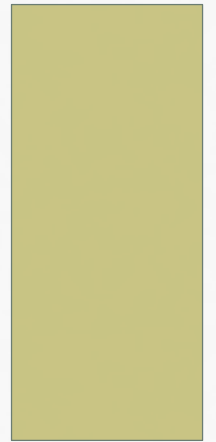


Collections Framework

Java

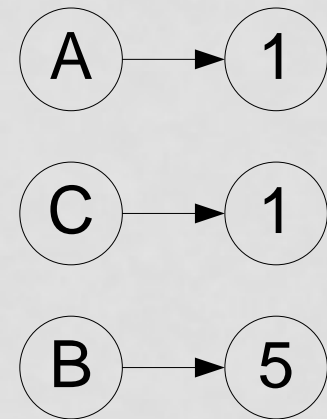
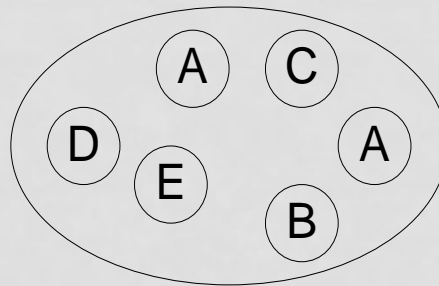


СОДЕРЖАНИЕ

1. Контейниер
 1. Коллекции
 2. Множества
 3. Списки
 4. Очереди и деки
2. Отображения
3. Упорядоченные коллекции
4. Алгоритмы
5. Заключение

COLLECTIONS FRAMEWORK

- Набор стандартных контейнеров (коллекций) и правил их использования
 - Интерфейсы
 - Реализации
 - Алгоритмы
- Пакет `java.util`



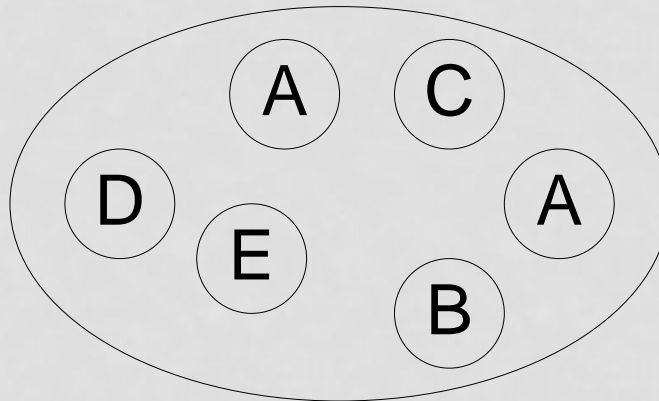
КОЛЛЕКЦИИ

ЧАСТЬ 1



КОЛЛЕКЦИИ

- Коллекция — неупорядоченный набор элементов
- Интерфейс `Collection<E>`
 - `<E>` — тип элемента



НЕМОДИФИЦИРУЮЩИЕ ОПЕРАЦИИ

- Определение размера
 - `size()` — количество элементов
 - `isEmpty()` — проверка на пустоту
- Проверки на вхождение
 - `contains(Object o)` — одного элемента
 - `containsAll(Collection<?> c)` — всех элементов `c`
- Почему `Collection<?>`

МОДИФИЦИРУЮЩИЕ ОПЕРАЦИИ

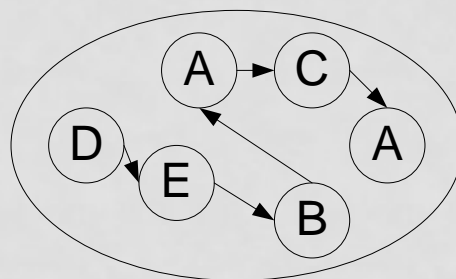
- Добавление элементов
 - `add(E e)` — одного элемента
 - `addAll(Collection<? extends E> c)` — элементов коллекции
- Удаление элементов
 - `remove(Object e)` — одного элемента
 - `removeAll(Collection<?> c)` — элементов коллекции
 - `retainAll(Collection<?> c)` — удаление элементов не из коллекции
 - `clear()` — удаление всех элементов
- Исключения
 - `UnsupportedOperationException`

ПРИМЕР. ЧТЕНИЕ В КОЛЛЕКЦИЮ

```
public int read(String file) throws IOException {  
    Scanner scanner = new Scanner(new File(file),  
    "Cp1251");  
  
    int read = 0;  
    while (scanner.hasNext()) {  
        read++;  
        c.add(scanner.next());  
    }  
  
    return read;  
}
```

