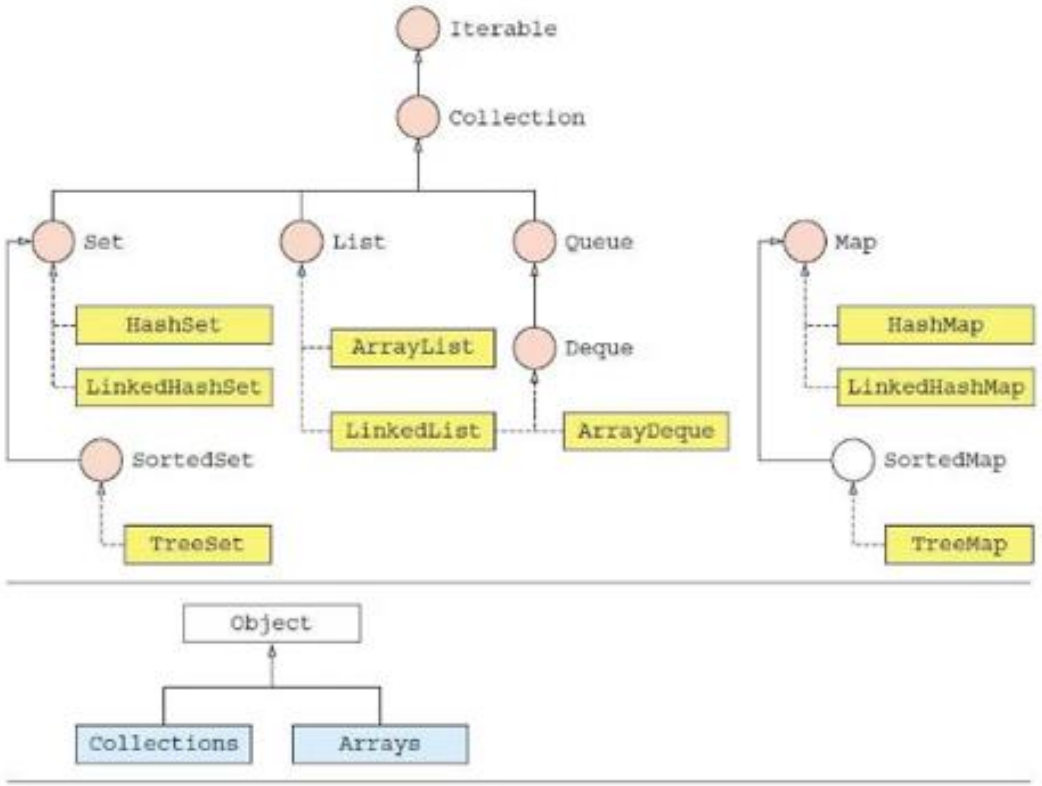
Коллекции в Java

Java 1

- Классы
  - Vector, Stack, Hashtable, BitSet
- Интерфейс
  - Enumeration

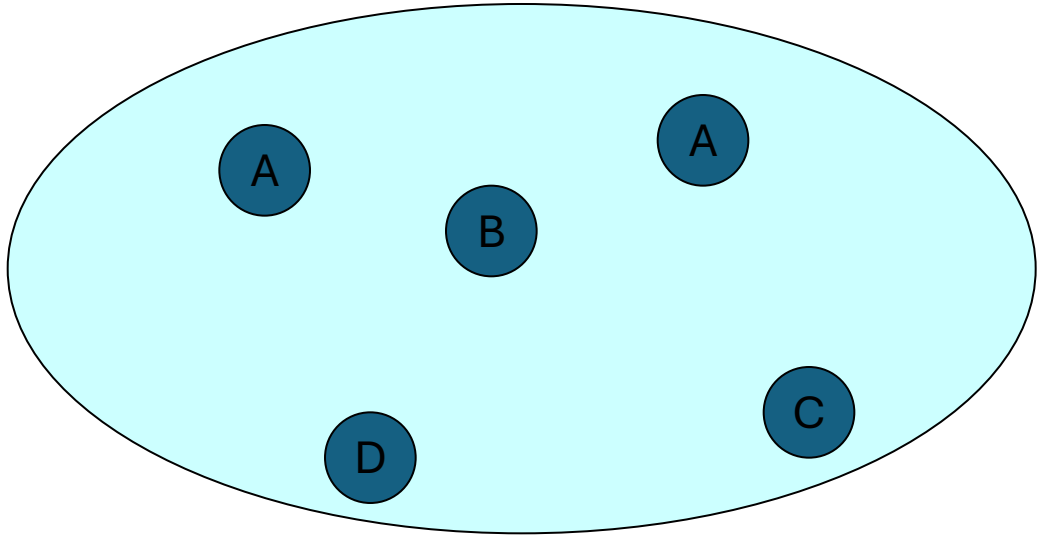
Начиная с Java 1.2

Базовые интерфейсы коллекций в пакете java.util.*



Коллекции

- Коллекция – неупорядоченный набор элементов
- Интерфейс Collection <E>



Интерфейс Collection<E>

Немодифицирующие операции

```
size()
isEmpty()
contains(Object o)
containsAll(Collection<?> c)
equals (Object o)
iterator()
toArray()
toArray(E[ ])
```

Модифицирующие операции

```
add(E e)
addAll(Collection <? Extends E> c)
remove(Object e)
removeAll (Collection <?> c)
retainAll (Collection <?> c)
clear()
```

✓ **Исключения**

- `UnsupportedOperationException`

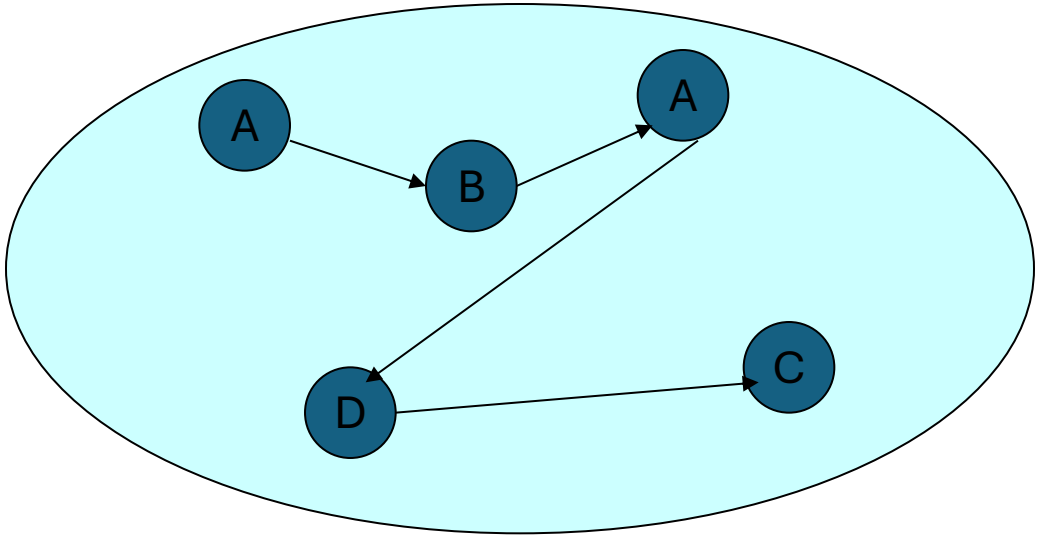
Перебор элементов коллекции

- Итератор
- Цикл for-each

Итераторы

- Итератор – для обхода коллекции
- Интерфейс Iterator <E>
- Метод коллекции

Iterator<E> iterator()



Интерфейс Iterator
java.util.Iterator<E>

`boolean hasNext()`	Возвращает `true`, если существует следующий элемент, к которому можно обратиться.
`E next()`	Возвращает следующий элемент. Генерируется исключение **`NoSuchElementException`**, если достигнут конец контейнера.
`void remove()`	Удаляет последний просмотренный элемент. Этот метод должен вызываться сразу после метода `next()`. Если этого не сделать генерируется исключение **`IllegalStateException.`** Если после чтения элемента набор данных изменился, данный метод генерирует исключение **`ConcurrentModificationException`**

Пример использования

```
import  java.util.*;
public class SimpleCollection {
  public static void main(String[ ] args) {
//Используем интерфейс, т.к. просто будем
// работать с особенностями Collection
Collection<String> c = new ArrayList<String>();
for(int i = 0; i < 10; i++)
    c.add(Integer.toString(i));
Iterator<String> it = c.iterator();
while(it.hasNext())
      System.out.println(it.next());
}
}
```

