**КЛАССЫ-КОЛЛЕКЦИИ**

При решении практических задач, в которых количество элементов заранее неизвестно, элементы надо часто удалять и добавлять, необходим быстрый поиск данных, можно использовать библиотеки классов-коллекций.

***Класс Vector.*** В языке Java с самых первых версий был разработан класс Vector, предназначенный для хранения переменного числа элементов. В классе Vector из пакета java.util хранятся элементы типа Object, а значит, любого типа. Количество элементов может быть любым и наперед не определяться. Элементы получают индексы 0, 1, 2, ....

Кроме количества элементов, называемого размером(size) вектора, есть еще размер буфера — емкость(capacity) вектора. Обычно емкость совпадает с размером вектора, но можно ее увеличить методом **ensureCapacity(int minCapacity)** или сравнять с размером вектора методом **trimToSize().**

В Java 2 класс Vector был переработан, чтобы включить его в иерархию классов-коллекций. Поэтому многие действия можно совершать старыми и новыми методами. Рекомендуется использовать новые методы.

В классе четыре конструктора:

**Vector** **()** — создает пустой объект нулевой длины;

**Vector (int capacity)** — создает пустой объект указанной емкости capacity;

**Vector (int capacity, int increment)** — создает пустой объект указанной емкости capacity и задает число increment, на которое увеличивается емкость при необходимости;

**Vector (Collection с)** — вектор создается по указанной коллекции. Если capacity отрицательно, создается исключительная ситуация.

***Методы класса Vector:***

**add (Object element)** — позволяет добавить элемент в конец вектора:

**add (int index, Object element)** — можно вставить элемент в указанное место index. Элемент, находившийся на этом месте, и все последующие элементы сдвигаются, их индексы увеличиваются на единицу;

**addAll (Collection coll)** — позволяет добавить в конец вектора все элементы коллекции coll;

**addAll(int index, Collection coll)** — вставляет в позицию index все элементы коллекции coll.

**set (int index, object element)** — заменяет элемент, стоявший в векторе в позиции index, на элемент element;

Количество элементов в векторе всегда можно узнать методом **size().**

**capacity()**— возвращает емкость вектора.

Логический метод **isEmpty()** возвращает true, если в векторе нет ни одного элемента.

Обратиться к первому элементу вектора можно методом **firstEiement()**, к последнему — методом **lastEiement()**, к любому элементу — методом **get(int index)**. Эти методы возвращают объект класса Object. Перед использованием его следует привести к нужному типу.

Получить все элементы вектора в виде массива типа Object[] можно методами **toArray()**.

Логический метод **contains(Object element)** возвращает true, если элемент element находится в векторе.

Логический метод **containsAll(Collection с)** возвращает true, если вектор содержит все элементы указанной коллекции.

Четыре метода позволяют отыскать позицию указанного элемента element:

**indexOf(Object element)** — возвращает индекс первого появления элемента в векторе;

**indexOf(Object element, int begin)** — ведет поиск, начиная с индекса begin включительно;

**lastIndexOf(object element)** — возвращает индекс последнего появления элемента в векторе;

**lastIndexOf (Object element, int start)** — ведет поиск от индекса start включительно к началу вектора.

Если элемент не найден, возвращается —1.

Логический метод **remove(Object element)** удаляет из вектора первое вхождение указанного элемента element. Метод возвращает true, если элемент найден и удаление произведено.

Метод **remove (int index)** удаляет элемент из позиции index и возвращает его в качестве своего результата типа object.

Удалить диапазон элементов можно методом **removeRange(int begin, int end)**, не возвращающим результата. Удаляются элементы от позиции begin включительно до позиции end исключительно.

Удалить из данного вектора все элементы коллекции coll возможно методом **removeAll(Collection coll)**.

Удалить последние элементы можно, просто урезав вектор методом **setSize(int newSize)**.

