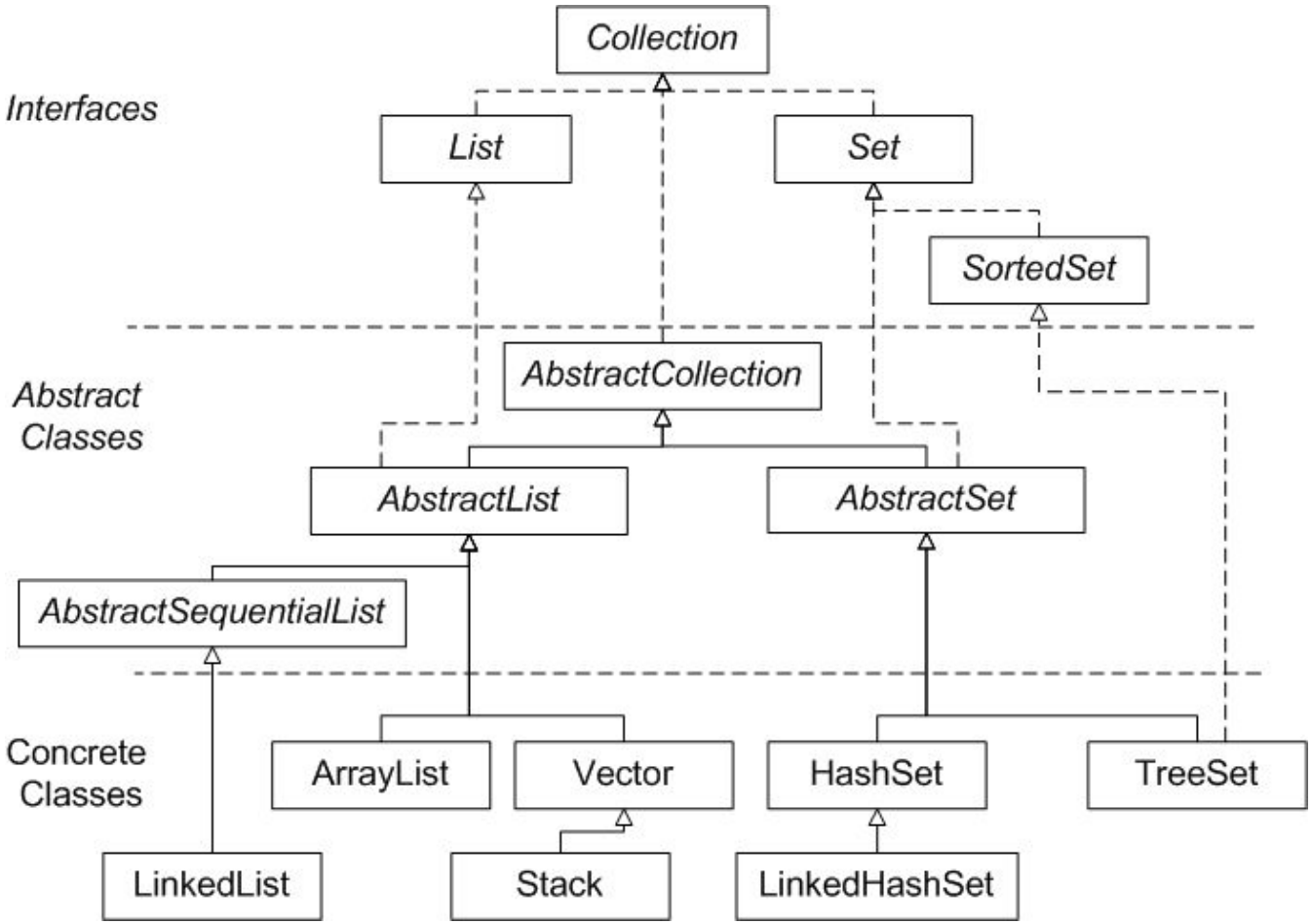
Определение коллекций

Интерфейсы коллекций:

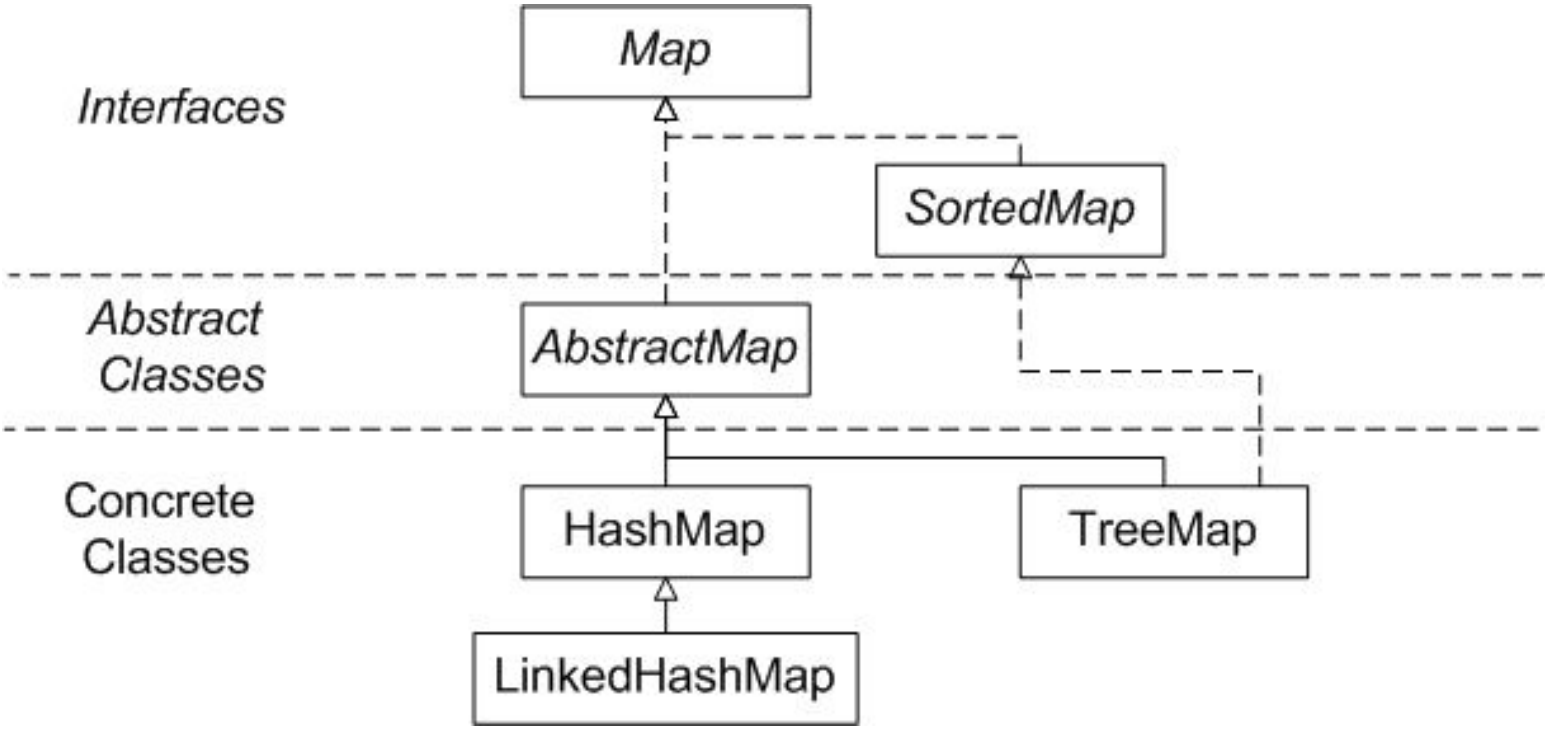
- **Collection<E>** – вершина иерархии остальных коллекций;
- **List<E>** – специализирует коллекции для обработки списков;
- **Set<E>** – специализирует коллекции для обработки множеств, содержащих уникальные элементы;
- **Map<K,V>** – карта отображения вида “ключ-значение”.

Интерфейсы позволяют манипулировать коллекциями независимо от деталей конкретной реализации, реализуя тем самым принцип полиморфизма

Определение коллекций



Определение коллекций



Определение коллекций

Все **конкретные классы** Java Collections Framework **реализуют** **Cloneable** и **Serializable** интерфейсы, следовательно, их экземпляры могут быть клонированы и сериализованы.

Определение коллекций

Реализации (Implementations)

Конкретные реализации интерфейсов могут быть следующих типов:

- General-purpose implementations
- Special-purpose implementations
- Concurrent implementations
- Wrapper implementations
- Convenience implementations
- Abstract implementations

Определение коллекций

General-Purpose Implementations - реализации общего назначения, наиболее часто используемые реализации,

- **HashSet, TreeSet, LinkedHashSet.**
- **ArrayList , LinkedList.**
- **HashMap, TreeMap, LinkedHashMap.**
- **PriorityQueue**

Определение коллекций

Special-Purpose Implementations - реализации специального назначения, разработаны для использования в специальных ситуациях и предоставляют нестандартные характеристики производительности, ограничения на использование или на поведение

- **EnumSet , CopyOnWriteArraySet.**
- **CopyOnWriteArrayList**
- **EnumMap, WeakHashMap, IdentityHashMap**

Определение коллекций

Concurrent implementations – потоковые реализации

- **ConcurrentHashMap**
- **LinkedBlockingQueue**
- **ArrayBlockingQueue**
- **PriorityBlockingQueue**
- **DelayQueue**
- **SynchronousQueue**
- **LinkedTransferQueue**

Определение коллекций

Wrapper implementations – реализация обертки, применяется для реализации нескольких типов в одном, чтобы обеспечить добавленную или ограниченную функциональность, все они находятся в классе **Collections.**

- public static <T> Collection<T> **synchronizedCollection**(Collection<T> c); public static <T> Set<T> **synchronizedSet**(Set<T> s); public static <T> List<T> **synchronizedList(**List<T> list); public static <K,V> Map<K,V> **synchronizedMap**(Map<K,V> m); public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s); и др.
- public static <T> Collection<T> **unmodifiableCollection**(Collection<? extends T> c); public static <T> Set<T> **unmodifiableSet**(Set<? extends T> s); public static <T> List<T> **unmodifiableList**(List<? extends T> list); public static <K,V> Map<K, V> **unmodifiableMap**(Map<? extends K, ? extends V> m); public static <T> SortedSet<T> **unmodifiableSortedSet**(SortedSet<? extends T> s); public static <K,V> SortedMap<K, V> **unmodifiableSortedMap**(SortedMap<K, ? extends V> m);

Определение коллекций

Convenience implementations – удобные реализации, выполнены обычно с использованием реализаций общего назначения и применением static factory methods для предоставления альтернативных путей создания (например, единичной коллекции)

Получить такие коллекции можно при помощи следующих методов

- **Arrays.asList**
- **Collections.nCopies**
- **Collections.singleton**
- **emptySet, emptyList, emptyMap. (из Collections)**

Определение коллекций

Abstract implementations – основа всех реализаций коллекций, которая облегчает создание собственных коллекций.

- **AbstractCollection**
- **AbstractSet**
- **AbstractList**
- **AbstractSequentialList**
- **AbstractQueue**
- **AbstractMap**

Определение коллекций

Алгоритмы (Algorithms)

Это методы, которые выполняют некоторые вычисления, такие как **поиск, сортировка** объектов, реализующих интерфейс **Collection**.

Они также реализуют принцип **полиморфизма**, таким образом один и тот же метод может быть использован в различных реализациях **Collection** интерфейса.

По существу, алгоритмы представляют универсальную функциональность.