В JDK 5.0 цикл “for each”

```
for (String el : c){
 System.out.println(el);
}
```

Компилятор преобразует такой цикл в цикл с итератором

Особенности использования метода remove()

```
Iterator<String> it = c.iterator();
it.next(); //пропускаем первый элемент
it.remove(); //удаляем его
```

При вызове метода `next()` итератор “перескакивает” через следующий элемент и возвращает ссылку на него

Класс AbstractCollection<E>

- Абстрактный
- Позволяет быстро реализовывать коллекции
- Реализация неизменяемых коллекций (абстрактные методы – реализация обязательно)
  - `iterator()`
  - `size()`
- Реализация изменяемых коллекций - дополнительно
  - `add(Object o)`
  - `iterator.remove()`
- Остальные методы интерфейса Collection реализуются при необходимости

Интерфейсы и реализации

Interfaces	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue			PriorityQueue		
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Интерфейс List<E>

- Добавляет методы, использующие позицию элемента в коллекции (индекс)

```
E       get(int index)
void    add(int index, E element)
E       set(int index, E element)
boolean addAll(int index, Collection<? extends E> c)
E       remove(int index)
```

- Поиск возвращает позицию

```
int     indexOf(Object o)
int     lastIndexOf(Object o)
```

- Выделение подсписка

```
List<E> subList(int fromIndex, int toIndex)
```

Интерфейс List<E>

- Добавлены методы, возвращающие итератор с интерфейсом ListIterator<E>

```
ListIterator<E> listIterator()
ListIterator<E> listIterator(int index)
```

- ListIterator<E> позволяет навигацию в двух направлениях и изменение элемента по позиции итератора

ArrayList <E> и LinkedList<E>

- ArrayList <E>
  - имеет начальную емкость, которая может увеличиваться
  - добавление – в конец и по позиции
  - удаление по позиции, по значению и через итератор
- LinkedList <E>
  - добавление/удаление в конце и в начале
  - есть обратный итератор
  - есть операции, чтобы работать как со стеком/деком

Любой массив можно преобразовать в список

```
String arr[ ] =new String[size];
List<String> list = Arrays.asList(arr);
```

У коллекций есть обратное преобразование в массив

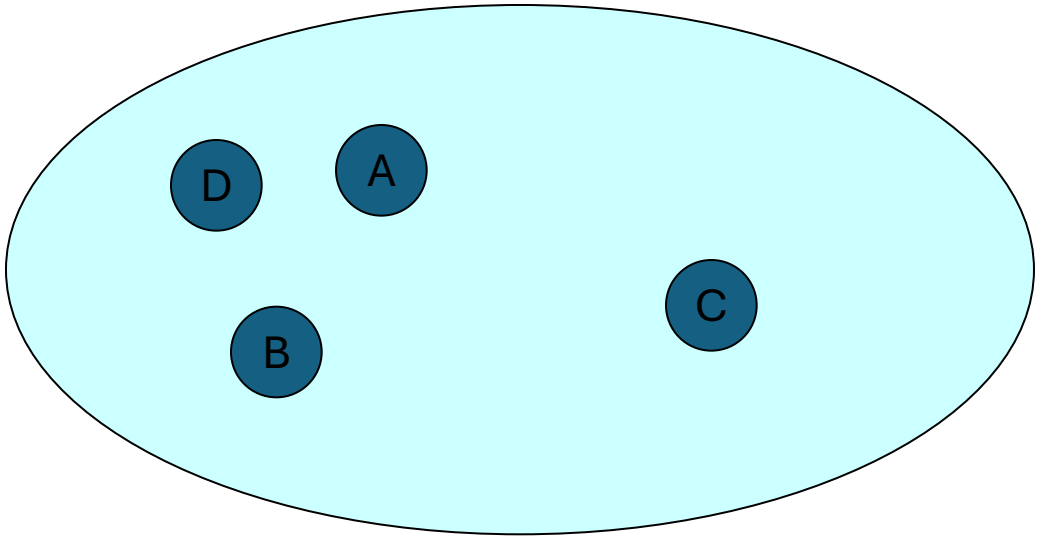
Множества

- Множество — коллекция без повторяющихся элементов
- Интерфейс `Set <E>`
- Реализация