Удалить все элементы, кроме входящих в указанную коллекцию coil, разрешает логический метод **retainAll(Collection coll).**

Удалить все элементы вектора можно методом **clear()** или старым методом **removeAllElements()** или обнулив размер вектора методом **setSize(0).**

***Пример 4.1. Работа с вектором***

Vector v = new Vector();

String s = "Строка, которую мы хотим разобрать на слова.";

StringTokenizer st = new StringTokenizer(s, " \t\n\r,.");

while (st.hasMoreTokens()) { // Получаем слово и

v.add(st.nextToken()); // добавляем в конец вектора

}

System.out.println(v.firstElement()); // Первый элемент

System.out.println(v.lastElement()); // Последний элемент

v.setSize(4); // Уменьшаем число элементов

v.add("собрать."); // Добавляем в конец укороченного вектора

v.set(3, "опять"); // Ставим в позицию 3

for (int i = 0; i < v.size(); i++) // Перебираем весь вектор

System.out.print(v.get(i) + " - ");

System.out.println();

Класс Vector является примером того, как можно объекты класса Object, a значит, любые объекты, объединить в коллекцию.

***Класс Stack.***Класс Stack расширяет класс Vector. Стек(stack) реализует порядок работы с элементами по принципу, который называется LIFO (Last In — First Out). Перед работой создается пустой стек конструктором Stack(). Затем на стек кладутся и снимаются элементы, причем доступен только "верхний" элемент, тот, что положен на стек последним.

Дополнительно к методам класса Vector класс Stack содержит пять методов, позволяющих работать с коллекцией как со стеком:

**push(Object item)** —помещает элемент item в стек;

**pop()** — извлекает верхний элемент из стека;

**peek()** — читает верхний элемент, не извлекая его из стека;

**empty()** — проверяет, не пуст ли стек;

**search(Object item)** — находит позицию элемента item в стеке. Верхний элемент имеет позицию 1, под ним элемент 2 и т. д. Если элемент не найден, возвращается — 1.

***Пример 4.2. Пример стека***

Stack stack = new Stack();

stack.push("aaa"); stack.push("bbb"); stack.push("ccc");

for (int i = 0; i < stack.size(); i++)

System.out.print(stack.get(i) + " - ");

System.out.println();

stack.pop();

for (int i = 0; i < stack.size(); i++)

System.out.print(stack.get(i) + " - ");

System.out.println();

***Класс Hashtable.*** Класс Hashtable расширяет абстрактныйкласс Dictionary**.** В объектах этого класса хранятся пары "ключ — значение". Из таких пар "Фамилия И. О. — номер" состоит, например, телефонный справочник.

Каждый объект класса Hashtable кроме размера(size) — количества пар, имеет еще две характеристики: емкость(capacity) — размер буфера, и показатель загруженности(load factor) — процент заполненности буфера, по достижении которого увеличивается его размер.

Для создания объектов класс Hashtable предоставляет четыре конструктора:

**Hashtable()** — создает пустой объект с начальной емкостью в 101 элемент и показателем загруженности 0,75;

**Hashtable (int capacity)** — создает пустой объект с начальной емкостью capacity и показателем загруженности 0,75;

**Hashtable(int capacity, float loadFactor)** — создает пустой объект с начальной емкостью capacity и показателем загруженности loadFactor;

**Hashtable (Map f)** — создает объект класса Hashtable, содержащий все элементы отображения f, с емкостью, равной удвоенному числу элементов отображения f, но не менее 11, и показателем загруженности 0,75.

Для заполнения объекта класса Hashtable используются два метода:

**Object put(Object key, Object value)** — добавляет пару "key— value", если ключа key не было в таблице, и меняет значение value ключа key, если он уже есть в таблице. Возвращает старое значение ключа или null, если его не было. Если хотя бы один параметр равен null, возникает исключительная ситуация.

**void putAll(Map f)** — добавляет все элементы отображения f. В объектах-ключах key должны быть реализованы методы hashCode() и equals ().