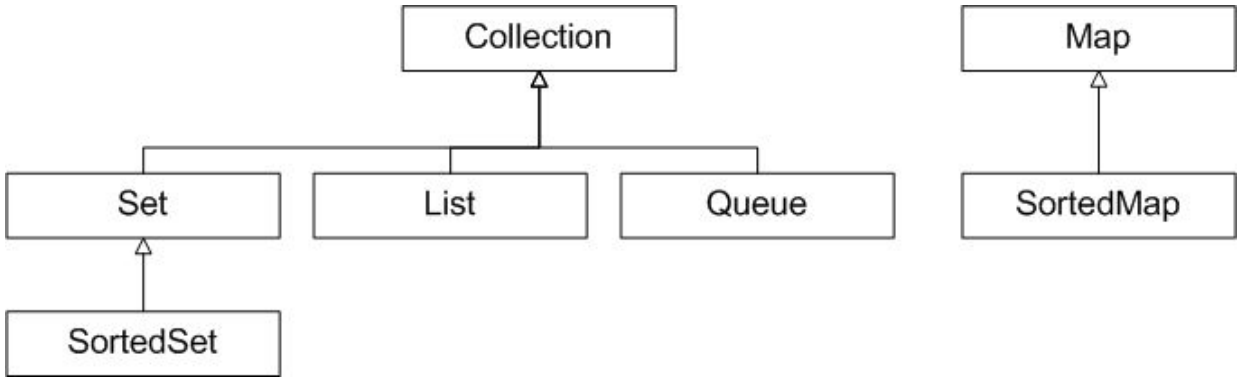
ИНТЕРФЕЙС COLLECTION

Интерфейс Collection

Интерфейс **Collection** - вершина иерархии коллекций

Интерфейс **Collection** - наименьший набор характеристик, реализуемых всеми коллекциями

JDK не предоставляет прямых реализаций этого интерфейса, но существует множество реализаций более специфичных подинтерфейсов таких как **Set** и **List**.



Интерфейс Collection

public interface Collection<E> extends Iterable<E> {

- **boolean equals(Object o);**
- **int size();** //возвращает количество элементов в коллекции;
- **boolean isEmpty();** // возвращает **true**, если коллекция пуста;
- **boolean contains(Object element);** //возвращает **true**, если коллекция содержит элемент **element**;
- **boolean add(E element);** //добавляет **element** к вызывающей коллекции и возвращает **true**, если объект добавлен, и **false**, если **element** уже элемент коллекции;
- **boolean remove(Object element);** //удаляет **element** из коллекции;
- **Iterator<E> iterator();** //возвращает итератор

Интерфейс Collection

- **boolean containsAll(Collection<?> c);** //возвращает **true**, если коллекция содержит все элементы из **c**;
- **boolean addAll(Collection<? extends E> c);** //добавляет все элементы коллекции к вызывающей коллекции;
- **boolean removeAll(Collection<?> c);** //удаление всех элементов данной коллекции, которые содержаться в **c**;
- **boolean retainAll(Collection<?> c);** //удаление элементов данной коллекции, которые не содержаться в коллекции **c**;
- **void clear();** //удаление всех элементов.
- **Object[] toArray();** //копирует элементы коллекции в массив объектов
- **<T> T[] toArray(T[] a);** //возвращает массив, содержащий все элементы коллекции

}

Интерфейс Collection

interface Iterable<T>{

- **Iterator<T> iterator();** // возвращает итератор по множеству элементов T

}

Интерфейс Collection

Класс **AbstractCollection** - convenience class, предоставляет частичную реализацию интерфейса **Collection**, реализует все методы, за исключением **size**() и **iterator**().

Интерфейс Collection

Некоторые методы интерфейса **Collection** могут быть не реализованы в подклассах (нет необходимости их реализовывать). В этом случае метод генерирует **java.lang.UnsupportedOperationException** (подкласс **RuntimeException**)

Это хорошее решение, которое следует использовать.

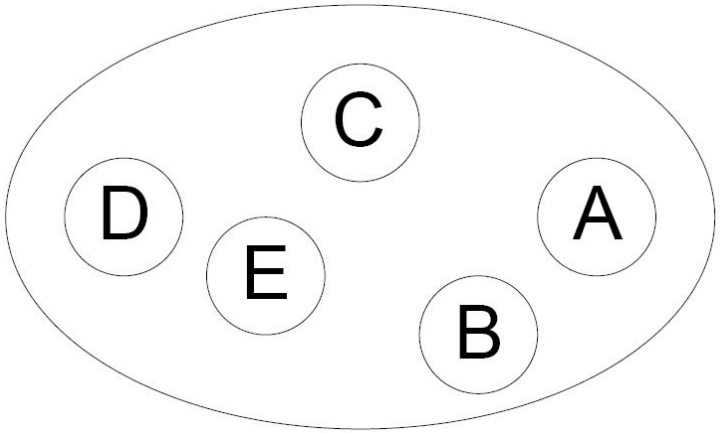
```
public void someMethod(){
    throw new java.lang.UnsupportedOperationException();
}
```

МНОЖЕСТВА SET

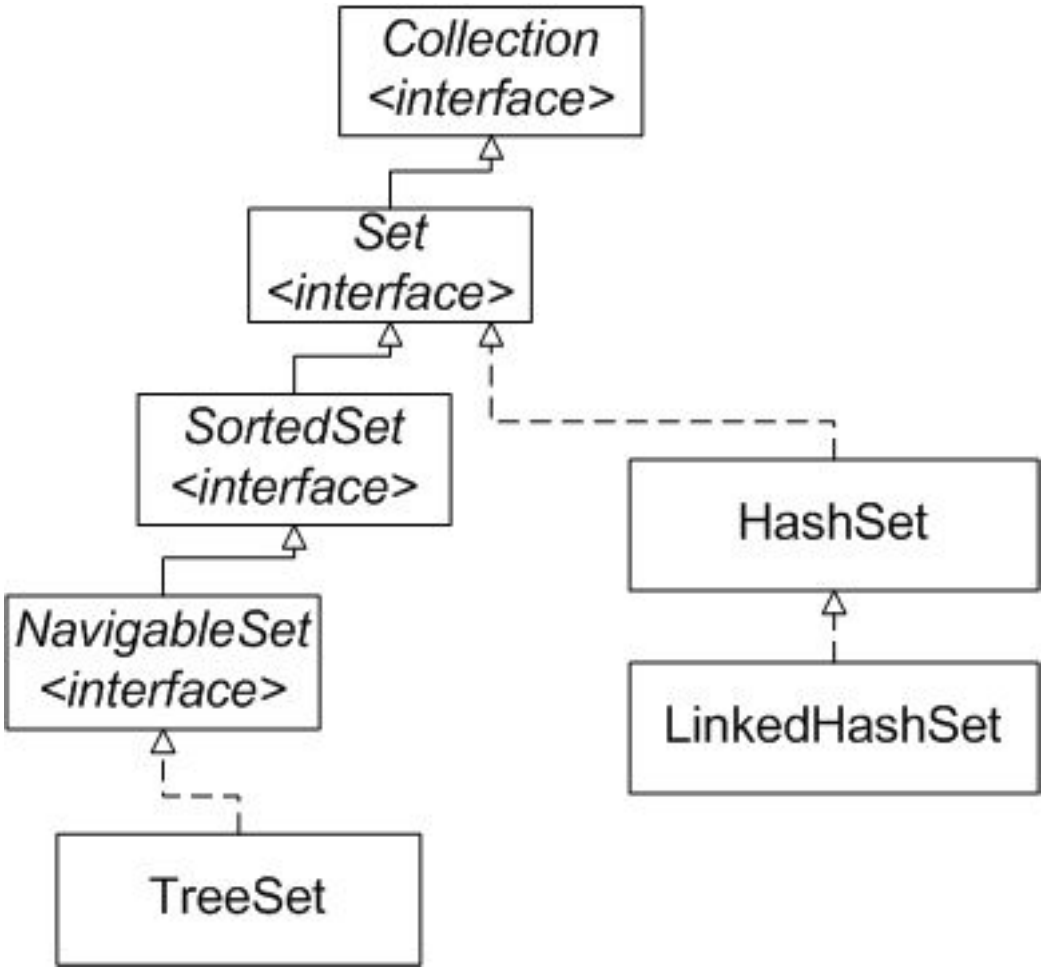
Множества Set

Множество — коллекция без повторяющихся элементов

Интерфейс **Set<E>** содержит методы, унаследованные **Collection<E>** и добавляет запрет на дублирующиеся элементы.



Множества Set



Множества Set

Интерфейс **Set** заботится об уникальности хранимых объектов, уникальность определятся реализацией метода **equals()**.

Множества Set

public interface Set<E> extends Collection<E> {

- **int size();** //возвращает количество элементов в множестве
- **boolean isEmpty();** //возвращает **true**, если множество пусто;
- **boolean contains(Object element);** //возвращает **true**, если множество содержит элемент **element**
- **boolean add(E element);** //добавляет **element** к вызывающему множеству и возвращает **true**, если объект добавлен, и **false**, если **element** уже элемент множества
- **boolean remove(Object element);** // удаляет **element** из множества
- **Iterator<E> iterator();** // возвращает итератор по множеству

Множества Set

- **boolean containsAll(Collection<?> c);** // возвращает **true**, если множество содержит все элементы коллекции **c**
- **boolean addAll(Collection<? extends E> c);** //добавление всех элементов из коллекции **c** во множество, если их еще нет
- **boolean removeAll(Collection<?> c);** //удаляет из множества все элементы, входящие в коллекцию **c**
- **boolean retainAll(Collection<?> c);** //сохраняет элементы во множестве, которые также содержаться и в коллекции **c**
- **void clear();** //удаление всех элементов
- **Object[] toArray();** //копирует элементы множества в массив объектов
- **<T> T[] toArray(T[] a);** //возвращает массив, содержащий все элементы множества

Множества Set

Set также добавляет соглашение на поведение методов **equals** и **hashCode**, позволяющих сравнивать множества даже если их реализации различны

- Два множества считаются равными, если они содержат одинаковые элементы

Множества Set

Правила сравнения на равенство

Метод **boolean equals(Object o)**

- **Рефлексивность**

  o1.equals(o1)

- **Симметричность**

  o1.equals(o2) == e2.equals(o1)

- **Транзитивность**

  o1.equals(o2) && o2.equals(o3) => o1.equals(o3)

- **Устойчивость**

  o1.equals(o2) не изменяется, если o1 и o2 не изменяются

- **Обработка null**

  o1.equals(null) == false

Множества Set

Интерфейс **SortedSet** из пакета **java.util**, расширяющий интерфейс **Set**, описывает упорядоченное множество, отсортированное по естественному порядку возрастания его элементов или по порядку, заданному реализацией интерфейса **Comparator**.

Множества Set

public interface SortedSet<E> extends Set<E>{

- **Comparator<? super E> comparator();** // возвращает способ упорядочения коллекции;
- **E first();** // минимальный элемент
- **SortedSet<E> headSet(E toElement);** //подмножество элементов, меньших **toElement**
- **E last();** // максимальный элемент
- **SortedSet<E> subSet(E fromElement, E toElement);** // подмножество элементов, меньших **toElement** и больше либо равных **fromElement**
- **SortedSet<E> tailSet(E fromElement);** // подмножество элементов, больших либо равных **fromElement**

}

Множества Set

Интерфейс **NavigableSet** добавляет возможность перемещения, "навигации" по отсортированному множеству.

Множества Set

public interface NavigableSet<E> extends SortedSet<E>{

- **E lower(E e);**
- **E floor(E e);**
- **E higher(E e);**
- **E ceiling(E e);**

// методы позволяют получить соответственно меньший, меньше или равный, больший, больше или равный элемент по отношению к заданному.

// метода возвращают соответственно первый и последний элементы, удаляя их из набора

- **E pollFirst();**
- **E pollLast();**

// возвращают итераторы коллекции в порядке возрастания и убывания элементов соответственно.

- **Iterator<E> iterator();**
- **Iterator<E> descendingIterator();**
- **NavigableSet<E> descendingSet();**

Множества Set

// методы, позволяющие получить подмножество элементов. Параметры **fromElement** и **toElement** ограничивают подмножество снизу и сверху, а флаги **fromInclusive** и **toInclusive** показывают, нужно ли в результирующий набор включать граничные элементы. **headSet** возвращает элементы с начала набора до указанного элемента, а **tailSet** - от указанного элемента до конца набора. Перегруженные методы без логических параметров включают в выходной набор первый элемент интервала, но исключают последний.

- **SortedSet<E> headSet(E toElement)**
- **NavigableSet<E> headSet(E toElement, boolean inclusive)**
- **NavigableSet<E> subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)**
- **SortedSet<E> subSet(E fromElement, E toElement)**
- **SortedSet<E> tailSet(E fromElement)**
- **NavigableSet<E> tailSet(E fromElement, boolean inclusive)**

Множества Set

Класс **AbstractSet** - convenience class , который наследуется от **AbstractCollection** и реализует интерфейс **Set**.