`HashSet<E>`

`TreeSet<E>`

`LinkedHashSet<E>`



Сравнение элементов

Метод `Object.equals(Object object)`

- Рефлексивность `o1.equals(o1)==true`
- Симметричность `o1.equals(o2) == o2.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`

Интерпретация операций над множествами

Для элементов

- `add (E o)`
- `contains (Object o)`
- `remove (Object o)`

Для множеств

- `addAll(Collection<? extends E> c)` – объединение
- `retainAll(Collection <?> c)` – пересечение
- `containsAll(Collection <?> c)` – проверка вхождения
- `removeAll(Collection <?> c)` – разность

Классы HashSet и TreeSet

- `HashSet <E>`– множество на основе хэш-таблицы
- `TreeSet <E>` – отсортированное множество (на основе дерева)

HashSet

Метод `Object.hashCode()`

- Устойчивость

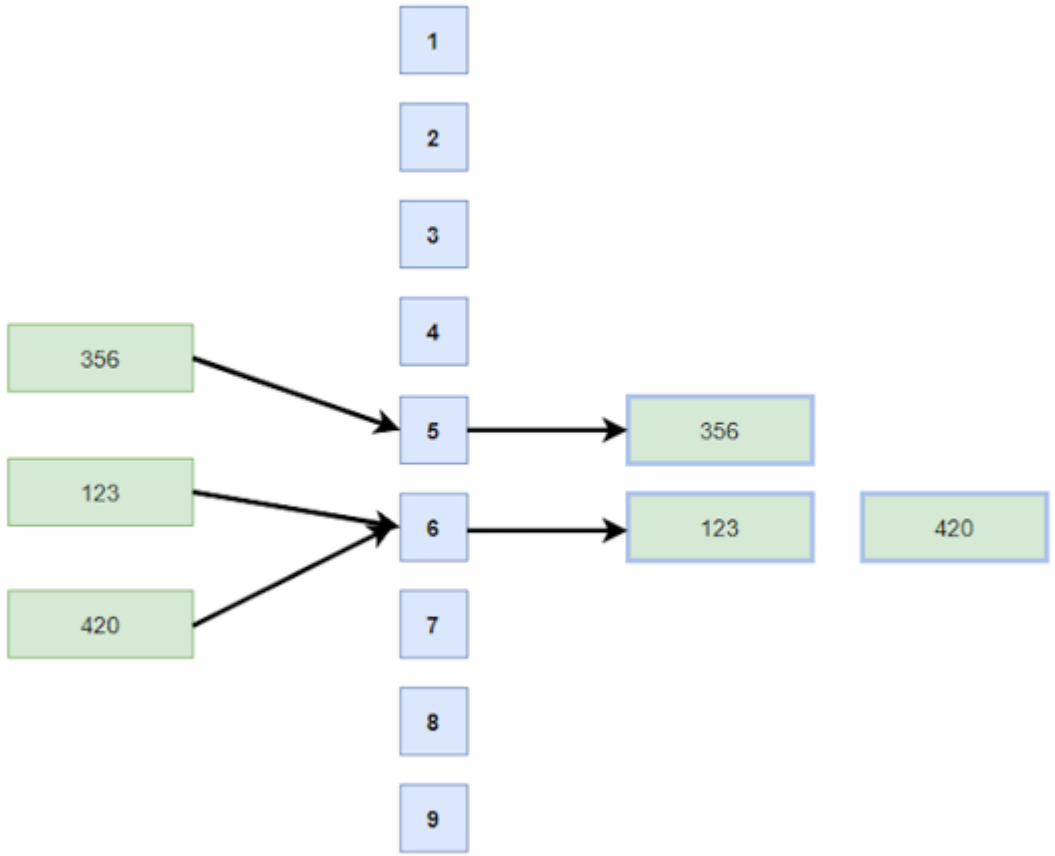
`hashCode()` не изменяется, если объект не изменяется

- Согласованность с `equals()`

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

- Быстрое предварительное сравнение объектов. Если хеш-коды разные, объекты точно разные. Если одинаковые — объекты могут быть разными (коллизия), и тогда вызывается equals()