**get (Object key)** — возвращает значение элемента с ключом key в виде объекта класса Object. Для дальнейшей работы его следует преобразовать к конкретному типу;

**containsKey(Object key)** — возвращает true, если в таблице есть ключ key;

**containsValue(Object value**) — возвращает true, если в таблице есть ключи со значением value;

**isEmpty()** — возвращает true, если в таблице нет элементов;

**values()** — представляет все значения value таблицы в виде интерфейса Collection. Все модификации в объекте Сollection изменяют таблицу, и наоборот;

**keySet()**— предоставляет все ключи key таблицы в виде интерфейса Set. Все изменения в объекте Set корректируют таблицу, и наоборот;

**entrySet()** — представляет все пары " key— value " таблицы в виде интерфейса Set. Все модификации в объекте set изменяют таблицу, и наоборот;

**toString ()** — возвращает строку, содержащую все пары;

**remove(Object key)** — удаляет пару с ключом key, возвращая значение этого ключа, если оно есть, и null, если пара с ключом key не найдена;

**clear()** — удаляет все элементы, очищая таблицу.

***Пример 4.3. Телефонный справочник***

class PhoneBook {

public static void main(String[] args) {

Hashtable yp = new Hashtable();

String name = null;

yp.put("John", "123-45-67");

yp.put ("Lemon", "567-34-12");

yp.put("Bill", "342-65-87");

yp.put("Gates", "423-83-49");

yp.put("Batman", "532-25-08");

try {

name = args[0];

}

catch(Exception e) {

System.out.println("Usage: Java PhoneBook Name");

return;

}

if (yp.containsKey(name))

System.out.println(name + "'s phone = " + yp.get(name));

else System.out.println("Sorry, no such name");

}

}

***Класс Properties****.* Класс Properties расширяет класс Hashtable. Он предназначен в основном для ввода и вывода пар свойств системы и их значений (рис. 4.1) . Пары хранятся в виде строк типа string. В классе Properties два конструктора:

**Properties()** — создает пустой объект;

**Properties(Properties default)** — создает объект с заданными парами свойств default.

Кроме унаследованных от класса Hashtable методов в классе Properties есть еще следующие методы.

Два метода, возвращающих значение ключа-строки в виде строки:

**string getProperty (String key)** — возвращает значение по ключу key;

**String getProperty(String.key, String defaultValue)** — возвращает значение по ключу key,если такого ключа нет, возвращается defaultValue.

**setProperty(String key, String value)** — добавляет новую пару, если ключа key нет, и меняет значение, если ключ key есть;

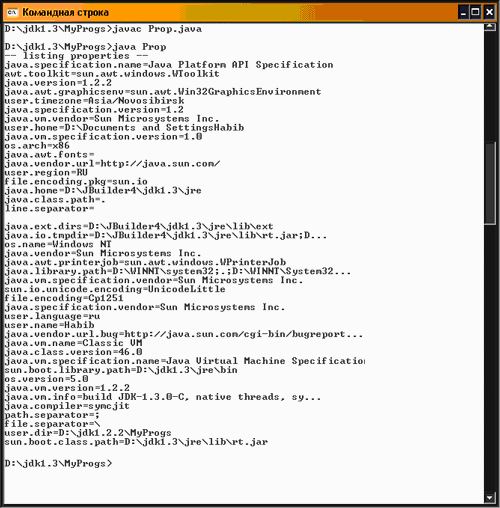
**load(InputStream in)** — загружает свойства из входного потока in;

**list(PrintStream out)** и **list(PrintWriter out)** — выводят свойства в выходной поток out;

**store(OutputStream out, String header)** — выводит свойства в выходной поток out с заголовком header.

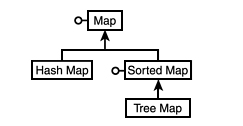
Пример вывода системных свойств:

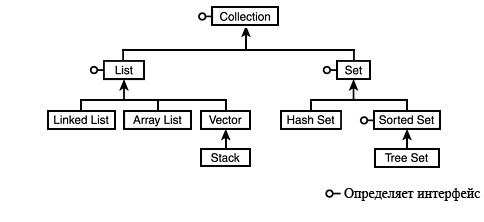
System.getProperties().list(System.out);



*Рис. 4.1.* Системные свойства

В Java 2 разработана целая иерархия коллекций. Она приведена на рис. 4.2.