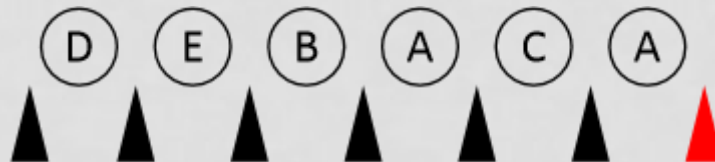

ИТЕРАТОРЫ

- Итератор — обход коллекции
- Интерфейс `Iterator<E>`
- Метод `Iterator<E> Collection<E>.iterator()`



МЕТОДЫ ИТЕРАТОРОВ

- `hasNext()` — определение наличия следующего элемента
- `next()` — взятие следующего элемента
- `remove()` — удаление элемента
- Исключения
 - `NoSuchElementException` — бросается при достижении конца коллекции
 - `ConcurrentModificationException` — бросается при изменении коллекции



ПРИМЕНЕНИЕ ИТЕРАТОРОВ

- Обход коллекции

```
for(Iterator<E> i = c.iterator(); i.hasNext(); ) {  
    final E element = i.next();  
    ...  
}
```

- Фильтрация коллекции

```
for(Iterator<E> i = c.iterator(); i.hasNext(); ) {  
    if (!p(i.next())) {  
        i.remove();  
    }  
}
```

Интерфейс Iterable

- Улучшенный for
 - Интерфейс `Iterable<T>`
 - Метод `iterator()`

- Код

```
for (T element : collection) {  
    ...  
}
```

- ЭКВИВАЛЕНТЕН

```
for (Iterator<T> i = collection.iterator(); i.hasNext(); ) {  
    T element = i.next();  
    ...  
}
```

ПРИМЕР. ВЫВОД КОЛЛЕКЦИИ НА ЭКРАН

```
public void dump(Collection<String> c) {  
    for (Iterator<String> i = c.iterator(); i.hasNext(); ) {  
        final String word = i.next();  
        System.out.print(word + ", ");  
    }  
    System.out.println();  
}
```

ПРЕОБРАЗОВАНИЕ В МАССИВ

- `Object[] toArray()` — создает новый массив
- `T[] toArray(T[] a)` — использует переданный массив
- Пример использования
`String[] i = c.toArray(new String[c.size()]);`
- В каком порядке идут элементы?
 - В порядке обхода итератором

КЛАСС AbstractCollection

- Позволяет быстро реализовывать коллекции
- Реализация неизменяемых коллекций
 - `iterator()`
 - `size()`
- Реализация изменяемых коллекций
 - `add(E o)`
 - `iterator.remove()`

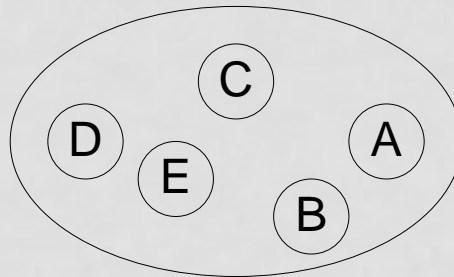
МНОЖЕСТВА

ЧАСТЬ 2



МНОЖЕСТВА

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



СРАВНЕНИЕ ЭЛЕМЕНТОВ

- Метод `Object.equals(Object object)`

Требования:

- Рефлексивность
`x.equals(x)`
- Симметричность
`x.equals(y) == y.equals(x)`
- Транзитивность
`x.equals(y) && y.equals(z) => x.equals(z)`
- Устойчивость
`o1.equals(o2)` не изменяется, если `o1` и `o2` не изменяются
- Обработка `null`
`o1.equals(null) == false`

Equals(1)

Точка на плоскости

```
public class Point {  
    protected int x;  
    protected int y;  
  
    public boolean equals(Object o) {  
        if (o instanceof Point) {  
            Point that = (Point) o;  
            return this.x == that.x && this.y == that.y;  
        }  
        return false;  
    }  
}
```

Equals(2)

Цветная Точка на плоскости

```
public class ColorPoint extends Point {
    protected int c;

    public boolean equals(Object o) {
        if (o instanceof ColorPoint) {
            ColorPoint that = (ColorPoint) o;
            return this.x == that.x && this.y == that.y &&
                this.c == that.c;
        }
        return false;
    }
}
```

