Лабораторная работа №4

Работа с коллекциями в Java

1. Введение

Для хранения большого количества однотипных данных могут использоваться массивы, но они не всегда являются идеальным решением. Во-первых, длина массива задается заранее и в случае, если количество элементов заранее неизвестно, придется либо выделять память «с запасом», либо предпринимать сложные действия по переопределению массива. Во-вторых, элементы массива имеют жестко заданное размещение в его ячейках, поэтому, например, удаление элемента из массива не является простой операцией.

В программировании давно и эффективно используются такие структуры данных как стек, очередь, список, множество и т.д., объединенные общим названием коллекция. Коллекция — это группа элементов с операциями добавления, извлечения и поиска элемента. Механизм работы операций существенно различается в зависимости от типа коллекции. Например, элементы стека упорядочены в последовательность, добавление нового элемента может происходить только в конец этой последовательности, и получить можно только элемент, находящийся в конце (то есть, добавленный последним). Очередь, напротив, позволяет получить лишь первый элемент (элементы добавляются в один конец последовательности, а «забираются» с другого). Другие коллекции (например, список) позволяют получить элемент из любого места последовательности, а множество вообще не упорядочивает элементы и позволяет (помимо добавления и удаления) только узнать, содержится ли в нем данный элемент.

Язык Java предоставляет библиотеку стандартных коллекций, которые собраны в пакете java.util, поэтому нет необходимости программировать их самостоятельно.

При работе с коллекциями главное избегать ошибки начинающих — пользоваться наиболее универсальной коллекцией вместо той, которая необходима для решения задачи — например, списком вместо стека. Если логика работы программы такова, что данные должны храниться в стеке (появляться и обрабатываться в обратной последовательности), следует использовать именно стек. В этом случае вы не сможете нарушить логику обработки данных, обратившись напрямую к середине последовательности, а значит, шанс появления трудно обнаруживаемых ошибок резко уменьшается.

Чтобы выбрать коллекцию, которая лучше всего подходит условию задачи, необходимо знать особенности каждой из них. Эти знания являются обязательными для любого программиста, поскольку без применения тех или иных коллекций редко обходится любая современная задача. Некоторые сведения вы сможете почерпнуть из дальнейшего изложения.

2. Общие сведения

2.1 Классы-коллекции

Коллекции в библиотеке java.util представлены набором классов и интерфейсов.

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

Элементы большинства коллекций имеют тип Object. Это значит, что (в отличие от обычного массива) вы не должны заранее указывать тип элементов, которые будете помещать в коллекцию. Вы можете добавлять в нее объекты любого класса, поскольку все классы

являются наследниками класса Object, более того — в одной коллекции могут храниться объекты совершенно разных классов.

Конечно, это может привести и к трудностям. Если вы захотите совершать какие-то операции над элементом коллекции (а вы помещаете в коллекцию объекты именно для того, чтобы потом их извлекать и обрабатывать), вы не сможете воспользоваться его методами, не приведя объект к его «настоящему» классу посредством явного приведения типов.

Например, у вас есть объект класса String (обычная строка) и вы хотите добавить его в стек stack с помощью метода push(Object item):

```
String str = "Ценная строка"; stack.push(str);
```

Коллекция, хранящая элементы типа Object, сразу же «забывает», что ваш объект — строка, поскольку при его добавлении было осуществлено автоматическое приведение типа String к типу Object. Вы можете извлечь ваш объект методом pop(), но чтобы вернуть ему прежний тип (а без этого вы не сможете воспользоваться ни одним из его методов), придется осуществить явное преобразование оператором (String):

```
str = (String)stack.pop();
```

Это не является проблемой, просто нужно помнить, объекты какого класса вы помещаете в коллекцию и выполнять соответствующее преобразование типа над каждым побывавшим в коллекции элементом.