*Рис. 4.2.* Иерархия интерфейсов и классов-коллекций в Java.

В таблице 4.1 приведены классы коллекций, построенные на основе интерфейсов.

Таблица. 4.1.

Реализация коллекций

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Интерфейс** | **Реалаизация** | | | | |
|  | **Хэш-таблицы** | **Массивы** | **Деревья** | **Связанные списки** | **Хэш-таблицы +списки** |
| Set | HashSet |  | TreeSet |  | LinkedHashSet |
| List |  | ArrayList |  | LinkedList |  |
| Map | HashMap |  | TreeMap |  | LinkedHashMap |

***Интерфейс Collection.*** Интерфейс Collection из пакета java.util описывает общие свойства коллекций List и Set. Он содержит методы добавления и удаления элементов, проверки и преобразования элементов:

**boolean add(Object obj)** — добавляет элемент obj в конец коллекции; возвращает false, если такой элемент в коллекции уже есть, а коллекция не допускает повторяющиеся элементы; возвращает true, если добавление прошло удачно;

**boolean addAll(Collection coll)** — добавляет все элементы коллекции coll в конец данной коллекции;

**void** **clear( )** — удаляет все элементы коллекции;

**boolean contains(Object obj)** — проверяет наличие элемента obj в коллекции;

**boolean containsAll(Collection coll )** — проверяет наличие всех элементов коллекции coll в данной коллекции;

**boolean isEmpty()** — проверяет, пуста ли коллекция;

**Iterator iterator()** — возвращает итератор данной коллекции;

**boolean remove(object obj)** — удаляет указанный элемент из коллекции; возвращает false, если элемент не найден, true, если удаление прошло успешно;

**boolean removeAll(Collection coll)** — удаляет элементы указанной коллекции, лежащие в данной коллекции;

**boolean retainAll(Collection coll )** — удаляет все элементы данной коллекции, кроме элементов коллекции coll ;

**int size()** — возвращает количество элементов в коллекции;

**Object[] toArray()** — возвращает все элементы коллекции в виде массива;

**Object[] toArray(Object[] a)** — записывает все элементы коллекции в массив а, если в нем достаточно места.

***Интерфейс List. Интерфейс*** List из пакета java.util, расширяющий интерфейс Collection, описывает методы работы с упорядоченными коллекциями. Иногда их называют последовательностями (sequence). Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу. В отличие от коллекции Set элементы коллекции List могут повторяться.

Класс Vector — одна из реализаций интерфейса List.

Интерфейс List добавляет к методам интерфейса Collection методы, использующие индекс index элемента:

**void add(int index, Object obj)** — вставляет элемент obj в позицию index; старые элементы, начиная с позиции index, сдвигаются, их индексы увеличиваются на единицу;

**boolean addAll(int index, Collection coll)** — вставляет все элементы коллекции coll ;

**Object get(int index)** — возвращает элемент, находящийся в позиции index;

**int indexOf(Object obj)** — возвращает индекс первого появления элемента obj в коллекции;

**int lasIindexOf(object obj)** — возвращает индекс последнего появления элемента obj в коллекции;

**ListIterator listIterator()** — возвращает итератор коллекции;

**ListIterator listIterator(int index)** — возвращает итератор конца коллекции от позиции index ;

**Object set (int index, object obj)** — заменяет элемент, находящийся в позиции index, элементом obj ;

**List subList(int from, int to)** — возвращает часть коллекции от позиции from включительно до позиции to исключительно.