Equals(2)

Цветная Точка на плоскости

```
public class ColorPoint extends Point {  
    protected int c;  
  
    public boolean equals(Object o) {  
        if (o instanceof ColorPoint) {  
            ColorPoint that = (ColorPoint) o;  
            return this.x == that.x && this.y == that.y &&  
                this.c == that.c;  
        }  
        return false;  
    }  
}
```

Проблемы: **несимметричность**

Equals(3)

Цветная Точка на плоскости

```
public class ColorPoint extends Point {
    protected int c;

    public boolean equals(Object o) {
        if (o instanceof ColorPoint) {
            ColorPoint that = (ColorPoint) o;
            return this.x == that.x && this.y == that.y &&
                this.c == that.c;
        }
        return super.equals(o);
    }
}
```

Equals(3)

Цветная Точка на плоскости

```
public class ColorPoint extends Point {
    protected int c;

    public boolean equals(Object o) {
        if (o instanceof ColorPoint) {
            ColorPoint that = (ColorPoint) o;
            return this.x == that.x && this.y == that.y &&
                this.c == that.c;
        }
        return super.equals(o);
    }
}
```

Проблемы: **нетранзитивность**

Equals

- Наследование и equals => проблемы
- Варианты решения
 - Сравнить объекты только одинакового класса
 - Сравнить как предков
 - Послойное сравнение
 - `canEquals`

Equals(4)

Point:

```
public boolean equals(Object other) {
    boolean result = false;
    if (other instanceof Point) {
        Point that = (Point) other;
        result = (this.getX() == that.getX() && this.getY() == that.getY() &&
this.getClass().equals(that.getClass()));
    }
    return result;
}
```

ColorPoint:

```
public boolean equals(Object other) {
    boolean result = false;
    if (other instanceof ColoredPoint) {
        ColoredPoint that = (ColoredPoint) other;
        result = (this.color.equals(that.color) && super.equals(that));
    }
    return result;
}
```

Equals(5)

Point:

```
public boolean equals(Object other) {
    boolean result = false;
    if (other instanceof Point) {
        Point that = (Point) other;
        result = (that.canEqual(this) && this.getX() == that.getX() && this.getY() == that.getY());
    }
    return result;
}
public boolean canEqual(Object other) {
    return (other instanceof Point);
}
```

ColorPoint:

```
public boolean equals(Object other) {
    boolean result = false;
    if (other instanceof ColoredPoint) {
        ColoredPoint that = (ColoredPoint) other;
        result = (that.canEqual(this) && this.color.equals(that.color) && super.equals(that));
    }
    return result;
}
public boolean canEqual(Object other) {
    return (other instanceof ColoredPoint);
}
```

ОПЕРАЦИИ НАД МНОЖЕСТВАМИ

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

КЛАССЫ HashSet И LinkedHashSet

- `HashSet<E>` — множество на основе хэша
- `LinkedHashSet<E>` — множество на основе хэша с сохранение порядка обхода

Конструкторы

- `(Linked)HashSet<E>()` — пустое множество
- `(Linked) HashSet<E>(Collection<?> c)` — элементы коллекции
- `(Linked) HashSet<E>(int initialCapacity[, double loadFactor])` — начальная вместимость и степень заполнения

ВЫЧИСЛЕНИЕ ХЭШЕЙ

- Метод `Object.hashCode()`
- Устойчивость
`hashCode()` не изменяется, если объект не изменяется
- Согласованность с `equals`
`o1.equals(o2) => o1.hashCode() == o2.hashCode()`

КЛАСС AbstractSet

- Позволяет быстро реализовывать множества
- Неизменяемые множества
 - `iterator()`
 - `size()`
- Изменяемые множества
 - `add(E o)`
 - `iterator.remove()`

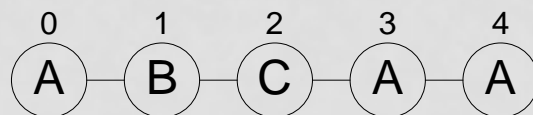
СПИСКИ

ЧАСТЬ 3



СПИСКИ

- Список — коллекция с индексированными элементами
- Интерфейс `List<E>` extends `Collection<E>`