Но если попытаться привести объект к неправильному типу, возникнет ошибка в программе:

```
stack.push(new Dog("Шарик", 12)); // помещаем в стек собаку
str = (Mouse)stack.pop(); // вынимаем собаку и пытаемся объявить ее мышью, но эти классы не связаны наследованием и программа выдаст ошибку во время выполнения
```

Так что, возможностью помещать в коллекцию *любые* объекты, воспользоваться скорее всего не получится. Представьте себе, что ваша программа должна «помнить», что первый элемент списка — строка, второй — число, третий — собака, четвертый — опять число. И обработать их все в цикле она, наверное, не сможет, поскольку у этих объектов нет общих методов. В результате теряются все преимущества использования коллекции. Поэтому в каждую коллекцию следует помещать объекты только одного класса (и производных от него).

2.2 Интерфейсы коллекций

Некоторые коллекции в пакете java.utils не представлены самостоятельными классами. Например, очередь и список. Но для этих полезных структур данных определены соответствующие интерфейсы, то есть можно пользоваться любым классом, реализующим такой интерфейс.

Интерфейс может использоваться в качестве типа данных так же, как и класс. Разница лишь в том, что объекты интерфейсного типа нельзя создать напрямую — необходимо выбрать класс, поддерживающий этот интерфейс и вызвать его конструктор.

Пусть, к примеру, нужен список, одна из наиболее удобных и часто употребляемых структур данных. Список — это упорядоченная последовательность элементов, каждый из которых можно получить по его позиции. Кроме того можно добавлять элементы в требуемые позиции списка и удалять их, при этом (и это главное удобство в отличие от массива) остальные элементы автоматически «раздвигаются» или «сдвигаются» так, что непрерывная связность списка сохраняется.

Список представлен в пакете java.util интерфейсом List. Вы можете создавать переменные этого типа и объявлять функции с таким параметром. Например, класс Game в программе игры в шашки имеет поле checkers типа List, хранящее все черные и белые шашки (объекты типа Checker). Когда шашку съедают, ее надо просто удалить из списка с помощью одного из удобных методов, определенных в интерфейсе List:

```
checkers.remove(check);  // удаляем из списка checkers съеденную шашку check
```

Когда программе понадобится узнать обо всех оставшихся шашках (например, чтобы нарисовать их на экране), метод getCheckers() класса Game передаст ей список:

```
List ch = currentGame.getCheckers(); // здесь currentGame – объект класса Game
```

Теперь программа может работать с переменной ch как со списком (например, по очереди получить все его объекты).

В момент создания новой игры (т.е. в конструкторе класса Game) надо, очевидно, создать 24 шашки, расположенные на стандартных позициях и добавить их в список checkers. Но список тоже необходимо создать, а мы не можем воспользоваться конструкцией

```
checkers = new List();
```

поскольку List не является классом и не имеет конструктора. Нам нужно выбрать любой *класс, реализующий интерфейс* List и создать объект этого класса. Например, класс Vector

```
checkers = new Vector();
```

или класс ArrayList*:

```
checkers = new ArrayList();
```

Независимо от того, какой именно класс мы выберем, поле checkers будет иметь тип List и на дальнейшую работу с ним наш выбор не повлияет. Мы будем добавлять шашки в список, удалять их из него, возвращать хранящиеся в списке шашки и т.д. посредством методов интерфейса List.

2.2.1 Интерфейс Collection.

Данный интерфейс является корнем всей иерархии классов-коллекций. Он определяет базовую функциональность любой коллекции - набор методов, которые позволяют добавлять, удалять, выбирать элементы коллекции. Классы, которые реализуют интерфейс Collection, могут содержать дубликаты и пустые (null) значения.

AbstractCollection, как абстрактный класс, служит основой для создания конкретных классов коллекций и содержит реализацию некоторых методов, определенных в интерфейсе Collection.

Интерфейс Collection содержит набор общих методов, которые используются в большинстве коллекций. Рассмотрим основные из них.

`add(Object item)` — добавляет в коллекцию новый элемент. Метод возвращает true, если добавление прошло удачно и false — если нет. Если элементы коллекции каким-то образом упорядочены, новый элемент добавляется в конец коллекции.

`clear()` — удаляет все элементы коллекции.

`contains(Object obj)` — возвращает true, если объект obj содержится в коллекции и false, если нет.