***Интерфейс Set.*** Интерфейс Set из пакета java.util, расширяющий интерфейс Collection, описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов. Это соответствует математическому понятию множества (set). Такие коллекции удобны для проверки наличия или отсутствия у элемента свойства, определяющего множество. Новые методы в интерфейс Set не добавлены. Этот интерфейс расширен интерфейсом SortedSet.

***Интерфейс SortedSet.*** 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.

Элементы не нумеруются, но есть понятие первого, последнего, большего и меньшего элемента.

Дополнительные методы интерфейса отражают эти понятия:

**Comparator comparator()** — возвращает способ упорядочения коллекции;

**Object first()** — возвращает первый, меньший элемент коллекции;

**SortedSet headSet(Object toElement)** — возвращает начальные, меньшие элементы до элемента toElement исключительно;

**Object last()** — возвращает последний, больший элемент коллекции;

**SortedSet subSet(Object fromElement, Object toElement)** — возвращает подмножество коллекции от элемента fromElement включительно до элемента toElement исключительно;

**SortedSet tailSet(Object fromElement)** — возвращает последние, большие элементы коллекции от элемента fromElement включительно.

***Интерфейс Map****.*  Интерфейс Map из пакета java.util описывает коллекцию, состоящую из пар "ключ — значение". У каждого ключа только одно значение, что соответствует математическому понятию однозначной функции или отображения(mар). Такую коллекцию часто называют еще словарем(dictionary) или ассоциативным массивом(associative array).

Интерфейс Map содержит методы, работающие с ключами и значениями:

**boolean containsKey(Object key)** **—** проверяет наличие ключа  key ;

**boolean containsValue(Object value)** — проверяет наличие значения value ;

**Set entrySet()** — представляет коллекцию в виде множества, каждый элемент которого — пара из данного отображения, с которой можно работать методами вложенного интерфейса Map.Entry;

**Object get(Object key)** — возвращает значение, отвечающее ключу key;

**Set keyset()** — представляет ключи коллекции в виде множества;

**Object put(Object key, Object value)** — добавляет пару "key— value", если такой пары не было, и заменяет значение ключа key, если такой ключ уже есть в коллекции;

**void putAll (Map m)** — добавляет к коллекции все пары из отображения m;

**Collection values()** — представляет все значения в виде коллекции.

В интерфейс Mар вложен интерфейс Map.Entry, содержащий методы работы с отдельной парой.

***Вложенный интерфейс Map.Entry***. Этот интерфейс описывает методы работы с парами, полученными методом entrySet(): методы getKey() и getvalue() позволяют получить ключ и значение пары; метод setValue(Object value) меняет значение в данной паре.

***Интерфейс SortedMap.*** Интерфейс SortedMap, расширяющий интерфейс Map, описывает упорядоченную по ключам коллекцию мар. Сортировка производится либо в естественном порядке возрастания ключей, либо, в порядке, описываемом в интерфейсе Comparator.

Элементы не нумеруются, но есть понятия большего и меньшего из двух элементов, первого, самого маленького, и последнего, самого большого элемента коллекции. Эти понятия описываются следующими методами:

**Comparator comparator()** — возвращает способ упорядочения коллекции;

**Object firstKey()** — возвращает первый, меньший элемент коллекции;

**SortedMap headMap(Object toKey)** — возвращает начало коллекции до элемента с ключом toKey исключительно;

**Object lastKey()** — возвращает последний, больший ключ коллекции**;**

**SortedMap subMap(Object fromKey, Object toKey)** **—** возвращает часть коллекции от элемента с ключом fromKey включительно до элемента с ключом toKey исключительно;

**SortedMap tailMap(object fromKey)** — возвращает остаток коллекции от элемента fromKey включительно.

Вы можете создать свои коллекции, реализовав рассмотренные интерфейсы. Это дело трудное, поскольку в интерфейсах много методов. Чтобы облегчить эту задачу, в  Java API введены частичные реализации интерфейсов — абстрактные классы-коллекции.