ОПЕРАЦИИ СО СПИСКАМИ

- Доступ по индексу
 - `get(int i)` — чтение
 - `set(int l, E e)` — запись
 - `add(int i, E e)` — добавление
 - `remove(int i)` — удаление
- Поиск элементов
 - `indexOf(Object e)` — поиск с начала
 - `lastIndexOf(Object e)` — поиск с конца
- Взятие вида
 - `List<E> subList(int from, int to)`

ПОДСПИСКИ

Метод `List<E> subList(int from, int to)`

- `sl = l.subList(1, 4)`

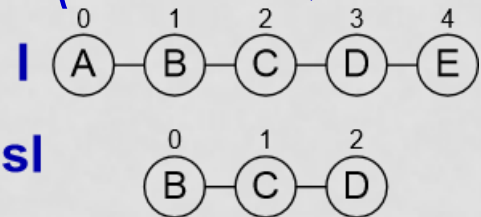
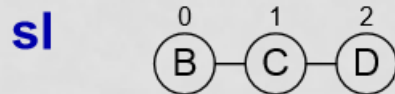
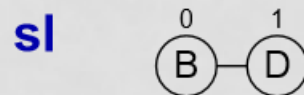
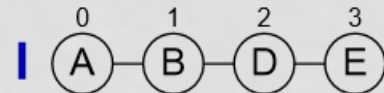


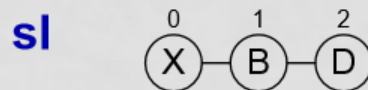
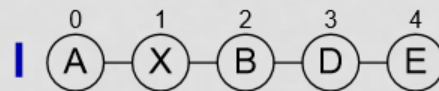
Diagram illustrating the result of `sl = l.subList(1, 4)`. List `l` contains elements A, B, C, D, E at indices 0, 1, 2, 3, 4. The sub-list `sl` contains elements B, C, D at indices 0, 1, 2.



- `l.delete(2)` либо `sl.delete(1)`

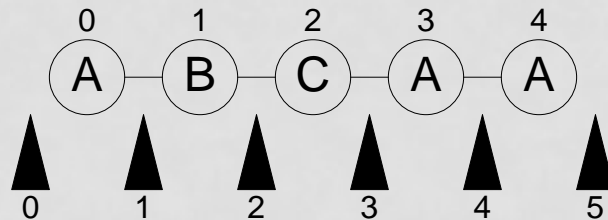


- `l.add(1, X)` либо `sl.add(0, X)`



ИТЕРАТОР ПО СПИСКУ

- Интерфейс `ListIterator<E>` extends `Iterator<E>`
- Метод `listIterator()`
- Предыдущий / Следующий элементы



ОПЕРАЦИИ ИТЕРАТОРА ПО СПИСКУ

- Передвижение
 - `hasNext()` / `hasPrevious()` — проверка
 - `next()` / `previous()` — взятие элемента
 - `nextIndex()` / `previousIndex()` — определение индекса
- Изменение
 - `remove()` — удаление элемента
 - `set(E e)` — изменение элемента
 - `add(E e)` — добавление элемента

КЛАСС ArrayList

- `ArrayList<E>` — список на базе массива
- Плюсы
 - Быстрый доступ по индексу
 - Быстрая вставка и удаление элементов с конца
- Минусы
 - Медленная вставка и удаление элементов
- Вместимость — реальное количество элементов
 - `ensureCapacity(int c)` — определение вместимости
 - `trimToSize()` — “подгонка” вместимости
- Конструкторы
 - `ArrayList(Collection<? Extends E> c)` — копия коллекции
 - `ArrayList([int initialCapacity])` — пустой список заданной вместимости

ПРИМЕР. ВЫВОД ARRAYLIST НА ЭКРАН

```
List<E> list = new ArrayList<>();
```

```
...
```

```
for (int i = list.size() - 1; i >= 0; i--) {  
    System.out.println(list.get(i));  
}
```

КЛАСС LinkedList

- `LinkedList<E>` — двусвязный список
- Плюсы
 - Быстрое добавление и удаление элементов
- Минусы
 - Медленный доступ по индексу
- Конструкторы
 - `LinkedList<E>()` — пустой список
 - `LinkedList<E>(Collection<?> c)` — копия коллекции
- Методы
 - `addFirst(E o)` — добавить в начало списка
 - `addLast(E o)` — добавить в конец списка
 - `removeFirst()` — удалить первый элемент
 - `removeLast()` — удалить последний элемент