`isEmpty()` — проверяет, пуста ли коллекция.

`remove(Object obj)` — удаляет из коллекции элемент obj. Возвращает false, если такого элемента в коллекции не нашлось.

`size()` — возвращает количество элементов коллекции.

Существуют разновидности перечисленных методов, которые в качестве параметра принимают любую другую коллекцию. Например, метод addAll(Collection coll) добавляет все элементы другой коллекции coll в конец данной, метод removeAll(Collection coll) удаляет из коллекции все элементы, которые присутствуют также в коллекции coll, а метод retainAll(Collection coll) поступает наоборот, удаляя все элементы, кроме содержащихся в coll.

Метод toArray() возвращает все элементы коллекции в виде массива.

2.2.2 Интерфейс Set

Классы, которые реализуют этот интерфейс, не допускают наличия дубликатов. В коллекции этого типа разрешено наличие только одной ссылки типа null. Интерфейс Set расширяет интерфейс Collection, таким образом, любой класс, имплементирующий Set, реализует все методы, определенные в Collection. Любой объект, добавляемый в Set, должен реализовать метод equals, чтобы его можно было сравнить с другими.

AbstractSet, являясь абстрактным классом, представляет собой основу для реализации различных вариантов интерфейса Set.

2.2.3 Интерфейс List

Классы, реализующие этот интерфейс, содержат упорядоченную последовательность объектов (объекты хранятся в том порядке, в котором они были добавлены). В JDK 1.2 был переделан класс Vector, так, что он теперь реализует интерфейс List. Интерфейс List расширяет интерфейс Collection, и любой класс, имплементирующий List, реализует все методы, определенные в Collection, и в то же время вводятся новые методы, которые позволяют добавлять и удалять элементы из списка. List также обеспечивает ListIterator, который позволяет перемещаться как вперед, так и назад по элементам списка.

AbstractList, как абстрактный класс, представляет собой основу для реализации различных вариантов интерфейса List.