- Класс **AbstractSet** предоставляет реализацию методов **equals** и **hashCode**;
- **hash**-код множества – это сумма всех **hash-**кодов его элементов;
- методы **size** и **iterator** не реализованы.

Множества Set

HashSet – неотсортированная и неупорядоченная коллекция, для вставки элемента используются методы **hashCode()** и **equals(…).**

Чем эффективней реализован метод **hashCode()**, тем эффективней работает коллекция.

HashSet используется в случае, когда порядок элементов не важен, но важно чтобы в коллекции все элементы были уникальны.

Множества Set

Конструкторы HashSet

- **HashSet()** — создает пустое множество;
- **HashSet(Collection<? extends E> c)** — создает новое множество с элементами коллекции **c**;
- **HashSet(int initialCapacity)** — создает новое пустое множество размера **initialCapacity;**
- **HashSet(int initialCapacity, float loadFactor)** — создает новое пустое множество размера **initialCapacity** со степенью заполнения **loadFactor**.

Выбор слишком большой первоначальной вместимости (capacity) может обернуться потерей памяти и производительности.

Выбор слишком маленькой первоначальной вместимости (capacity) уменьшает производительность из-за копирования данных каждый раз, когда вместимость увеличивается.

Множества Set

Для эффективности объекты, добавляемые в множество должны реализовывать **hashCode**.

Метод **int hashCode()** - возвращает значение хэш-кода множества

Правила:

- Устойчивость

  hashCode() не изменяется, если объект не изменяется

- Согласованность с equals()

  o1.equals(o2) => o1.hashCode() == o2.hashCode()

Множества Set. Example 01

```
package _java._se._06.set;
import java.util.HashSet;
import java.util.Set;
public class SetExample {
    public static void main(String[] args) {
        Set<String> set = new HashSet<String>();
        set.add("London");
        set.add("Paris");
        set.add("New York");
        set.add("San Francisco");
        set.add("Berling");
        set.add("New York");
        System.out.println(set);
        for (Object element : set)
            System.out.print(element.toString());
    }
}
```

Результат:

```
[San Francisco, New York, Paris, Berling,
London]
San Francisco New York Paris Berling London
```

Множества Set

LinkedHashSet<E> — множество на основе хэша с сохранением порядка обхода.

Множества Set. Example 02

```
package _java._se._06.set;
import java.util.LinkedHashSet;
import java.util.Set;
public class LinkedHashSetExample {
    public static void main(String[] args) {
        Set<String> set = new LinkedHashSet<String>();
        // Add strings to the set
        set.add("London");
        set.add("Paris");
        set.add("New York");
        set.add("San Francisco");
        set.add("Berling");
        set.add("New York");
        System.out.println(set);
        // Display the elements in the hash set
        for (Object element : set)
            System.out.print(element.toString() + " ");
    }
}
```

Результат:

```
[London, Paris, New York, San Francisco,
Berling]
London Paris New York San Francisco Berling
```

Множества Set

TreeSet<E> – реализует интерфейс **NavigableSet<E>**, который поддерживает элементы в отсортированном по возрастанию порядке.

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 все добавляемые элементы реализуют интерфейсы Comparator и Comparable.

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

Множества Set

Используется в том случае, если необходимо использовать операции, определенные в интерфейсе **SortedSet**, **NavigableSet** или итерацию в определенном порядке.

Множества Set

Конструкторы TreeSet:

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

Множества Set

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

Методы **SortedSet<E> subSet(E from, E to), SortedSet<E> tailSet(E from)** и **SortedSet<E> 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 Set. Example 03

```
package _java._se._06.set;
import java.util.HashSet;
import java.util.Set;
import java.util.TreeSet;
public class TreeSetExample {
  public static void main(String[] args) {
    // Create a hash set
    Set<String> set = new HashSet<String>();
    // Add strings to the set
    set.add("London");
    set.add("Paris");
    set.add("New York");
    set.add("San Francisco");
    set.add("Berling");
    set.add("New York");
    TreeSet<String> treeSet = new TreeSet<String>(set);
    System.out.println(treeSet);
    // Display the elements in the hash set
    for (Object element : set)
        System.out.print(element.toString() + " ");
  }
}
```

Множества Set. Example 03

Результат:

```
[Berling, London, New York, Paris, San
Francisco]
San Francisco New York Paris Berling London
```

ИНТЕРФЕЙС ITERATOR

Интерфейс Iterator

Для обхода коллекции можно использовать:

- **for-each**

  Конструкция for-each является краткой формой записи обхода коллекции с использованием цикла for.

```
for (Object o: collection)
    System.out.println(o);
```

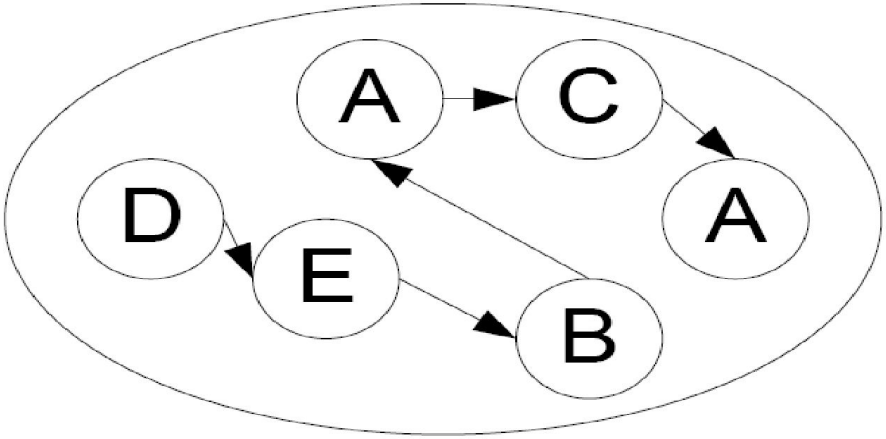
- **Iterator**

  Итератор это объект, который позволяет осуществлять обход коллекции и при желании удалять избранные элементы.

Интерфейс Iterator

Интерфейс **Iterator<E>** используется для доступа к элементам коллекции

Iterator<E> iterator() – возвращает итератор



Интерфейс Iterator

public interface Iterator {

- **boolean hasNext();** // возвращает true при наличии следующего элемента, а в случае его отсутствия возвращает false. Итератор при этом остается неизменным;
- **Object next();** // возвращает объект, на который указывает итератор, и передвигает текущий указатель на следующий итератор, предоставляя доступ к следующему элементу. Если следующий элемент коллекции отсутствует, то метод next() генерирует исключение ;
- **void remove();** // удаляет объект, возвращенный последним вызовом метода next()

}

Интерфейс Iterator

Исключения:

- **NoSuchElementException** – генерируется при достижении конца коллекции
- **ConcurrentModificationException** – генерируется при изменении коллекции

Интерфейс Iterator. Example 04

```
package _java._se._06.set;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
public class IteratorExample {
    public static void main(String[] args) {
        Set<String> set = new HashSet<String>();
        set.add("London");
        set.add("Paris");
        set.add("New York");
        set.add("San Francisco");
        set.add("Berling");
        set.add("New York");
        System.out.println(set);
        // Obtain an iterator for the hash set
        Iterator iterator = set.iterator();
        // Display the elements in the hash set
        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
    }
}
```

Интерфейс Iterator. Example 04

Результат:

```
[San Francisco, New York, Paris, Berling,
London]
San Francisco New York Paris Berling London
```

Интерфейс Iterator

Используйте Iterator вместо for-each если вам необходимо удалить текущий элемент.

- Конструкция **for-each** скрывает итератор, поэтому нельзя вызвать **remove**
- Также конструкция **for-each** не применима для фильтрации.

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext();)
        if (!cond(it.next()))
            it.remove();
}
```

Интерфейс Iterator

Чтобы удалить все экземпляры определенного элемента e из коллекции c воспользуйтесь следующим кодом:

```
c.removeAll(Collections.singleton(e));
```

Удалить все элементы null из коллекции

```
c.removeAll(Collections.singleton(null));
```

Collections.singleton(), статический метод, который возвращает постоянное множество, содержащее только определенный элемент.

СРАВНЕНИЕ ОБЪЕКТОВ. COMPATATOR, COMPARABLE

Сравнение коллекций. Comparator, Comparable

Естественный порядок сортировки (natural sort order) — естественный и реализованный по умолчанию (реализацией метода **compareTo** интерфейса **java.lang.Comparable**) способ сравнения двух экземпляров одного класса.

- **int compareTo(E other)** — сравнивает **this** объект с **other** и возвращает отрицательное значение если this<other, 0 — если они равны и положительное значение если this>other.

Сравнение коллекций. Comparator, Comparable

Реализация Comparable позволяет:

- Вызвать **Collections.sort** и**Collections.binarySearch**
- Вызывать **Arrays.sort** и **Arrays.binarySearch**
- Использовать такие объекты, как **keys** в **TreeMap**
- Использовать такие объекты, как **elements** в**TreeSet**

Сравнение коллекций. Comparator, Comparable

Метод **compareTo** должен выполнять следующие условия.

- **sgn(x.compareTo(y)) == -sgn(y.compareTo(x))**
- если **x.compareTo(y)** выбрасывает исключение, то и **y.compareTo(x)** должен выбрасывать то же исключение
- если **x.compareTo(y)>0** и **y.compareTo(z)>0**, тогда **x.compareTo(z)>0**
- если **x.compareTo(y)==0**, и **x.compareTo(z)==0**, то и **y.compareTo(z)==0**
- **x.compareTo(y)==0**, тогда и только тогда, когда **x.equals(y) ;** (правило рекомендуемо но не обязательно)

Сравнение коллекций. Comparator, Comparable. Example 05

```
package _java._se._06.comparable;
public class Person implements Comparable {
    private String firstName;
    private String lastName;
    private int age;
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
```

Сравнение коллекций. Comparator, Comparable. Example 05

```
public int getAge() {
     return age;
}
public void setAge(int age) {
     this.age = age;
}


public int compareTo(Object anotherPerson)
               throws ClassCastException {
     if (!(anotherPerson instanceof Person))
          throw new ClassCastException(
               "A Person object expected.");
     int anotherPersonAge =
               ((Person) anotherPerson).getAge();
     return this.age - anotherPersonAge;
}
}
```

Сравнение коллекций. Comparator, Comparable. Example 05

```
package _java._se._06.comparable;
import java.util.Arrays;
public class Testing {
    public static void main(String[] args) {
        Person[] persons = new Person[4];
        persons[0] = new Person();
        persons[0].setFirstName("Elvis");
        persons[0].setLastName("Goodyear");
        persons[0].setAge(56);
        persons[1] = new Person();
        persons[1].setFirstName("Stanley");
        persons[1].setLastName("Clark");
        persons[1].setAge(8);
        persons[2] = new Person();
        persons[2].setFirstName("Jane");
        persons[2].setLastName("Graff");
        persons[2].setAge(16);
        persons[3] = new Person();
        persons[3].setFirstName("Nancy");
        persons[3].setLastName("Goodyear");
        persons[3].setAge(69);
```

Сравнение коллекций. Comparator, Comparable. Example 05

```
    System.out.println("Natural Order");
    for (int i = 0; i < 4; i++) {
        Person person = persons[i];
        String lastName = person.getLastName();
        String firstName = person.getFirstName();
        int age = person.getAge();
        System.out.println(lastName + ", " + firstName + ". Age:"
+ age);
    }
    Arrays.sort(persons);
    System.out.println();
    System.out.println("Sorted by age");
    for (int i = 0; i < 4; i++) {
        Person person = persons[i];
        String lastName = person.getLastName();
        String firstName = person.getFirstName();
        int age = person.getAge();
        System.out.println(lastName + ", " + firstName + ". Age:"
+ age);
    }
}
}
```

Сравнение коллекций. Comparator, Comparable. Example 05

Результат:

```
Natural Order
Goodyear, Elvis.
Age:56
Clark, Stanley. Age:8
Graff, Jane. Age:16
Goodyear, Nancy.
Age:69
Sorted by age
Clark, Stanley. Age:8
Graff, Jane. Age:16
Goodyear, Elvis.
Age:56
Goodyear, Nancy.
Age:69
```

Сравнение коллекций. Comparator, Comparable

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

Для этого необходимо реализовать метод **int compare(T ob1, T ob2)**, принимающий в качестве параметров два объекта для которых должно быть определено возвращаемое целое значение, знак которого и определяет правило сортировки.

Этот метод автоматически вызывается методом **public 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 правило сортировки.

Сравнение коллекций. Comparator, Comparable

java.util.Comparator — содержит два метода:

- **int compare(T o1, T o2)** — сравнение, аналогичное **compareTo**
- **boolean equals(Object obj)** — **true** если **obj** это **Comparator** и у него такой же принцип сравнения.

Сравнение коллекций. Comparator, Comparable. Example 06

```
package _java._se._06.comparator;
public abstract class GeometricObject {
     public abstract double getArea();
}
```

```
package _java._se._06.comparator;
public class Rectangle extends GeometricObject {
     private double sideA;
     private double sideB;
     public Rectangle(double a, double b) {
          sideA = a;
          sideB = b;
     }
     @Override
     public double getArea() {
          return sideA * sideB;
     }
}
```

Сравнение коллекций. Comparator, Comparable. Example 06

```
package _java._se._06.comparator;
public class Circle extends GeometricObject {

	private double radius;

	public Circle(double r){
		radius = r;
	}
	@Override
	public double getArea() {
		// TODO Auto-generated method stub
		return 2*3.14*radius*radius;
	}
}
```

Сравнение коллекций. Comparator, Comparable. Example 06

```
package _java._se._06.comparator;
import java.util.Comparator;
public class GeometricObjectComparator
        implements Comparator<GeometricObject>,
        java.io.Serializable {
    private static final long serialVersionUID = 1L;

    public int compare(GeometricObject o1,
                       GeometricObject o2) {
        double area1 = o1.getArea();
        double area2 = o2.getArea();
        if (area1 < area2) {
            return -1;
        } else if (area1 == area2) {
            return 0;
        } else {
            return 1;
        }
    }
}
```

Сравнение коллекций. Comparator, Comparable. Example 06

```
package _java._se._06.comparator;
import java.util.Comparator;
import java.util.Set;
import java.util.TreeSet;
public class TestTreeSetWithComparator {
  public static void main(String[] args) {
    Comparator comparator
        = new GeometricObjectComparator();
    Set<GeometricObject> set
          = new TreeSet<GeometricObject>(comparator);
    set.add(new Rectangle(4, 5));
    set.add(new Circle(40));
    set.add(new Circle(40));
    set.add(new Rectangle(4, 1));
    System.out.println("A sorted set of geometric objects");
    for (GeometricObject elements : set) {
          System.out.println("area = " + elements.getArea());
    }
  }
}
```

Сравнение коллекций. Comparator, Comparable. Example 06

Результат:

```
A sorted set of geometric
objects
area = 4.0
area = 20.0
area = 10048.0
```

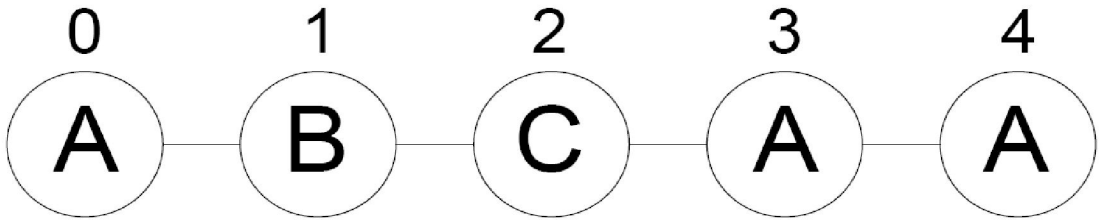
СПИСКИ LIST

Списки List

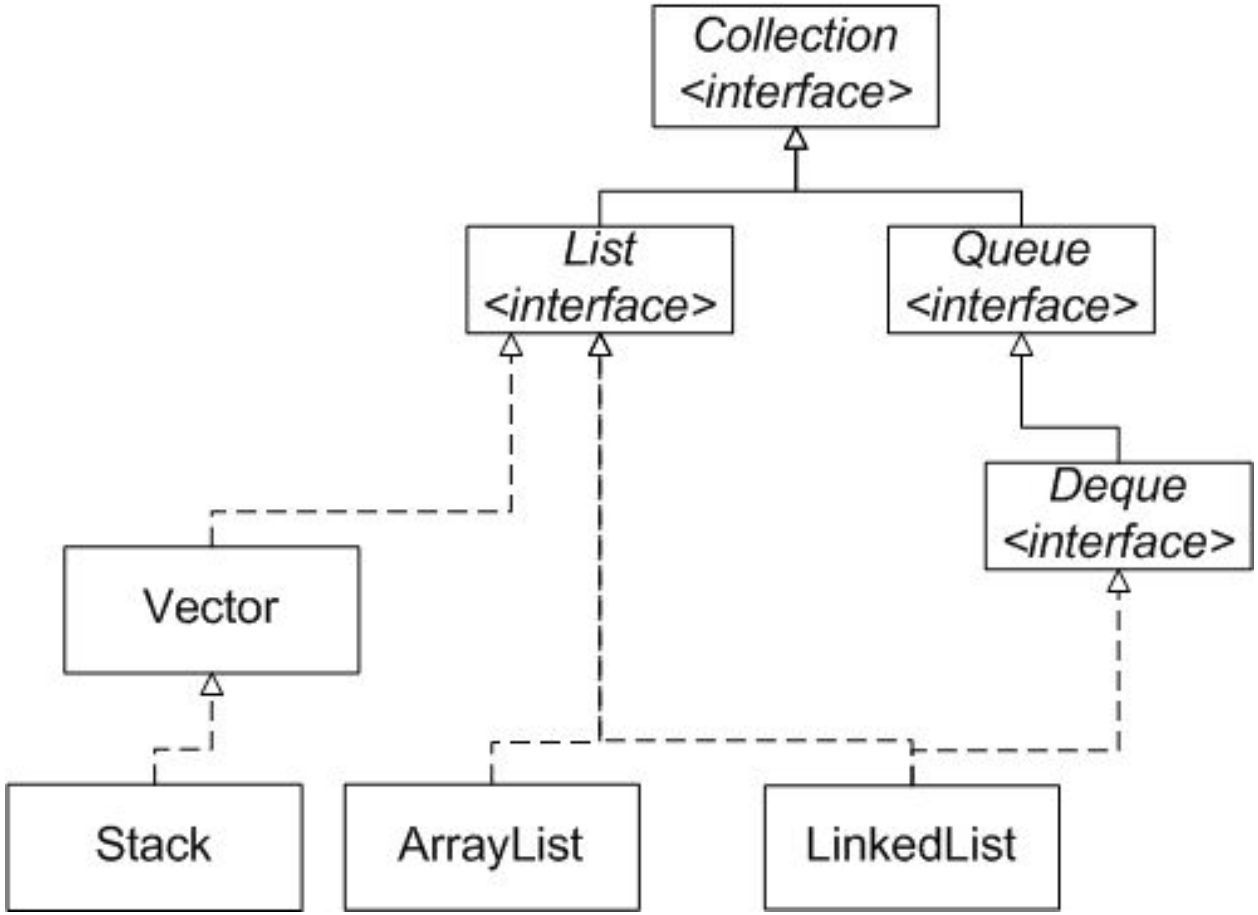
Список - упорядоченная коллекция (иногда называется sequence)

Список может содержать повторяющиеся элементы.

Интерфейс List сохраняет последовательность добавления элементов и позволяет осуществлять доступ к элементу по индексу.



Списки List



Списки List

public interface List<E> extends Collection<E> {

- **E get(int index);** //возвращает объект, находящийся в позиции **index**;
- **E set(int index, E element);** //заменяет элемент, находящийся в позиции **index** объектом **element**;
- **boolean add(E element);** //добавляет элемент в список
- **void add(int index, E element);** //вставляет элемент **element** в позицию **index**, при этом список раздвигается
- **E remove(int index);** //удаляет элемент, находящийся на позиции **index**
- **boolean addAll(int index, Collection<? extends E> c);** //добавляет все элементы коллекции c в список, начиная с позиции **index**

Списки List

- **int indexOf(Object o);** //возвращает индекс первого появления элемента **o** в списке;
- **int lastIndexOf(Object o);** //возвращает индекс последнего появления элемента **o** в списке;
- **ListIterator<E> listIterator();** //возвращает итератор на список
- **ListIterator<E> listIterator(int index);** //возвращает итератор на список, установленный на элемент с индексом **index**
- **List<E> subList(int from, int to);** //возвращает новый список, представляющий собой часть данного (начиная с позиции **from** до позиции **to-1** включительно).

}

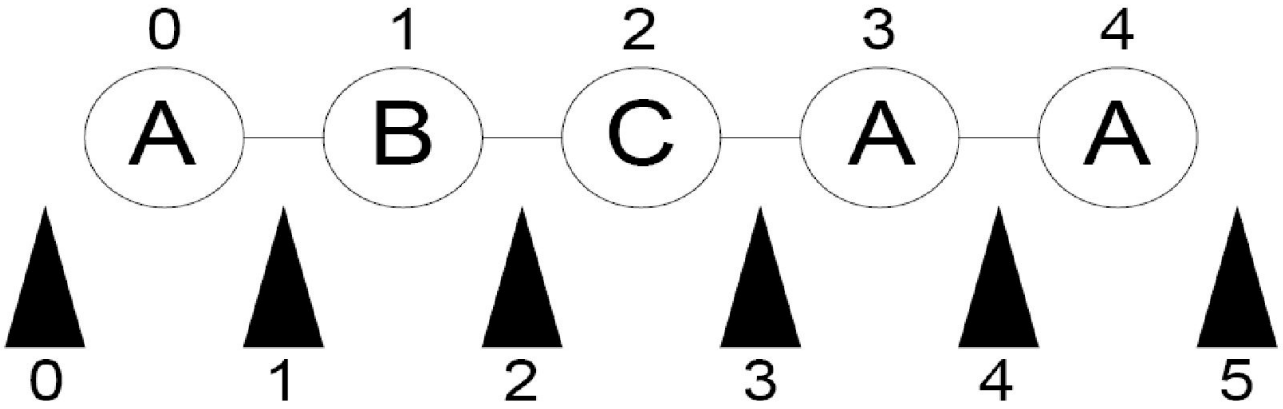
Списки List

Класс **AbstractList** предоставляет частичную реализацию для интерфейса **List**.

Класс **AbstractSequentialList** расширяет **AbstractList**, чтобы предоставить поддержку для связанных списков.

Списки List

ListIterator<E> - это итератор для списка



Списки List

interface ListIterator<E> extends Iterator{

- **boolean hasNext() / boolean hasPrevious()** // проверка
- **E next() / E previous ()** // взятие элемента
- **int nextIndex() / int previousIndex()** // определение индекса
- **void remove()** // удаление элемента
- **void set(E o)** // изменение элемента
- **void add(E o)** // добавление элемента

}

```
List list = new LinkedList();
…
for (ListIterator li = list.listIterator(list.size()); li.hasPrevious(); ){
      System.out.println(li.previous());
}
```

Списки List

ArrayList<E> – список на базе массива (реализация List)

- Достоинства
  - Быстрый доступ по индексу
  - Быстрая вставка и удаление элементов с конца
- Недостатки
  - Медленная вставка и удаление элементов

Аналогичен **Vector** за исключением *потокобезопасности*

Применения:

- “Бесконечный” массив
- Стек

Конструкторы ArrayList

- **ArrayList()** — пустой список
- **ArrayList(Collection<? extends E> c)** — копия коллекции
- **ArrayList(int initialCapacity)** — пустой список заданной вместимости

Вместимость — реальное количество элементов

Дополнительные методы

- **void ensureCapacity(int minCapacity)** — определение вместимости
- **void trimToSize()** — “подгонка” вместимости

Списки List

LinkedList<E> – двусвязный список (реализация List)

- Достоинства
  - Быстрое добавление и удаление элементов
- Недостатки
  - Медленный доступ по индексу

Рекомендуется использовать, если необходимо часто добавлять элементы в начало списка или удалять внутренний элемент списка

Применения:

- Стек
- Очередь
- Дек

Списки List

Конструкторы LinkedList

- **LinkedList<E> ()** //пустой список
- **LinkedList(Collection<? extends E> c)** //копия коллекции

Дополнительные методы

- **void addFirst(E o)** //добавить в начало списка
- **void addLast(E o)** // добавить в конец списка
- **E removeFirst()** // удалить первый элемент
- **E removeLast()** //удалить последний элемент
- **E getFirst()**
- **E getLast()**

Списки List. Example 07