ПРИМЕР. ВЫВОД LINKEDLIST НА ЭКРАН

```
List<E> list = new LinkedList<>();
```

```
...
```

```
for (ListIterator li = list.listIterator(list.size());
```

```
    li.hasPrevious(); )
```

```
{
```

```
    System.out.println(li.previous());
```

```
}
```


КЛАСС AbstractList

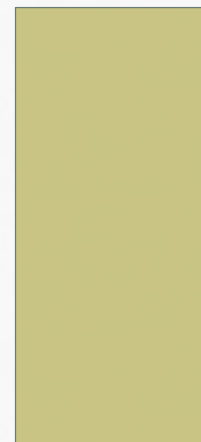
- Позволяет быстро реализовывать списки с произвольным доступом
- Неизменяемые списки
 - `get(index)`
 - `size()`
- Изменяемые списки
 - `set(index, element)`
- Списки переменной длины
 - `add(index, element)`
 - `remove(index)`

КЛАСС `AbstractSequentialList`

- Позволяет быстро реализовывать списки с последовательным доступом
- Неизменяемые списки
 - `listIterator()` (методы перемещения)
 - `size()`
- Изменяемые списки
 - `ListIterator.set(index, element)`
- Списки переменной длины
 - `ListIterator.add(element)`
 - `ListIterator.remove(element)`

ОЧЕРЕДИ И ДЕКИ

ЧАСТЬ 4



ОЧЕРЕДЬ

- Очередь – хранилище элементов для обработки
- Интерфейс `Queue<E> extends Collection<E>`
- Свойства очередей
 - Порядок выдачи элементов определяется конкретной реализацией
 - Очереди не могут хранить `null`
 - У очереди может быть ограничен размер
 - Могут не принять элемент

МЕТОДЫ ОЧЕРЕДЕЙ

- Обычные методы
 - `add(E o)` – добавить элемент
 - Бросает `IllegalStateException`
 - `E element()` – вершина очереди
 - Бросает `NoSuchElementException`
 - `E remove()` – удалить элемент из вершины
 - Бросает `NoSuchElementException`
- Методы, не бросающие исключений
 - `offer(E o)` – добавить элемент
 - `E peek()` – вершина очереди
 - `E poll()` – удалить элемент из вершины

КЛАСС LinkedList

- Очередь на двусвязном списке

КЛАСС AbstractQueue

- Позволяет быстро реализовывать очереди
- Методы
 - `size()`
 - `offer(E o)`
 - `peek()`
 - `poll()`
 - `iterator()`
- Исключение не дешевая операция!

ДЕКИ

- Интерфейс `Deque`
- Класс `ArrayDeque` – циклическая очередь
- Класс `LinkedList` – двусвязный список

Действие	Голова		Хвост	
	Исключение	Код возврата	Исключение	Код возврата
Вставка	<code>addFirst</code>	<code>offerFirst</code>	<code>addLast</code>	<code>offerLast</code>
Доступ	<code>getFirst</code>	<code>peekFirst</code>	<code>getLast</code>	<code>peekLast</code>
Удаление	<code>removeFirst</code>	<code>pollFirst</code>	<code>removeLast</code>	<code>pollLast</code>

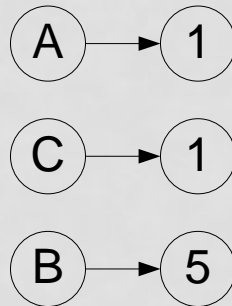
ОТОБРАЖЕНИЯ

ЧАСТЬ 5



ОТОБРАЖЕНИЕ

- Отображение — множество пар ключ-значение при уникальности ключа
- Интерфейс `Map<K, V>`



МЕТОДЫ ОТОБРАЖЕНИЙ (1)

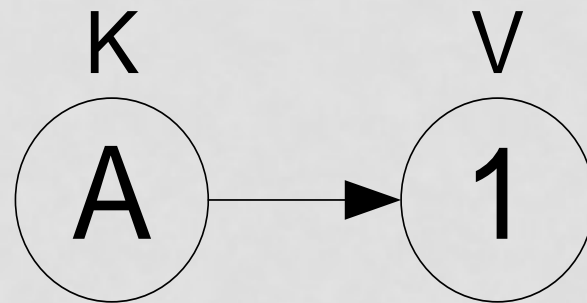
- Доступ
 - `get(K)` — получение значение
 - `put(K, V)` — запись
 - `remove(K)` — удаление
- Проверки
 - `containsKey(K)` — наличие ключа
 - `containsValue(V)` — наличие значения
- Определения размера
 - `size()` — размер отображения
 - `isEmpty()` — проверка на пустоту