***Интерфейс Iterator.*** В Java API введен интерфейс Iterator, описывающий способ обхода всех элементов коллекции. В каждой коллекции есть метод **iterator**(), возвращающий реализацию интерфейса Iterator для указанной коллекции. Получив эту реализацию, можно обходить коллекцию в порядке, определенном данным итератором, с помощью методов, описанных в интерфейсе Iterator и реализованных в этом итераторе (пример 4.4).

В интерфейсе Iterator описаны всего три метода:

* логический метод **hasNext()** возвращает true, если обход еще не завершен;
* метод **next()** делает текущим следующий элемент коллекции и возвращает его в виде объекта класса Object;
* метод **remove()** удаляет текущий элемент коллекции.

Итератор — это указатель на элемент коллекции. При создании итератора указатель устанавливается перед первым элементом, метод next() перемещает указатель на первый элемент и показывает его. Следующее применение метода next() перемещает указатель на второй элемент коллекции и показывает его. Последнее применение метода next () выводит указатель за последний элемент коллекции.

***Пример 4.4. Использование итератора вектора***

Vector v = new Vector();

…….

for (int i = 0; i < v.size(); i++)

System.out.print(v.get(i) + ".");

System.out.println();

Iterator it = v.iterator (); // Получаем итератор вектора

try {

while(it.hasNext()) // Пока в векторе есть элементы,

System.out.println(it.next()); // выводим текущий элемент

}

catch(Exception e){}

***Интерфейс  Listlterator.*** Интерфейс ListIterator расширяет интерфейс Iterator, обеспечивая перемещение по коллекции, как в прямом, так и в обратном направлении. Он может быть реализован только в тех коллекциях, в которых есть понятия следующего и предыдущего элемента и где элементы пронумерованы.

В интерфейс ListIterator добавлены следующие методы:

**void add(Object element)** — добавляет элемент element перед текущим элементом;

**boolean hasPrevious()** — возвращает true, если в коллекции есть элементы, стоящие перед текущим элементом;

**int nextIndex()** — возвращает индекс текущего элемента; если текущим является последний элемент коллекции, возвращает размер коллекции;

**Object previous()** — возвращает предыдущий элемент и делает его текущим;

**int previous index()** — возвращает индекс предыдущего элемента;

**void set(Object element)** **—** заменяет текущий элемент элементом element; выполняется сразу после next() или previous().

Как видите, итераторы могут изменять коллекцию, в которой они работают, добавляя, удаляя и заменяя элементы. Чтобы это не приводило к конфликтам, предусмотрена исключительная ситуация, возникающая при попытке использования итераторов параллельно "родным" методам коллекции. Именно поэтому в примере 4.4 действия с итератором заключены в блок try-catch().

ListIterator lit = v.listIterator(); // Получаем итератор вектора

try {

while(lit.hasNext()) // Пока в векторе есть элементы

// Переходим к следующему элементу и выводим его

System.out.println(lit.next());

// Теперь указатель за концом вектора. Пройдем к началу

while (lit. hasPrevious ())

System.out.println(lit. previous());

}

catch (Exception e){}

В составе Java API есть полностью реализованные классы-коллекции помимо уже рассмотренных классов Vector, Stack, Hashtable и Properties. Это классы ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap и другие.

***Классы, наследующие интерфейс List.*** Класс **ArrayList** очень похож на класс Vector, имеет тот же набор методов и может использоваться в тех же ситуациях.

В классе ArrayList три конструктора:

**ArrayList()**—создает пустой объект;

**ArrayList(Collection coll)** — создает объект, содержащий все элементы коллекции coll;

**ArrayList(int initCapacity)** — создает пустой объект емкости initCapacity.

Единственное отличие класса ArrayList от класса Vector заключается  в  том, что класс ArrayList не синхронизован. Это означает, что одновременное изменение экземпляра этого класса несколькими подпроцессами приведет к непредсказуемым результатам.