```
package _java._se._06.list;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
public class ListExample {
  public static void main(String[] args) {
    List<Integer> arrayList =  new ArrayList<Integer>();
    arrayList.add(1);  // 1 is autoboxed to new Integer(1)
    arrayList.add(2);
    arrayList.add(3);
    arrayList.add(1);
    arrayList.add(4);
    arrayList.add(0, 10);
    arrayList.add(3, 30);
    System.out.println(
                        "A list of integers in the array list:" );
    System.out.println(arrayList);
```

Списки List. Example 07

```
    LinkedList<Object> linkedList
             = new LinkedList<Object>(arrayList);
    linkedList.add(1, "red");
    linkedList.removeLast();
    linkedList.addFirst("green");
    System.out.println("Display the linked list forward:");

    ListIterator listIterator = linkedList.listIterator();
    while (listIterator.hasNext()) {
        System.out.print(listIterator.next() + " ");
    }
    System.out.println();
    System.out.println("Display the linked list backward:");
    listIterator = linkedList.listIterator(linkedList.size());
    while (listIterator.hasPrevious()) {
        System.out.print(listIterator.previous() + " ");
    }
  }
}
```

Списки List. Example 07

Результат:

```
A list of integers in the array
list:
[10, 1, 2, 30, 3, 1, 4]
Display the linked list forward:
green 10 red 1 2 30 3 1
Display the linked list backward:
1 3 30 2 1 red 10 green
```

ОЧЕРЕДИ QUEUE

Очереди Queue

Очередь, предназначенная для размещения элемента перед его обработкой.

Расширяет коллекцию методами для вставки, выборки и просмотра элементов

Очередь – хранилище элементов, предназначенных для обработки.

Очереди Queue

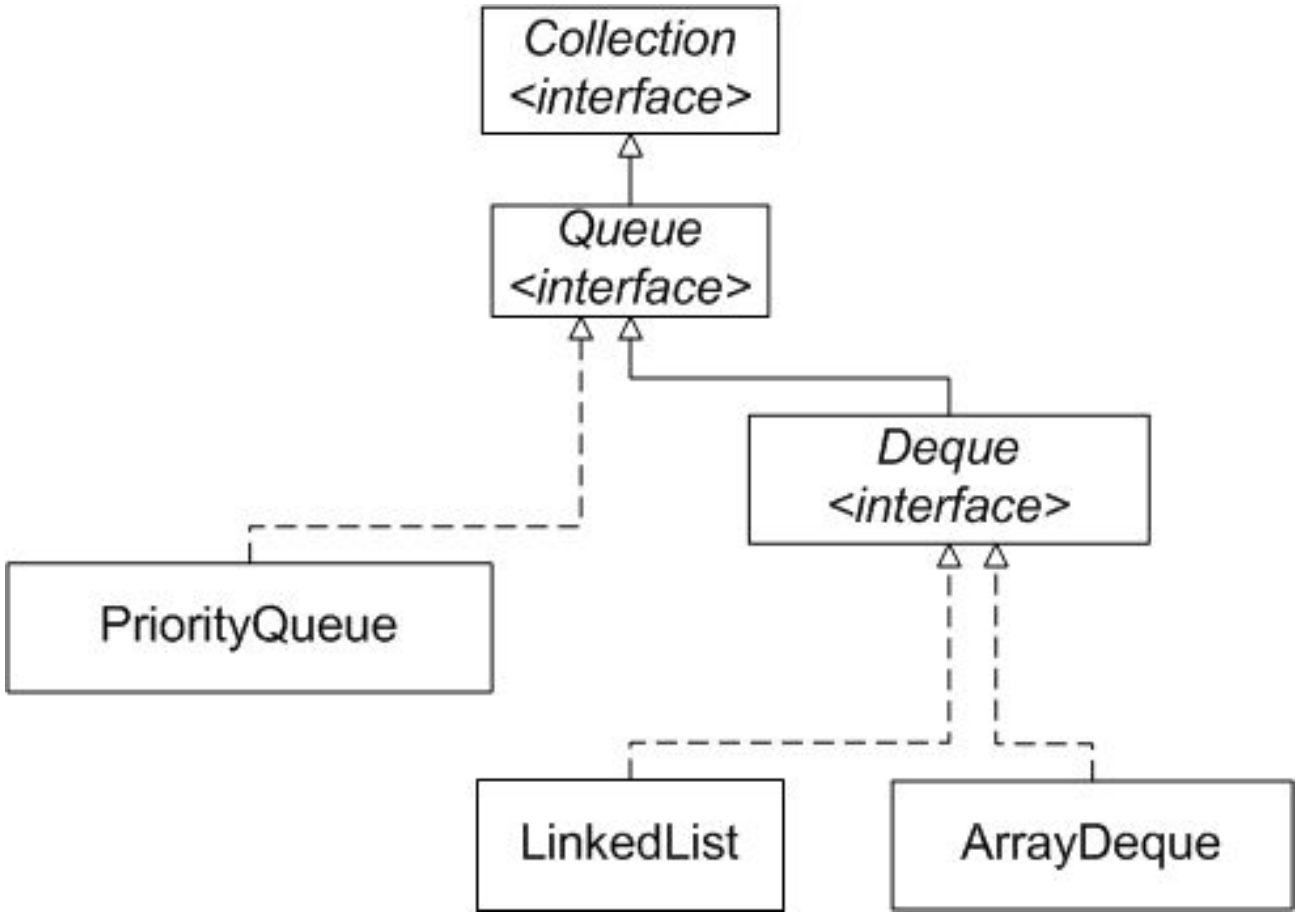
Кроме базовых методов **Collection** очередь(**Queue**) предоставляет дополнительные методы по добавлению, извлечению и проверке элементов.

Чаще всего порядок выдачи элементов соответствует **FIFO (first-in, first-out)**, но в общем случае определяется конкретной реализацией.

Очереди не могут хранить **null**.

У очереди может быть ограничен размер.

Очереди Queue



Очереди Queue

public interface Queue<E> extends Collection<E> {

- **E element();** // возвращает, но не удаляет головной элемент очереди
- **boolean offer(E o);** //добавляет в конец очереди новый элемент и возвращает true, если вставка удалась.
- **E peek();** // возвращает первый элемент очереди, не удаляя его.
- **E poll();** // возвращает первый элемент и удаляет его из очереди
- **E remove();** // возвращает и удаляет головной элемент очереди

}

Очереди Queue

Класс **AbstractQueue** – реализует методы интерфейса Queue:

- size()
- offer(Object o)
- peek()
- poll()
- iterator()

Очереди Queue. Example 08

```
package _java._se._06.queue;
public class QueueExample {
  public static void main(String[] args) {
    java.util.Queue<String> queue
            = new java.util.LinkedList<String>();
    queue.offer("Oklahoma");
    queue.offer("Indiana");
    queue.offer("Georgia");
    queue.offer("Texas");
    while (queue.size() > 0)
        System.out.print(queue.remove() + " ");
  }
}
```

Результат:

```
Oklahoma Indiana Georgia
Texas
```

Очереди Queue

Интерфейс **Deque** позволяет реализовать двунаправленная очередь, разрешающую вставку и удаление элементов в два конца очереди.

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

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 какое-либо из значений (**null** или **false** в зависимости от типа операции).

Очереди Queue

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

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

Метод **add(e)** унаследован от интерфейса **Queue** и абсолютно аналогичен методу **addLast(e)** интерфейса **Deque.**

Очереди Queue

ArrayDeque - эффективная реализация интерфейса **Deque** переменного размера

Конструкторы:

- **ArrayDeque();** // создает пустую двунаправленную очередь с вместимостью 16 элементов
- **ArrayDeque(Collection<? extends E> c);** // создает двунаправленную очередь из элементов коллекции с в том порядке, в котором они возвращаются итератором коллекции **c**.
- **ArrayDeque(int numElements);** // создает пустую двунаправленную очередь с вместимостью **numElements**.

Очереди Queue. Example 09

```
package _java._se._06.queue;
public class DequeExample {
  public static void main(String[] args) {
    java.util.Deque<String> deque
            = new java.util.LinkedList<String>();
    deque.offer("Oklahoma");
    deque.offer("Indiana");
    deque.addFirst("Texas");
    deque.offer("Georgia");
    while (deque.size() > 0)
        System.out.print(deque.remove() + " ");
  }
}
```

Результат:

```
Texas Oklahoma Indiana
Georgia
```

Очереди Queue. Example 10

```
package _java._se._06.queue;
import java.util.ArrayDeque;
import java.util.Deque;
public class ArrayDequeExample {
    public static void main(String args[]) {
        Deque<String> stack = new ArrayDeque<String>();
        Deque<String> queue = new ArrayDeque<String>();
        stack.push("A");
        stack.push("B");
        stack.push("C");
        stack.push("D");
        while (!stack.isEmpty())
            System.out.print(stack.pop() + " ");
        queue.add("A");
        queue.add("B");
        queue.add("C");
        queue.add("D");
        while (!queue.isEmpty())
            System.out.print(queue.remove() + " ");
    }
}
```

Очереди Queue. Example 10

Результат:

```
D C B A A B C
D
```

Очереди Queue

PriorityQueue – это класс очереди с приоритетами. По умолчанию очередь с приоритетами размещает элементы согласно естественному порядку сортировки используя Comparable. Элементу с наименьшим значением присваивается наибольший приоритет. Если несколько элементов имеют одинаковый наивысший элемент – связь определяется произвольно.

Также можно указать специальный порядок размещения, используя **Comparator**

Очереди Queue

Конструкторы PriorityQueue:

- **PriorityQueue();** // создает очередь с приоритетами начальной емкостью 11, размещающую элементы согласно естественному порядку сортировки (**Comparable**).
- **PriorityQueue(Collection<? extends E> c);**
- **PriorityQueue(int initialCapacity);**
- **PriorityQueue(int initialCapacity, Comparator<? super E> comparator);**
- **PriorityQueue(PriorityQueue<? extends E> c);**
- **PriorityQueue(SortedSet<? extends E> c);**

Очереди Queue. Example 11

```
package _java._se._06.queue;
import java.util.Collections;
import java.util.PriorityQueue;
public class PriorityQueueExample {
  public static void main(String[] args) {

    PriorityQueue<String> queue1
                = new PriorityQueue<String>();
    queue1.offer("Oklahoma");
    queue1.offer("Indiana");
    queue1.offer("Georgia");
    queue1.offer("Texas");
    System.out.println("Priority queue using Comparable:" );
    while (queue1.size() > 0) {
        System.out.print(queue1.remove() + " ");
    }
```

Очереди Queue. Example 11

```
    PriorityQueue<String> queue2
            = new PriorityQueue<String>(4,
            Collections.reverseOrder());
    queue2.offer("Oklahoma");
    queue2.offer("Indiana");
    queue2.offer("Georgia");
    queue2.offer("Texas");
    System.out.println("\nPriority queue using Comparator:" );
    while (queue2.size() > 0) {
        System.out.print(queue2.remove() + " ");
    }
  }
}
```

Очереди Queue. Example 11

Результат:

```
Priority queue using
Comparable:
Georgia Indiana Oklahoma Texas
Priority queue using
Comparator:
Texas Oklahoma Indiana Georgia
```

КАРТЫ ОТОБРАЖЕНИЙ MAP

Карты отображений Map

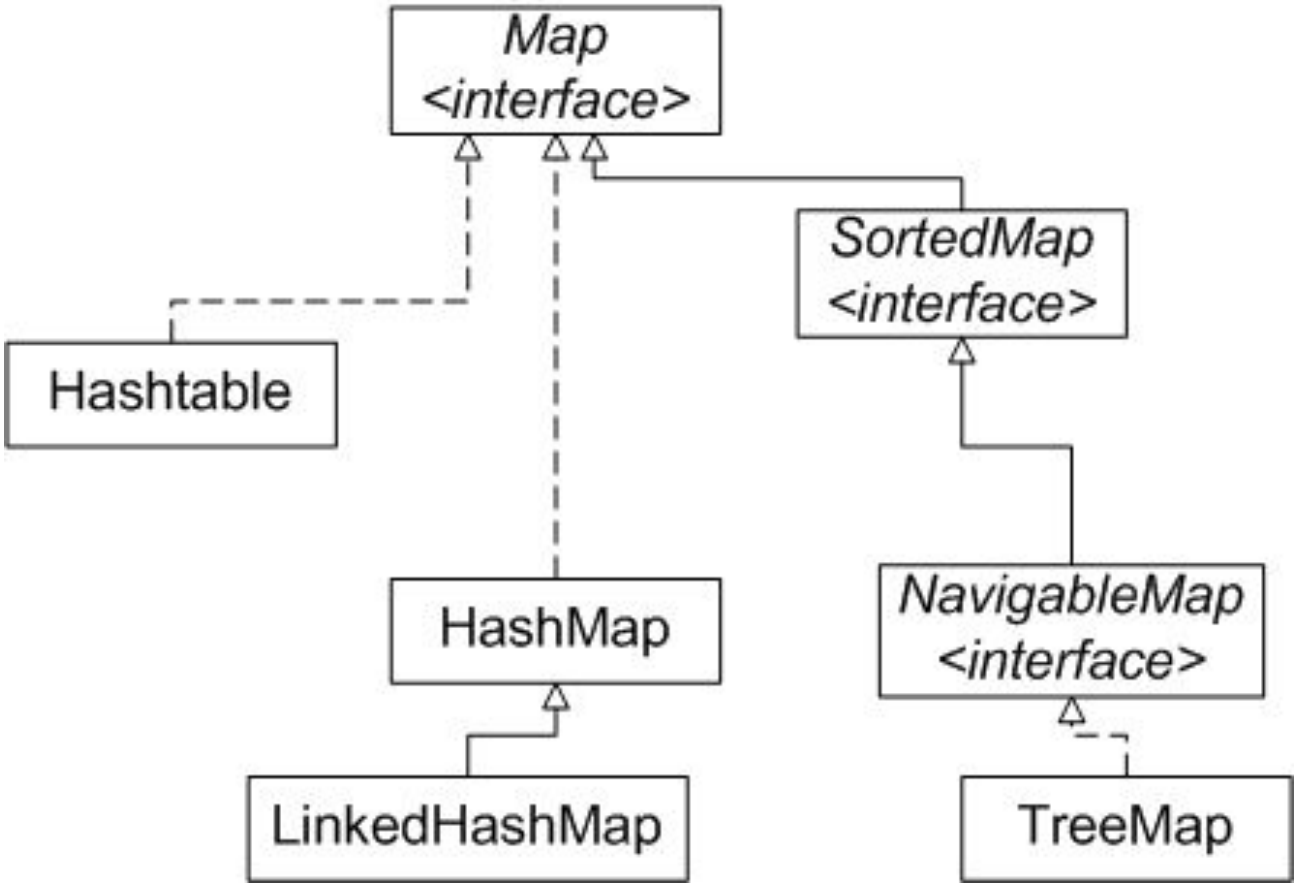
Интерфейс **Map** работает с наборами пар объектов «ключ-значение»

Все ключи в картах уникальны.

Уникальность ключей определяет реализация метода **equals**(…).

Для корректной работы с картами необходимо переопределить методы **equals**(…) и **hashCode**(), допускается добавление объектов без переопределения этих методов, но найти эти объекты в Map вы не сможете.

Карты отображений Map



Карты отображений Map

public interface Map<K,V> {

- **V put(K key, V value);** // запись
- **V get(Object key);** // получение значение
- **V remove(Object key);** // удаление
- **boolean containsKey(Object key);** // наличие ключа
- **boolean containsValue(Object value);** // наличие значения
- **int size();** // размер отображения
- **boolean isEmpty();** // проверка на пустоту
- **void putAll(Map<? extends K, ? extends V> m);** // добавление всех пар
- **void clear();** // полная очистка
- **public Set<K> keySet();** // множество ключей
- **public Collection<V> values();** // коллекция значений
- **public Set<Map.Entry<K,V>> entrySet();** // множество пар

Карты отображений Map

public static interface Map.Entry<K,V> {

- **boolean equals(Object o);** // сравнивает объект o с сущностью this на равенство
- **K getKey();** // возвращает ключ карты отображения
- **V getValue();** // возвращает значение карты отображения
- **int hashCode();** // возвращает hash-код для карты отображения
- **V setValue(V value);** // устанавливает значение для карты отображения

}

Карты отображений Map

public interface SortedMap<K,V> extends Map<K,V>{

- **Comparator<? super K> comparator();** // возвращает компаратор, используемый для упорядочивания ключей иди **null**, если используется естественный порядок сортировки
- **Set<Map.Entry<K,V>> entrySet();** // возвращает множество пар
- **K firstKey();** // минимальный ключ
- **SortedMap<K,V> headMap(K toKey);** // отображение ключей меньших **toKey**
- **Set<K> keySet();** // возвращает множество ключей
- **K lastKey();** // максимальный ключ
- **SortedMap<K,V> subMap(K fromKey, K toKey);** // отображение ключей меньших **toKey** и больше либо равных **fromKey**
- **SortedMap<K,V> tailMap(K fromKey);** // отображение ключей больших либо равных **fromKey**
- **Collection<V> values(); // возвращает коллекцию всех значений**

}

Карты отображений Map

public interface NavigableMap<K,V> extends SortedMap<K,V>{

// Методы данного интерфейса соответствуют методам NavigableSet, но позволяют, кроме того, получать как ключи карты отдельно, так и пары "ключ-значение"

- **Map.Entry<K,V> lowerEntry(K key);**
- **Map.Entry<K,V> floorEntry(K key);**
- **Map.Entry<K,V> higherEntry(K key);**
- **Map.Entry<K,V> ceilingEntry(K key);**
- **K lowerKey(K key);**
- **K floorKey(K key);**
- **K higherKey(K key);**
- **K ceilingKey(K key);**

// методы позво-ляют получить со-ответственно мень-ший, меньше или равный, больший, больше или рав-ный элемент по отношению к за-данному.

Карты отображений Map

// Методы pollFirstEntry и pollLastEntry возвращают соответственно первый и последний элементы карты, удаляя их из коллекции. Методы firstEntry и lastEntry также возвращают соответствующие элементы, но без удаления.

- **Map.Entry<K,V> pollFirstEntry();**
- **Map.Entry<K,V> pollLastEntry();**
- **Map.Entry<K,V> firstEntry();**
- **Map.Entry<K,V> lastEntry();**

// Метод descendingMap возвращает карту, отсортированную в обратном порядке:

- **NavigableMap<K,V> descendingMap();**

Карты отображений Map

// Методы, позволяющие получить набор ключей, отсортированных в прямом и обратном порядке соответственно:

- **NavigableSet navigableKeySet();**
- **NavigableSet descendingKeySet();**

Карты отображений Map

// Методы, позволяющие извлечь из карты подмножество. Параметры **fromKey** и **toKey** ограничивают подмножество снизу и сверху, а флаги **fromInclusive** и **toInclusive** показывают, нужно ли в результирующий набор включать граничные элементы. **headMap** возвращает элементы с начала набора до указанного элемента, а **tailMap** - от указанного элемента до конца набора. Перегруженные методы без логических параметров включают в выходной набор первый элемент интервала, но исключают последний.

- **NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive);**
- **NavigableMap<K,V> headMap(K toKey, boolean inclusive);**
- **NavigableMap<K,V> tailMap(K fromKey, boolean inclusive);**
- **SortedMap<K,V> subMap(K fromKey, K toKey);**
- **SortedMap<K,V> headMap(K toKey);**
- **SortedMap<K,V> tailMap(K fromKey);**

}

Карты отображений Map

HashMap – неотсортированная и неупорядоченная карта, эффективность работы **HashMap** зависит от того, насколько эффективно реализован метод **hashCode**().

HashMap может принимать в качестве ключа **null**, но такой ключ может быть только один, значений **null** может быть сколько угодно.

```
HashMap<String, String> hashMap =
        new HashMap<String, String>();

hashMap.put("key", "Value for key");
System.out.println(hashMap .get("key"));
```

Карты отображений Map

LinkedHashMap –хранит элементы в порядке вставки.

LinkedHashMap добавляет и удаляет объекты медленнее чем **HashMap**, но перебор элементов происходит быстрее.

Карты отображений Map

TreeMap –хранит элементы в порядке сортировки.

По умолчанию **TreeMap** сортирует элементы по возрастанию от первого к последнему, также порядок сортировки может задаваться реализацией интерфейсов **Comparator** и **Comparable**.

Реализация **Comparator** передается в конструктор **TreeMap**, **Comparable** используется при добавлении элемента в карту.

Карты отображений Map. Example 12

```
package _java._se._06.map;

import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.TreeMap;

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

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

        hashMap.put("Smith", 30);
        hashMap.put("Anderson", 31);
        hashMap.put("Lewis", 29);
        hashMap.put("Cook", 29);
        System.out.println("Display entries in HashMap");
        System.out.println(hashMap);
```

Карты отображений Map. Example 12

```
        Map<String, Integer> treeMap =  new TreeMap<String,
Integer>(hashMap);
        System.out.println("\nDisplay entries in ascending order
of key");
        System.out.println(treeMap);
        Map<String, Integer> linkedHashMap =  new
LinkedHashMap<String, Integer>(
                16, 0.75f, true);
        linkedHashMap.put("Smith", 30);
        linkedHashMap.put("Anderson", 31);
        linkedHashMap.put("Lewis", 29);
        linkedHashMap.put("Cook", 29);
        System.out.println("\nThe age for " + "Lewis is "
                + linkedHashMap.get("Lewis").intValue());
        System.out.println(linkedHashMap);
    }
}
```

Карты отображений Map. Example 12

Результат:

```
Display entries in HashMap
{Smith=30, Lewis=29, Anderson=31, Cook=29}

Display entries in ascending order of key
{Anderson=31, Cook=29, Lewis=29, Smith=30}

The age for Lewis is 29
{Smith=30, Anderson=31, Cook=29, Lewis=29}
```

Карты отображений Map. Example 13

```
package _java._se._06.map;
import java.util.Iterator;
import java.util.Map;
import java.util.Properties;
public class MapEntryExample {
    public static void main(String[] a) {
        Properties props = System.getProperties();
        Iterator iter = props.entrySet().iterator();
        while (iter.hasNext()) {
          Map.Entry entry = (Map.Entry) iter.next();
          System.out.println(entry.getKey() + " -- " +
entry.getValue());
        }
    }
}
```

Карты отображений Map. Example 13

Результат:

```
java.runtime.name -- Java(TM) SE Runtime Environment
sun.boot.library.path -- C:\Program Files\Java\jre6\bin
java.vm.version -- 20.2-b06
java.vm.vendor -- Sun Microsystems Inc.
java.vendor.url -- http://java.sun.com/
path.separator -- ;
java.vm.name -- Java HotSpot(TM) Client VM
file.encoding.pkg -- sun.io
sun.java.launcher -- SUN_STANDARD
user.country -- RU
sun.os.patch.level -- Dodatek Service Pack 3
java.vm.specification.name -- Java Virtual Machine
Specification
user.dir -- F:\ws\Java_SE_06
java.runtime.version -- 1.6.0_27-b07
java.awt.graphicsenv -- sun.awt.Win32GraphicsEnvironment
java.endorsed.dirs -- C:\Program Files\Java\jre6\lib\endorsed
os.arch -- x86
```

КЛАСС COLLECTIONS

Класс Collections

Collections — класс, состоящий из статических методов, осуществляющих различные служебные операции над коллекциями.

Методы Collections	Назначение
sort(List)	Сортировать список, используя merge sort алгоритм, с гарантированной скоростью O (n*log n).
binarySearch(List, Object)	Бинарный поиск элементов в списке.
reverse(List)	Изменить порядок элементов в списке на противоположный.
shuffle(List)	Случайно перемешать элементы.
fill(List, Object)	Заменить каждый элемент заданным.

Класс Collections

Методы Collections	Назначение
copy(List dest, List src)	Скопировать список src в dst.
min(Collection)	Вернуть минимальный элемент коллекции.
max(Collection)	Вернуть максимальный элемент коллекции.
rotate(List list, int distance)	Циклически повернуть список на указанное число элементов.
replaceAll(List list, Object oldVal, Object newVal)	Заменить все объекты на указанные.
indexOfSubList(List source, List target)	Вернуть индекс первого подсписка source, который эквивалентен target.
lastIndexOfSubList(List source, List target)	Вернуть индекс последнего подсписка source, который эквивалентен target.

Класс Collections

Методы Collections	Назначение
swap(List, int, int)	Заменить элементы в указанных позициях списка.
unmodifiableCollection (Collection)	Создает неизменяемую копию коллекции. Существуют отдельные методы для Set, List, Map, и т.д.
synchronizedCollection (Collection)	Создает потоко-безопасную копию коллекции. Существуют отдельные методы для Set, List, Map, и т.д.
checkedCollection(Collection<E> c, Class<E> type)	Создает типо-безопасную копию коллекции, предотвращая появление неразрешенных типов в коллекции. Существуют отдельные методы для Set, List, Map, и т.д.
<T> Set<T> singleton(T o);	Создает неизменяемый Set, содержащую только заданный объект. Существуют методы для List и Map.

Класс Collections

Методы Collections	Назначение
<T> List<T> nCopies(int n, T o)	Создает неизменяемый List, содержащий n копий заданного объекта.
frequency(Collection , Object)	Подсчитать количество элементов в коллекции.
reverseOrder()	Вернуть Comparator, которые предполагает обратный порядок сортировки элементов.
list(Enumeration<T> e)	Вернуть Enumeration в виде ArrayList.
disjoint(Collection, Collection)	Определить, что коллекции не содержат общих элементов.
addAll(Collection<? super T>, T[])	Добавить все элементы из массива в коллекцию
newSetFromMap(Map)	Создать Set из Map.
asLifoQueue(Deque)	Создать Last in first out Queut представление из Deque.

Класс Collections. Example 14

```
package _java._se._06.collections;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
public class CollectionsExample1 {
  public static void main(String[] args) {
    List<String> list1
        = Arrays.asList("red", "greean", "blue");
    Collections.sort(list1);
    System.out.println(list1);
    List<String> list2
          = Arrays.asList("greean", "red", "yellow", "blue");
    Collections.sort(list2, Collections.reverseOrder());
    System.out.println(list2);
  }
}
```

Результат:

```
[blue, greean, red]
[yellow, red, greean,
blue]
```

Класс Collections. Example 15

```
package _java._se._06.collections;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Random;
public class CollectionsExample2 {
  public static void main(String[] args) {
    List<String> list1
        = Arrays.asList("yellow", "red", "green", "blue");
    Collections.reverse(list1);
    System.out.println(list1);
    List<String> list2
        = Arrays.asList("yellow", "red", "green", "blue");
    Collections.shuffle(list2);
    System.out.println(list2);
  }
}
```

Результат:

```
[blue, green, red,
yellow]
[yellow, blue, green,
red]
```

Класс Collections. Example 16

```
package _java._se._06.collections;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Random;
public class CollectionsExample3 {
  public static void main(String[] args) {
    List<String> list3
        = Arrays.asList("yellow", "red", "green", "blue");
    List<String> list4
        = Arrays.asList("yellow", "red", "green", "blue");
    Collections.shuffle(list3, new Random(20));
    Collections.shuffle(list4, new Random(30));
    System.out.println(list3);
    System.out.println(list4);
  }
}
```

Результат:

```
[blue, yellow, red,
green]
[red, blue, yellow,
green]
```

Класс Collections. Example 17

```
package _java._se._06.collections;
import java.util.Arrays;
import java.util.Collections;
import java.util.GregorianCalendar;
import java.util.List;
public class CollectionsExample4 {
  public static void main(String[] args) {
    List<String> list1
        = Arrays.asList("yellow","red","green","blue");
    List<String> list2 = Arrays.asList("white","black");
    Collections.copy(list1, list2);
    System.out.println(list1);
    List<GregorianCalendar> list3
        = Collections.nCopies(5,
            new GregorianCalendar(2005,0,1));
  }
}
```

Результат:

```
[white, black, green,
blue]
```

Класс Collections. Example 18

```
package _java._se._06.collections;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
public class CollectionsExample5 {
  public static void main(String[] args) {
     List<Integer> list3 = Arrays
               .asList(2, 4, 7, 10, 11, 45, 50, 59, 60, 66);
     System.out.println("(1) Index: "
               + Collections.binarySearch(list3, 7));
     System.out.println("(2) Index: "
               + Collections.binarySearch(list3, 9));
     List<String> list4 = Arrays.asList("blue", "green", "red");
     System.out.println("(3) Index: „
               + Collections.binarySearch(list4, "red"));
     System.out.println("(4) Index: „
               + Collections.binarySearch(list4, "cyan"));
  }
}
```

Результат:

```
(1) Index: 2
(2) Index:
-4
(3) Index: 2
(4) Index:
-2
```

Класс Collections. Example 19

```
package _java._se._06.collections;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
public class CollectionsExample6 {
  public static void main(String[] args) {
    List<String> list = Arrays.asList("red", "green", "blue");
    Collections.fill(list, "black");
    System.out.println(list);
  }
}
```

Результат:

```
[black, black,
black]
```

Класс Collections. Example 20

```
package _java._se._06.collections;
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
public class CollectionsExample7 {
  public static void main(String[] args) {
    Collection<String> collection
        = Arrays.asList("red", "green", "blue");
    System.out.println(Collections.max(collection));
    System.out.println(Collections.min(collection));
  }
}
```

Результат:

```
[black, black,
black]
```

Класс Collections. Example 21

```
package _java._se._06.collections;
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
public class CollectionsExample8 {
  public static void main(String[] args) {
    Collection<String> collection1
        = Arrays.asList("red","cyan");
    Collection<String> collection2
        = Arrays.asList("red","blue");
    Collection<String> collection3
        = Arrays.asList("pink","tan");
    System.out.println(
        Collections.disjoint(collection1, collection2));
    System.out.println(
        Collections.disjoint(collection1, collection3));
  }
}
```

Результат:

```
false
true
```

Класс Collections. Example 22

```
package _java._se._06.collections;
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;
public class CollectionsExample9 {
  public static void main(String[] args) {
    Collection<String> collection
        = Arrays.asList("red","cyan","red");
    System.out.println(
        Collections.frequency(collection, "red"));
  }
}
```

Результат:

```
2
```

Класс Collections. Example 23

```
package _java._se._06.collections;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
public class CollectionsExample10 {
  public static void main(String[] args) {
     String init[]
          = { "One","Two","Three","One", "Two", "Three" };
     List list1 = new ArrayList(Arrays.asList(init));
     List list2 = new ArrayList(Arrays.asList(init));
     list1.remove("One");
     System.out.println(list1);
     list2.removeAll(Collections.singleton("One"));
     System.out.println(list2);
  }
}
```

Результат:

```
[Two, Three, One, Two,
Three]
[Two, Three, Two, Three]
```

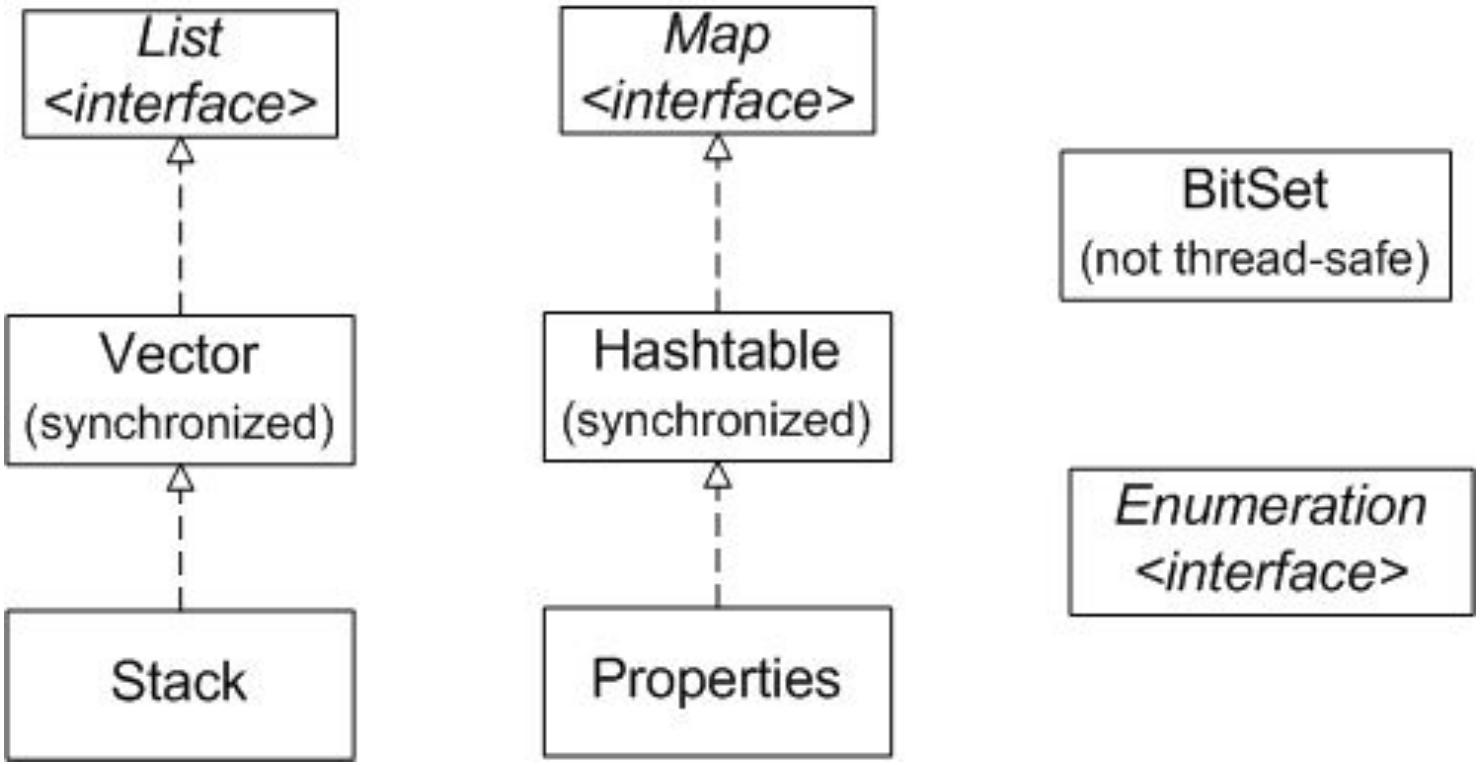
УНАСЛЕДОВАННЫЕ КОЛЛЕКЦИИ

Унаследованные коллекции

Унаследованные коллекции (**Legacy Collections**) – это коллекции языка Java 1.0/1.1

В ряде распределенных приложений, например с использованием сервлетов, до сих пор применяются унаследованные коллекции, более медленные в обработке, но при этом потокобезопасные, существовавшие в языке Java с момента его создания.

Унаследованные коллекции



Унаследованные коллекции

Vector –устаревшая версия ArrayList, его функциональность схожа с ArrayList за исключением того, что ключевые методы Vector синхронизированы для безопасной работы с многопоточностью. Из-за того что методы Vector синхронизированы, Vector работает медленее чем ArrayList.

Конструкторы класса Vector

- **Vector()**
- **Vector(Collection<? extends E> c).**
- **Vector(int initialCapacity)**
- **Vector(int initialCapacity, int capacityIncrement)**

Унаследованные коллекции. Example 24

```
package _java._se._06.legacy;
import java.util.Enumeration;
import java.util.Vector;
public class VectorExample {
    public static void main(String args[]) {
        // initial size is 3, increment is 2
        Vector v = new Vector(3, 2);
        System.out.println("Initial size: " + v.size());
        System.out.println("Initial capacity: "
                + v.capacity());
        v.addElement(new Integer(1));
        v.addElement(new Integer(2));
        v.addElement(new Integer(3));
        v.addElement(new Integer(4));
        System.out.println("Capacity after four additions: "
                + v.capacity());
        v.addElement(new Double(5.45));
        System.out.println("Current capacity: "
                + v.capacity());
```

Унаследованные коллекции. Example 24