МЕТОДЫ ОТОБРАЖЕНИЙ (2)

- Взятие видов
 - `Set<Map.Entry<K, V>> entrySet()` — множество пар
 - `Collection<V> values()` — коллекция значений
 - `Set<K> keySet()` — множество ключей
- Массовые операции
 - `putAll(Map<? extends K, ? extends V> map)` — добавление всех пар

ПАРЫ

- Пара — ключ + значение
- Интерфейс `Map.Entry<K, V>`
- Методы
 - `K getKey()`
 - `V getValue()`
 - `setValue(V)`



КЛАССЫ HashMap И LinkedHashMap

- `HashMap<K, V>` — отображение на основе хэшей
- `LinkedHashMap<K, V>` — отображение на основе хэшей с сохранением порядка обхода

Конструкторы

- `HashMap<K, V>()` — пустое отображение
- `HashMap<K, V>(Map<? extends K, ? extends V> m)` — копия отображения
- `HashMap(int initialCapacity)` — начальная вместимость
- `HashMap(int initialCapacity[, int loadFactor])` — начальная вместимость и степень заполнения

КЛАСС AbstractMap

- Позволяет быстро реализовывать множества
- Метод
 - `entrySet()`

ПРИМЕР. ПОДСЧЕТ СЛОВ В ТЕКСТЕ (1)

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

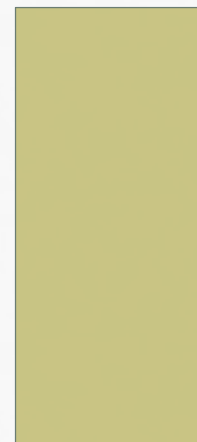
```
while (scanner.hasNext()) {  
    String word = scanner.next();  
    Integer count = map.get(word);  
    int value = (count == null)  
        ? 0  
        : count.intValue();  
    map.put(word, new Integer(value + 1));  
}
```


ПРИМЕР. ПОДСЧЕТ СЛОВ В ТЕКСТЕ (2)

```
for (  
    Iterator<String, Integer> i = map.entrySet().iterator();  
    i.hasNext();  
) {  
    Map.Entry<String, Integer> entry = i.next();  
    System.out.println(  
        entry.getKey() + " " + entry.getValue());  
}
```

УПОРЯДОЧЕННЫЕ КОЛЛЕКЦИИ

ЧАСТЬ 6



СРАВНЕНИЕ ЭЛЕМЕНТОВ (КОНТРАКТ)

- Транзитивность
- Антисимметричность
 $\text{sgn}(o1.compareTo(o2)) == -\text{sgn}(o2.compareTo(o1))$
- Согласованность с равенством
 $o1.compareTo(o2) == 0 \Rightarrow$
 $\text{sgn}(o1.compareTo(o3)) == \text{sgn}(o2.compareTo(o3))$
- Согласованность с `equals()`
 $o1.equals(o2) == (o1.compareTo(o2) == 0)$

УПОРЯДОЧЕННЫЕ МНОЖЕСТВА (1)

- Интерфейс `SortedSet<E>`
 - `first()` – минимальный элемент
 - `last()` – максимальный элемент
 - `headSet(E o)` – подмножество элементов меньших `o`
 - `tailSet(E o)` – подмножество элементов больших либо равных `o`
 - `subSet(E o1, E o2)` – подмножество элементов меньших `o2` и больше либо равных `o1`
- Класс `TreeSet<E>`

УПОРЯДОЧЕННЫЕ МНОЖЕСТВА (2)

- Интерфейс `NavigableSet<E>`
 - `pollLast()` – максимальный элемент
 - `lower(o)` – максимальный элемент $<$ данного
 - `floor(o)` – максимальный элемент \leq данного
 - `pollFirst()` – минимальный элемент
 - `higher(o)` – минимальный элемент $>$ данного
 - `ceiling(o)` – минимальный элемент \geq данного
 - `descendingSet()` – вид с обратным порядком
- Класс `TreeSet<e>`

УПОРЯДОЧЕННЫЕ ОТОБРАЖЕНИЯ (1)