Принцип организации хэш-таблиц



Пример использования

```
import java.util.*;
public class SetTest {
  public static void main(String [] args){
    Set<String> words = new HashSet<String>();
    Scanner in = new Scanner(System.in);
    while (in.hasNext()){
      String word = in.next();
      words.add(word);
    }
    System.out.println(”? ”+words.contains(”one”));
    Iterator <String> iter = words.iterator();
    while (iter.hasNext()){ //порядок определяется хэш-функцией
       System.out.println(iter.next()); }
  }
}
```

Сравним

```
HashSet<String> set =
                new HashSet<>();
        set.add("Иван");
        set.add("Марья");
        set.add("Пётр");
        set.add("Иван");

System.out.print(set.size()+":");
//ответ 3
```

```
class Person {
   String name;
. . .
}
 HashSet<Person> set =
                new HashSet<>();
 set.add(new Person("Иван“));
 set.add(new Person("Марья"));
 set.add(new Person("Пётр"));
 set.add(new Person("Иван"));

System.out.print(set.size()+":");
//ответ 4
```

Упорядоченные множества

- Нужно, чтобы класс, которому принадлежат объекты, составляющие `TreeSet`, реализовывал
  - интерфейс *`Comparable<E>`*
  - или интерфейс *`Comparator<E>`*

Интерфейс Comparable

`int compareTo(Object o)` – естественный порядок

0 – равны

«-» - меньше

«+» - больше

Пример использования

```
class Item implements Comparable<Item> {
  public int compareTo(Item other){
        return intNumber – other.intNumber;
  }
. . . . . .
  private int intNumber;
  private MyCl1 val;
  private double  s;
}
//-----------------------------------использование----------------
  ------------------
TreeSet <Item> tree = new TreeSet<Item>();
Item t = new Item(. . .);   tree.add(t);
. . . . .
for (Item el : tree) System.out.println(el);
```

Интерфейс Comparator

int compare(Object o1, Object o2) – сравнение элементов

Класс реализующий интерфейс Comparator не содержит полей –и имеет единственный мотод это *функциональный класс*

Объект такого класса можно передать в конструктор TreeSet

```
@FunctionalInterface
public interface Comparator<T> {

    int compare(T o1, T o2);

    // ...

}
```

Пример

```
class ItemComp implements Comparator<Item>{
public int compare (Item a,Item b) {
     MyCl1 v1 = a.getVal();
   MyCl1 v2 = b.getVal();
     return v1.compareTo(v2);
       }
}

//-------------------------использование-----------------------
ItemComp comp = new ItemComp();
TreeSet<Item> sortByVal = new TreeSet<>(comp);
```

Через безымянный внутренний класс

```
//----------------------------------------

TreeSet<Item> sortByVal = new TreeSet<>(
        new Comparator<Item>(){
                public int compare (Item a,Item b) {
                        return a.getVal().compareTo(b.getVal());
                 }
        }
);
```

Через лямбда-выражение

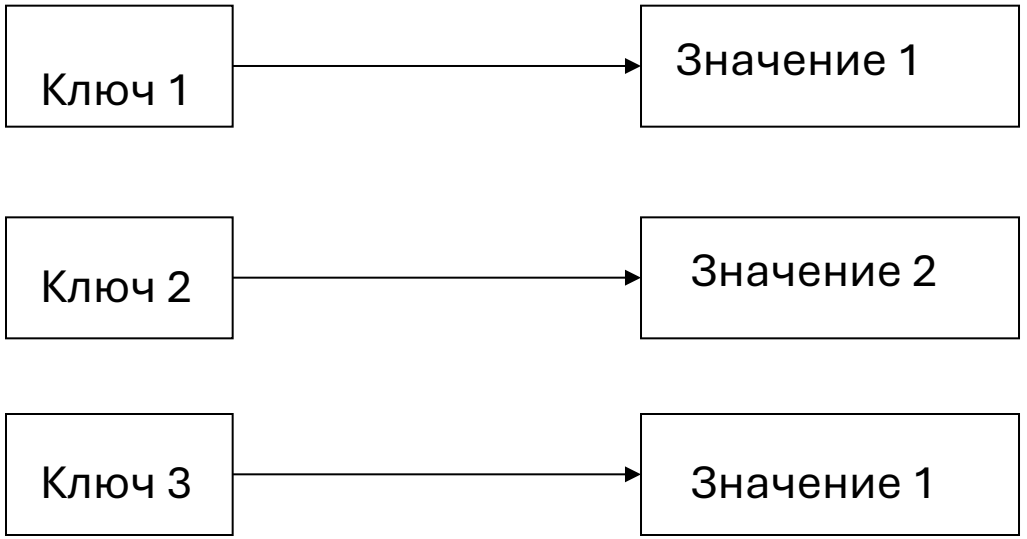
```
Comparator<Item> comp =
            (a,b) -> a.getVal().compareTo(b.getVal());

TreeSet<Item> sortByVal = new    TreeSet<>(comp);

Или

TreeSet<Item> sortByVal =
       new    TreeSet<>(
              (a,b) ->a.getVal().compareTo(b.getVal()));
```