```
v.addElement(new Double(6.08));
v.addElement(new Integer(7));
System.out.println("Current capacity: "
    + v.capacity());
v.addElement(new Float(9.4));
v.addElement(new Integer(10));
System.out.println("Current capacity: "
    + v.capacity());
v.addElement(new Integer(11));
v.addElement(new Integer(12));
System.out.println("First element: "
    + (Integer) v.firstElement());
System.out.println("Last element: "
    + (Integer) v.lastElement());
if (v.contains(new Integer(3)))
    System.out.println("Vector contains 3.");
```

Унаследованные коллекции. Example 24

```
        // enumerate the elements in the vector.
        Enumeration vEnum = v.elements();
        System.out.println("\nElements in vector:");
        while (vEnum.hasMoreElements())
         System.out.print(vEnum.nextElement() +  " ");
        System.out.println();
    }
}
```

Результат:

```
Initial size: 0
Initial capacity: 3
Capacity after four additions:
5
Current capacity: 5
Current capacity: 7
Current capacity: 9
First element: 1
Last element: 12
Vector contains 3.
Elements in vector:
1 2 3 4 5.45 6.08 7 9.4 10 11
12
```

Enumeration – объекты классов, реализующих данный интерфейс, используются для предоставления однопроходного последовательного доступа к серии объектов:

```
Hashtable<String, String> t = ...;

for (Enumeration<String> e = t.keys(); e.hasMoreElements();)
{
    String s = e.nextElement();
}
```

public interface Enumeration<E>{

- **boolean hasMoreElements();**
- **E nextElement() ;**

}

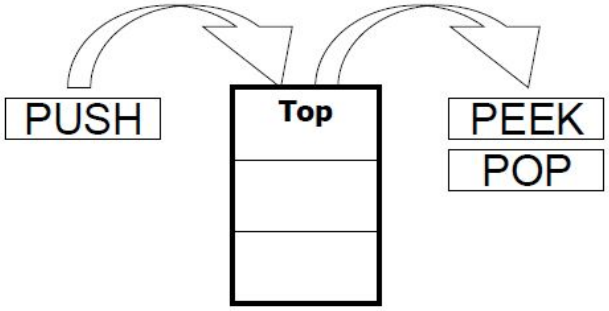
Унаследованные коллекции

Класс Stack позволяет создавать очередь типа last-in-first-out (LIFO)

public class Stack<E> extends Vector<E> {

- **public boolean empty();**
- **public synchronized E peek();**
- **public synchronized E pop();**
- **public E push(E object);**
- **public synchronized int search(Object o);**

}



Унаследованные коллекции. Example 25

```
package _java._se._06.legacy;
import java.util.Stack;
import java.util.StringTokenizer;
public class StackExample {
     static boolean checkParity(String expression,
     String open, String close) {
       Stack stack = new Stack();
       StringTokenizer st
                = new StringTokenizer(expression, " \t\n\r+*/-(){}", true);
       while (st.hasMoreTokens()) {
          String tmp = st.nextToken();
          if (tmp.equals(open))
               stack.push(open);
               if (tmp.equals(close))
               stack.pop();
       }
       if (stack.isEmpty()) return true;
       else return false;
     }
     public static void main(String[] args) {
          System.out.println(
          checkParity("a - (b - (c - a) / (b + c) - 2)", "(", ")"));
     }
}
```

Унаследованные коллекции

Hashtable – после модификации в JDK 1.2 реализует интерфейс **Map**. Порядок следования пар ключ/значение не определен.

Конструкторы Hashtable

- **Hashtable() ;**
- **Hashtable(int initialCapacity) ;**
- **Hashtable(int initialCapacity, float loadFactor) ;**
- **Hashtable(Map<? extends K,? extends V> t);**

Унаследованные коллекции. Example 26

```
package _java._se._06.legacy;

import java.util.Collection;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Iterator;

public class HashtableExample {
    public static void main(String[] args) {
        Hashtable<String, String> ht
            = new Hashtable<String, String>();
        ht.put("1", "One");
        ht.put("2", "Two");
        ht.put("3", "Three");
        Collection c = ht.values();
        Iterator itr = c.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
```

Унаследованные коллекции. Example 26

```
            c.remove("One");
            Enumeration e = ht.elements();
            while (e.hasMoreElements()) {
                System.out.println(e.nextElement());
            }
        }
}
```

Результат:

```
Three
Two
One
Three
Two
```

Унаследованные коллекции

Класс **Properties** предназначен для хранения набора свойств (параметров).

Методы

- **String getProperty(String key)**
- **String getProperty(String key,String defaultValue)**

позволяют получить свойство из набора.

С помощью метода

- **setProperty(String key, String value)**

это свойство можно установить.

Унаследованные коллекции

Метод

- **load(InputStream inStream)**

позволяет загрузить набор свойств из входного потока.

Параметры представляют собой строки представляющие сбой пары ключ/значение.

Предполагается, что по умолчанию используется кодировка ISO 8859-1.

Унаследованные коллекции. Example 27

```
package _java._se._06.legacy;

import java.util.Iterator;
import java.util.Properties;
import java.util.Set;

public class PropertiesExample {
    public static void main(String[] args) {
        Properties capitals = new Properties();
        Set states;
        String str;
        capitals.put("Illinois", "Springfield");
        capitals.put("Missouri", "Jefferson City");
        capitals.put("Washington", "Olympia");
        capitals.put("California", "Sacramento");
        capitals.put("Indiana", "Indianapolis");
```

Унаследованные коллекции. Example 27

```
        // Show all states and capitals in hashtable.
        states = capitals.keySet();  // get set-view of keys
        Iterator itr = states.iterator();
        while (itr.hasNext()) {
        str = (String) itr.next();
        System.out.println("The capital of "
              + str + " is "
           + capitals.getProperty(str) +  ".");
        }
        System.out.println();
        // look for state not in list  — specify default
        str = capitals.getProperty( "Florida", "Not Found");
        System.out.println("The capital of Florida is "
              + str + ".");
    }
}
```

Унаследованные коллекции. Example 27

Результат:

```
The capital of Missouri is Jefferson
City.
The capital of Illinois is Springfield.
The capital of Indiana is Indianapolis.
The capital of California is Sacramento.
The capital of Washington is Olympia.
The capital of Florida is Not Found.
```

Унаследованные коллекции

Класс **BitSet** предназначен для работы с последовательностями битов.

Каждый компонент. этой коллекции может принимать булево значение, которое обозначает установлен бит или нет.

Содержимое **BitSet** может быть модифицировано содержимым другого **BitSet** с использованием операций AND, OR или XOR (исключающее или).

BitSet имеет текущий размер (количество установленных битов) может динамически изменятся.

Унаследованные коллекции

По умолчанию все биты в наборе устанавливаются в 0 (**false**).

Установка и очистка битов в **BitSet** осуществляется методами **set(int index)** и **clear(int index)**.

Метод **int length()** возвращает "логический" размер набора битов, **int size()** возвращает количество памяти занимаемой битовой последовательностью **BitSet**.

Унаследованные коллекции. Example 28

```
package _java._se._06.legacy;
import java.util.BitSet;
public class BitSetExample {
     public static void main(String[] args) {
          BitSet bs1 = new BitSet();
          BitSet bs2 = new BitSet();
          bs1.set(0);
          bs1.set(2);
          bs1.set(4);
          System.out
               .println("Length = " + bs1.length()
                + " size = " + bs1.size());
          System.out.println(bs1);
          bs2.set(1);
          bs2.set(2);
          bs1.and(bs2);
          System.out.println(bs1);
     }
}
```

Результат:

```
Length = 5 size = 64
{0, 2, 4}
{2}
```

КОЛЛЕКЦИИ ДЛЯ ПЕРЕЧИСЛЕНИЙ

Коллекции для перечислений

Абстрактный класс **EnumSet<E extends Enum<E>>** (наследуется от абстрактного класса **AbstractSet) -** специально реализован для работы с типами **enum**.

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

Внутренне множество представимо в виде вектора битов, обычно единственного **long**.

Множества нумераторов поддерживают перебор по диапазону из нумераторов.

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

Коллекции для перечислений

Создание EnumSet

- **EnumSet<T> EnumSet.noneOf(T.class);** // создает пустое множество нумерованных констант с указанным типом элемента
- **EnumSet<T> EnumSet.allOf(T.class);** // создает множество нумерованных констант, содержащее все элементы указанного типа
- **EnumSet<T> EnumSet.of(e1, e2, …);** // создает множество, первоначально содержащее указанные элементы
- **EnumSet<T> EnumSet.copyOf(EnumSet<T> s);**
- **EnumSet<T> EnumSet.copyOf(Collection<T> t);**

Коллекции для перечислений

- **EnumSet<T> EnumSet.complementOf(EnumSet<T> s);** // создается множество, содержащее все элементы, которые отсутствуют в указанном множестве
- **EnumSet<T> range(T from, T to);** // создает множество из элементов, содержащихся в диапазоне, определенном двумя элементами

При передаче вышеуказанным методам в качестве параметра **null** будет сгенерирована исключительная ситуация **NullPointerException**

```
private enum PCounter {UNO, DOS, TRES, CUATRO, CINCO, SEIS, SIETE};
private Set<PCounter> es = null;
es = Collections.synchronizedSet(EnumSet.allOf(PCounter.class));
```

Коллекции для перечислений. Example 29

```
package _java._se._06.enum_;
import java.util.EnumSet;
enum Faculty {
      FFSM, MMF, FPMI, FMO, GEO
}
public class EnumSetExample {
  public static void main(String[] args) {
      /*
       * множество set1 содержит элементы типа enum из интервала,
       * определенного двумя элементами
       */
      EnumSet<Faculty> set1 = EnumSet.range(Faculty.MMF, Faculty.FMO);
      /*
       * множество set2 будет содержать все элементы, не содержащиеся _в
       * множестве set1
       */
      EnumSet<Faculty> set2 = EnumSet.complementOf(set1);
      System.out.println(set1);
      System.out.println(set2);
   }
}
```

Результат:

```
[MMF, FPMI, FMO]
[FFSM, GEO]
```

Коллекции для перечислений

EnumMap - высоко производительное отображение (map). В качестве ключей используются элементы перечисления, что позволяет реализовывать **EnumMap** на базе массива. **Null** ключи запрещены. **Null** значения допускаются. Не синхронизировано. Все основные операции с **EnumMap** совершаются за постоянное время. Как правило **EnumMap** работает быстрее, чем **HashMap**.

Создание EnumMap

- **EnumMap<K, V>(K.class);**
- **EnumMap<K, V>(EnumMap<K, V>);**
- **EnumMap<K, V>(Map<K, V>);**

Коллекции для перечислений

Создать объект **EnumMap**:

```
private EnumMap em = null;
private enum PCounter {UNO, DOS, TRES,
CUATRO};
em = new EnumMap(PCounter.class);
```

Создать синхронизированный объект **EnumMap**:

```
private Map em = null;
em = Collections.synchronizedMap( new
EnumMap(PCounter.class));
```

Коллекции для перечислений. Example 30

```
package _java._se._06.enum_;
import java.util.EnumMap;
enum Size {
    S, M, L, XL, XXL, XXXL;
}
public class EnumMapExample {
    public static void main(String[] args) {
        EnumMap<Size, String> sizeMap
            = new EnumMap<Size, String>(Size.class);
        sizeMap.put(Size.S, "S");
        sizeMap.put(Size.M, "M");
        sizeMap.put(Size.L, "L");
        sizeMap.put(Size.XL, "XL");
        sizeMap.put(Size.XXL, "XXL");
        sizeMap.put(Size.XXXL, "XXXL");
        for (Size size : Size.values()) {
            System.out.println(size
                + ":" + sizeMap.get(size));
        }
    }
}
```

Коллекции для перечислений

Результат:

```
S:S
M:M
L:L
XL:XL
XXL:XXL
XXXL:XXXL
```

СПАСИБО ЗА ВНИМАНИЕ!

ВОПРОСЫ?

Java.SE.06

Generic&Collections

Author: **Ihar Blinou, PhD**
Oracle Certified Java Instructor
Ihar_blinou@epam.com