- Интерфейс `SortedMap<K, V>`
 - `firstKey()` – минимальный ключ
 - `lastKey()` – максимальный ключ
 - `headMap(K)` – отображение ключей меньших `o`
 - `tailMap(K)` – отображение ключей больших либо равных `o`
 - `subMap(K k1, K k2)` – отображение ключей меньших `k2` и больше либо равных `k1`
- Класс `TreeMap<K, V>`

УПОРЯДОЧЕННЫЕ ОТОБРАЖЕНИЯ (2)

- Интерфейс `NavigableMap<K, V>`
 - `{pollLast | lower | floor | first | higher | ceiling}Key` – поиск ключа
 - `{pollLast | lower | floor | first | higher | ceiling}Entry` – поиск пары
 - `descendingMap()` – вид с обратным порядком
- Класс `TreeMap<K, V>`

КЛАСС PriorityQueue

- Очередь с приоритетами
- Реализована на основе двоичной кучи

ПРИМЕР. ПРИМЕНЕНИЕ TreeSet

- Естественный порядок

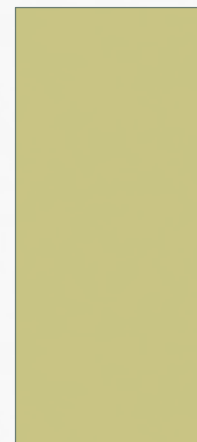
```
SortedSet<String> words = new TreeSet<String>();  
read(args[0], words);  
dump(words);
```

- Порядок без учета регистра

```
SortedSet<String> words = new  
    TreeSet<String>(String.CASE_INSENSITIVE_ORDER);  
read(args[0], words);  
c.dump();
```

АЛГОРИТМЫ

ЧАСТЬ 7



КЛАСС Collections

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

ПРОСТЫЕ ОПЕРАЦИИ

- Заполнение списка указанным значением
 - `fill(List<E>, E)`
- Переворачивание списка
 - `reverse(List<?> l)`
- Копирование из списка в список
 - `copy(List<? super E> to, List<? extends E> from)`
- Перемешивание
 - Генерирует случайную перестановку
 - `shuffle(List l)`
 - `shuffle(List l, Random r)`

СОРТИРОВКИ

- Устойчивая сортировка
- Алгоритм – Merge Sort
- Методы
 - `sort(List<?> l)` – сортировка списка (естественный порядок)
 - `sort(List<E> l, Comparator<? super E> c)` – сортировка списка (указанный порядок)

ДВОИЧНЫЙ ПОИСК

- Осуществляет двоичный поиск в списке
 - Найден – индекс элемента
 - Не найден – -1 – индекс места вставки
- Методы
 - `binarySearch(List<E> l, E o)` – ищет `o` в списке
 - `binarySearch(List<E> l, E o, Comparator<? super E> c)` – ищет `o` в списке

ПОИСК МИНИМУМА И МАКСИМУМА

- Поиск минимума
 - `min(Collection<E> c)` – минимальный элемент (естественный порядок)
 - `min(Collection<E> c, Comparator<? super E> cmp)` – минимальный элемент (указанный порядок)
- Поиск максимума
 - `max(Collection c)` – максимальный элемент (естественный порядок)
 - `max(Collection c, Comparator cmp)` – максимальный элемент (указанный порядок)

СПЕЦИАЛЬНЫЕ КОЛЛЕКЦИИ

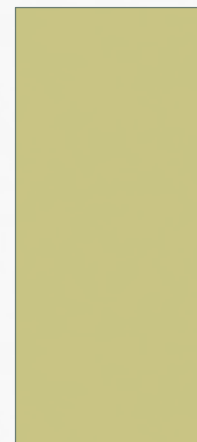
- Пустые коллекции
 - `emptySet()` – пустое множество
 - `emptyList()` – пустой список
 - `emptyMap()` – пустое отображение
- Коллекции из одного элемента
 - `singleton(E)` – множество
 - `singletonList(E)` – список
 - `singletonMap(K, V)` – отображение

ОБОЛОЧКИ КОЛЛЕКЦИЙ

- **Неизменяемые виды на коллекции**
 - `unmodifiableSet(Set<E> s)` – неизменяемое множество
 - `unmodifiableSortedSet(SortedSet<E> s)` – неизменяемое упорядоченное множество
 - `unmodifiableList(List<E> l)` – неизменяемый список
 - `unmodifiableMap(Map<E> m)` – неизменяемое отображение
 - `unmodifiableSortedMap(SortedMap<E> m)` – неизменяемое упорядоченное отображение