Класс **LinkedList** полностью реализует интерфейс List и содержит дополнительные методы, превращающие его в двунаправленный список. Он реализует итераторы типа Iterator и ListIterator.

Этот класс можно использовать для обработки элементов в стеке или двунаправленном списке.

В классе LinkedList два конструктора:

**LinkedList()** - создает пустой объект

**LinkedList(Collection coll)** — создает объект, содержащий все элементы коллекции coll.

***Классы, создающие отображения.*** Класс **HashMap** полностью реализует интерфейс Map, а также итератор типа Iterator. Класс HashMap очень похож на класс Hashtable и может использоваться в тех же ситуациях. Он имеет тот же набор функций и такие же конструкторы:

**HashMap()** — создает пустой объект с показателем загруженности 0,75;

**НаshМар(int capacity)** - создает пустой объект с начальной емкостью capacity и показателем загруженности 0,75;

**HashMap(int capacity, float loadFactor)** — создает пустой объект С начальной емкостью capacity и показателем загруженности loadFactor ;

**HashMap(Map f)** — создает объект класса HashMap, содержащий все элементы отображения f, с емкостью, равной удвоенному числу элементов отображения f, но не менее 11, и показателем загруженности 0,75.

***Упорядоченные отображения.*** Класс **ТгееМар** полностью реализует интерфейс SortedMap. Он реализован как бинарное дерево поиска, следовательно, его элементы хранятся в упорядоченном виде. Это значительно ускоряет поиск нужного элемента. Порядок задается либо естественным следованием элементов, либо объектом, реализующим интерфейс сравнения Comparator.

В этом классе четыре конструктора:

**ТгееМар()** — создает пустой объект с естественным  порядком элементов;

**TreeМар(Comparator с)** — создает пустой объект, в котором порядок задается объектом сравнения с;

**ТгееМар Map f)** — создает объект, содержащий все элементы отображения f, с естественным порядком его элементов;

**ТгееМар(SortedMap sf)** — создает объект, содержащий все элементы отображения sf, в том же порядке.

***Сравнение элементов коллекций.***

Интерфейс Comparator описывает два метода сравнения:

**int compare(Object obji, object obj2 )** — возвращает отрицательное число, если objl в каком-то смысле меньше obj2 ; нуль, если они считаются равными; положительное число, если objl больше obj2;

**boolean equals (Object obj)** — сравнивает данный объект с объектом obj, возвращая true, если объекты совпадают в каком-либо смысле, заданном этим методом.

Для каждой коллекции можно реализовать эти два метода, задав конкретный способ сравнения элементов, и определить объект класса SortedMap вторым конструктором. Элементы коллекции будут автоматически отсортированы в заданном порядке.

Пример 4.5 показывает один из возможных способов упорядочения комплексных чисел — объектов класса Complex. Здесь описывается класс ComplexCompare, реализующий интерфейс Comparator.

***Пример 4.5. Сравнение комплексных чисел***

class Complex {

public double Re = 0.0;

public double Im = 0.0;

public Complex(){}

public Complex(double re, double im) { Re = re; Im = im; }

}

class ComplexCompare implements Comparator {

public int compare(Object objl, Object obj2) {

Complex z1 = (Complex)objl;

Complex z2 = (Complex)obj2;

if (z1.Re != z2.Re) return (int)(z1.Re > z2.Re ? 1 : -1);

else if (z1.Im != z2.Im) return (int)(z1.Im > z2.Im ? 1 : -1);

else return 0;

}

public boolean equals(Object z) {

return compare(this, z) == 0;

}

}

***Классы, создающие множества.*** Класс **HashSet** полностью реализует интерфейс Set и итератор типа Iterator. Класс HashSet используется в тех случаях, когда надо хранить только одну копию каждого элемента.