Дополнительные операции

- `first()` – минимальный элемент
- `last()` – максимальный элемент
- `headSet(Object o)` – подмножество элементов меньших `o`
- `tailSet(Object o)` – подмножество элементов больших либо равных `o`
- `subSet(Object o1, Object o2)` – подмножество элементов меньших `o2` и больше либо равных `o1`

Отображения (карты)

- Отображение - множество пар ключ-значение при уникальности ключа
- Интерфейс *java.util.Map<K,V>*



Методы

- Доступ
  - `V get(K key)` - получение значения по ключу или null
  - `V put(K key, V value)` – запись ключа и значения, возвращает старое значение по ключу или null
  - `V remove(K key)` – удаление значения по ключу, возвращает удаленное значение или null

Методы

- Проверки
  - `boolean containsKey(Object k)` - наличие ключа
  - `boolean containsValue(Object v)` - наличие значения

Методы

- Формирование представлений
  - `Set <K> keySet()` - возвращает представление карты в виде множества всех ключей.
  - `Collection <V> values()` -возвращает представление карты в виде коллекции всех значений
  - `Set <Map.Entry<K,V>> entrySet()` -возвращает представление карты в виде множества объектов Map.Entry, т.е. пар “ключ – значение”

Из каждого представления можно удалять элементы. При этом ключи и соответствующие им значения удалятся из карты. Добавлять новые элементы нельзя

Пары

Внутренний класс `java.util.Map.Entry <K,V>`

- Методы

```
K getKey()
V getValue()
V setValue(V value)
```

Реализации карт

- Хеш-карта `HashMap<E>`
- Карта-дерево `TreeMap<E>`

Пример использования

```
Map<String, Abonent> sprav=
             new HashMap<String, Abonent>();

sprav.put(”+7(863)222-22-22”,new Abonent(”FIO”,
                                ”address”,”tariff”,...));

Abonent a=sprav.get(”+7(863)222-22-22”);
```

Пример использования

```
Set <String> tel=sprav.keySet();
for (String numb : tel){
  System.out.println(numb);
}
for (Map.Entry<String, Abonent>pair : sprav.entrySet())
{
 String tel = pair.getKey();
 Abonent ab = pair.getValue();
...
}
```

Алгоритмы

Класс `Collections`

- Алгоритмы для работы с коллекциями
  - Простые операции
  - Перемешивание
  - Сортировка
  - Двоичный поиск
  - Поиск минимума и максимума

Простые операции

- Заполнение списка указанным значением

```
void fill(List <? super T> l, T v)
```

- Переворачивание списка

```
void reverse(List <?> l)
```

- Копирование из списка в список

```
void copy(List <? super T> to, List <T> from)
```

Перемешивание и сортировка

```
void shuffle(List <?> e)
void shuffle(List <?> e, Random r)
void sort (List<T> e)
void sort (List <T> e, Comparator <?super T> c)
```

Двоичный поиск

```
int binarySearch(List< ? extends T> e, T key)
int binarySearch(List< ? extends T> e, T key, Comparator <? super T> c )
```

Устаревшие коллекции

Устаревшие коллекции являются синхронизированными

Vector <E>

Stack <E>

Dictionary <K,V>

Hashtable <K,V>

interface Enumeration <K>

Класс BitSet

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

Класс Properties

Класс `Properties` - предназначен для хранения множества свойств (параметров).

Это карты особого вида:

- ключ и значение являются строками
- карту можно сохранить в файле и загрузить из файла
- для используемых по умолчанию значений создается вторая карта

Методы

```
String getProperty(String key)
String getProperty(String key, String defaultValue)
void setProperty(String key, String value)
void load(InputStream in)
void store(OutputStream out, String header)
```