ЗАКЛЮЧЕНИЕ

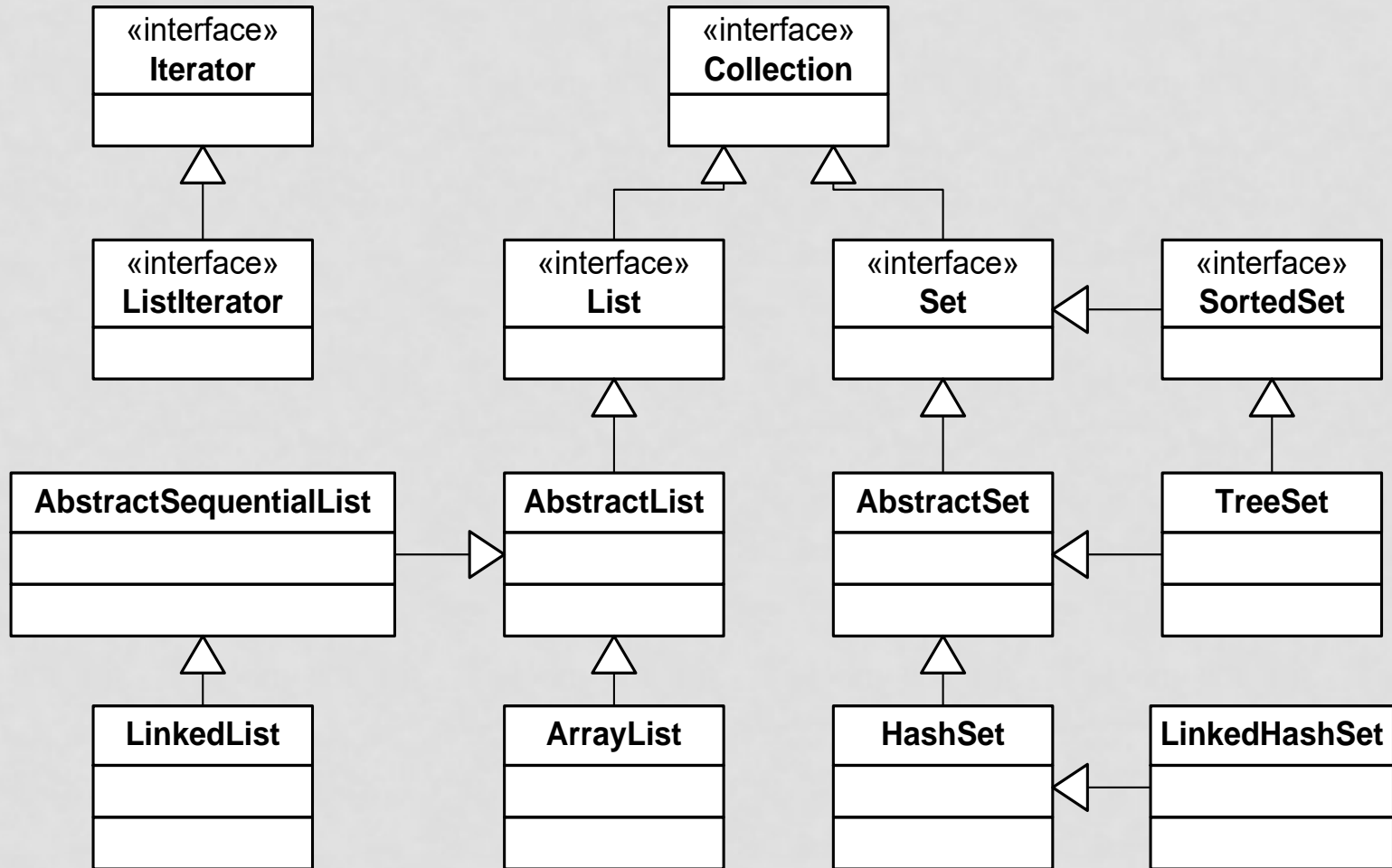
ЧАСТЬ 9



УСТАРЕВШИЕ КОЛЛЕКЦИИ

- Устаревшие коллекции являются синхронизированными
- Vector (ArrayList)
 - Stack (ArrayList)
- Dictionary (Map)
 - Hashtable (HashMap)
- Enumeration (Iterator)

СТРУКТУРА COLLECTIONS FRAMEWORK (1)



СТРУКТУРА COLLECTIONS FRAMEWORK (2)