В классе HashSet четыре конструктора:

**HashSet()** — создает пустой объект с показателем загруженности 0,75;

**HashSet(int capacity)** — создает пустой объект с начальной емкостью capacity и показателем загруженности 0,75;

**HashSet(int capacity, float loadFactor)** — создает пустой объект с начальной емкостью capacity и показателем загруженности loadFactor;

**HashSet(Collection coll)** — создает объект, содержащий все элементы коллекции coll, с емкостью, равной удвоенному числу элементов коллекции coll, но не менее 11, и показателем загруженности 0,75.

***Упорядоченные множества.*** Класс TreeSet полностью реализует интерфейс SortedSet и итератор типа Iterator. Класс TreeSet реализован как бинарное дерево поиска, его элементы хранятся в упорядоченном виде. Это значительно ускоряет поиск нужного элемента.

Порядок задается либо естественным следованием элементов, либо объектом, реализующим интерфейс сравнения Comparator.

В классе TreeSet четыре конструктора:

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

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

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

**TreeSet(SortedMap sf)** — создает объект, содержащий все элементы отображения sf, в том же порядке.

В пример 4.6 показано, как можно хранить комплексные числа в упорядоченном виде. Порядок задается объектом класса ComplexCompare.

***Пример 4.6. Хранение комплексных чисел в упорядоченном виде***

TreeSet ts = new TreeSet(new ComplexCompare());

ts.add(new Complex(1.2, 3.4));

ts.add(new Complex(-1.25, 33.4));

ts.add(new Complex(1.23, -3.45));

ts.add(new Complex(16.2, 23.4));

Iterator it = ts.iterator();

while(it.hasNext()) {

Complex c = (Complex)it.next();

System.out.println("(" + c.Re + "; " + c.Im + ")");

}

**Практические задания**

1. Изучить особенности реализации классов-коллекций в Java.

**Задачи:**

**Задача 1**. Сгенерируйте массив целых чисел, используя классRandom(). Используя методы классов-коллекций:

А) создайте новый массив, переписав в него часть элементов из первого массива

Б) определите уникальные числа

В) определите количество каждого уникального числа

Г) отсортируйте массив

Д) найдите макс, мин и сумму чисел массива

Е) создайте массив, содержащий все положительные числа первого массива

Ж) удалите из первого массива все нечетные числа

З) найдите указанное число в массиве

**Задача 2**. Проведите частотный анализ слов и символов из первой строки. Определите символы, которые:

А) входят и в первую и во вторую строку

Б) входят в первую и не входят во вторую

В) содержатся хотя бы в одной строке.

**Задача 3.**Напишите функцию проверки вхождения списка L1 в список L2.

**Задача 4.**Напишите процедуру формирования списка L путем включения в него по одному разу элементов, входящих одновременно в списки L1 и L2.

**Задача 5.**Опишите очередь и заполните ее вещественными числами. Выполните циклический сдвиг элементов в очереди так, чтобы в ее начале был расположен наибольший элемент.

**Задача 6**. Задача со скобками. Вводится символьная строка, в которой записано выражение со скобками трех типов: [], {} и (). Определить, верно ли расставлены скобки (не обращая внимания на остальные символы).

**Задача 7**. Ввести строки, записать их в стек. Вывести строки в обратном порядке.

**Задача 8.**Ввести число, занести его цифры в стек. Вывести в число, у которого цифры идут в обратном порядке.

**Задача 9.**Не используя вспомогательных объектов, переставить отрицательные элементы данного списка в конец, а положительные - в начало этого списка.

**Задача 10.**Выполнить попарное суммирование произвольного конечного ряда чисел следующим образом: на первом этапе суммируются попарно рядом стоящие числа, на втором этапе суммируются результаты первого этапа и т.д. до тех пор, пока не останется одно число.