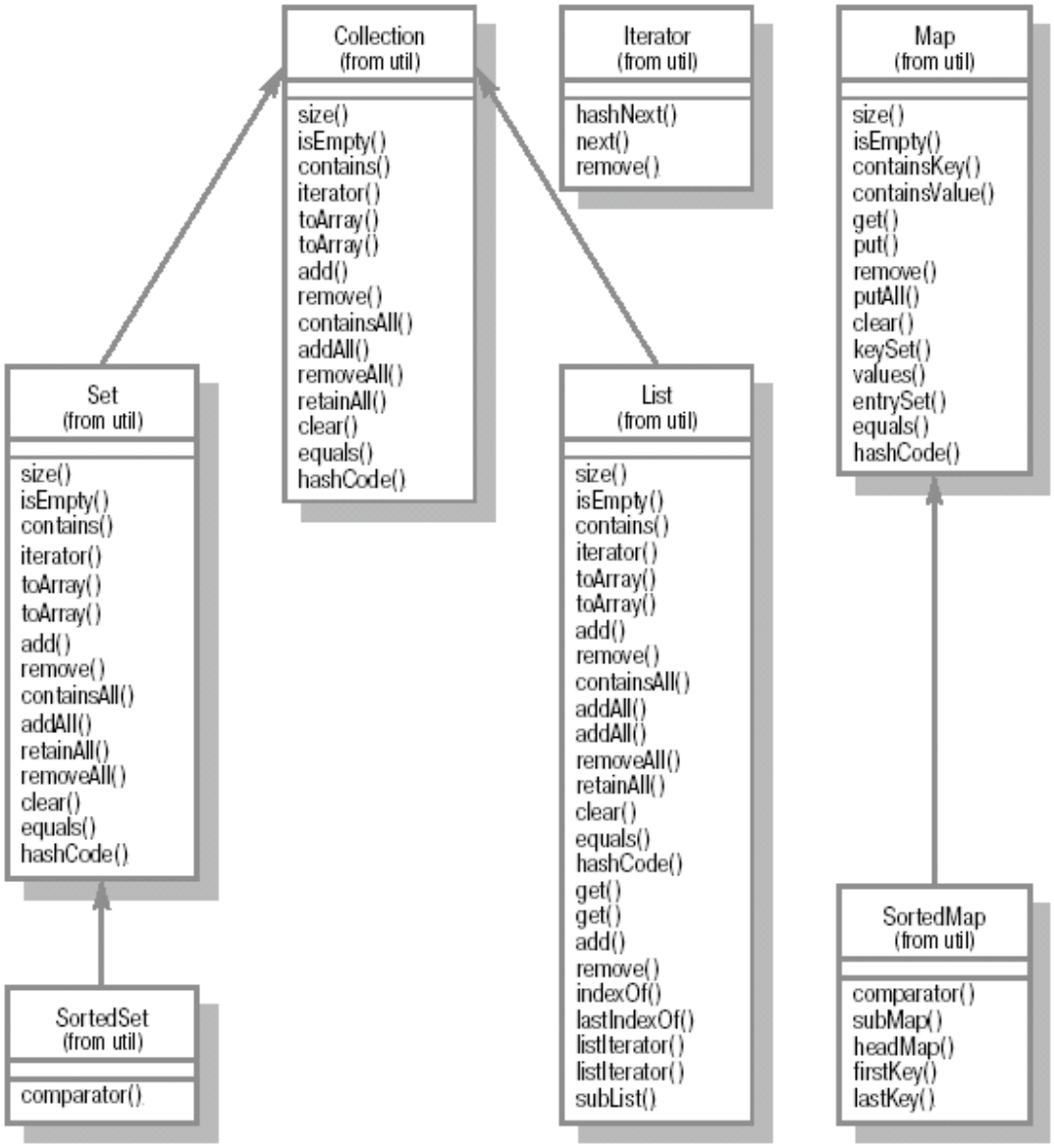

Рис. 1. Основные типы для работы с коллекциями.

2.2.4 Интерфейс Map.

Классы, которые реализуют этот интерфейс, хранят неупорядоченный набор объектов парами ключ/значение. Каждый ключ должен быть уникальным. Hashtable после модификации в JDK 1.2 реализует интерфейс Map. Порядок следования пар ключ/значение не определен.

Интерфейс Map не расширяет интерфейс Collection. AbstractMap, будучи абстрактным классом, представляет собой основу для реализации различных вариантов интерфейса Map.

2.2.5 Интерфейс SortedSet

Этот интерфейс расширяет Set, требуя, чтобы содержимое набора было упорядочено. Такие коллекции могут содержать объекты, которые реализуют интерфейс Comparable, либо могут сравниваться с использованием внешнего Comparator.

2.2.6 Интерфейс SortedMap

Этот интерфейс расширяет Map, требуя, чтобы содержимое коллекции было упорядочено по значениям ключей.

2.2.7 Интерфейс Iterator

В Java 1 для перебора элементов коллекции использовался интерфейс Enumeration. В Java 2 для этих целей должны применяться объекты, которые реализуют интерфейс Iterator. Все классы, которые реализуют интерфейс Collection, должны реализовать метод iterator, который возвращает объект, реализующий интерфейс Iterator. Iterator весьма похож на Enumeration, с тем лишь отличием, что в нем определен метод remove, который позволяет удалить объект из коллекции, для которой Iterator был создан.

Подводя итог, перечислим интерфейсы, используемые при работе с коллекциями:

```
java.util.Collection

java.util.Set

java.util.List

java.util.Map

java.util.SortedSet

java.util.SortedMap

java.util.Iterator
```

2.2 Абстрактные классы, используемые при работе с коллекциями

java.util.AbstractCollection - данный класс реализует все методы, определенные в интерфейсе Collection, за исключением iterator и size, так что для того, чтобы создать немодифицируемую коллекцию, нужно переопределить эти методы. Для реализации модифицируемой коллекции необходимо еще переопределить метод public void add(Object o) (в противном случае при его вызове будет возбуждено исключение UnsupportedOperationException).

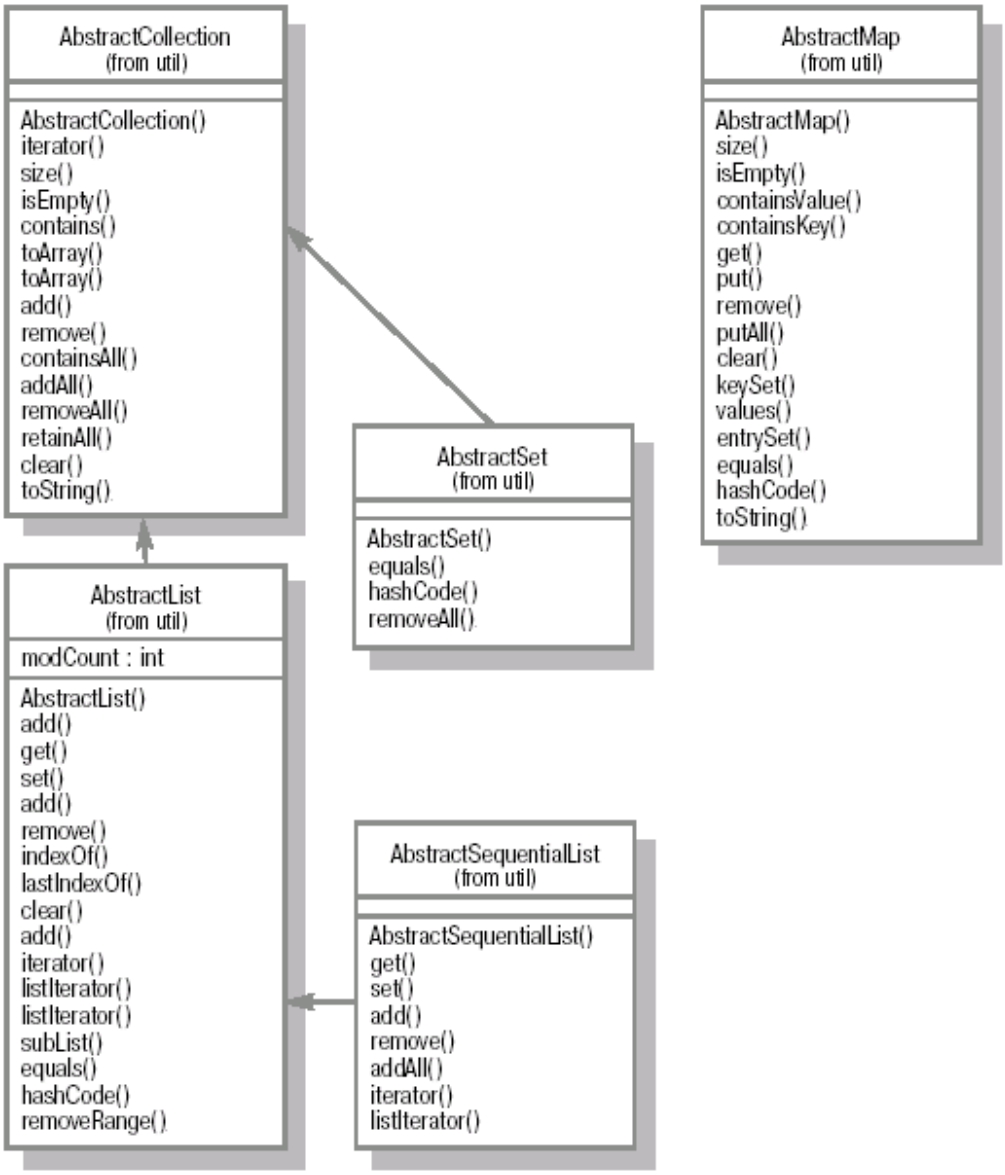

Рис. 2. Базовые абстрактные классы.

Необходимо также определить два конструктора: без аргументов и с аргументом Collection. Первый должен создавать пустую коллекцию, второй - коллекцию на основе существующей. Данный класс расширяется классами AbstractList и AbstractSet.

java.util.AbstractList - этот класс расширяет AbstractCollection и реализует интерфейс List. Для создания немодифицируемого списка необходимо имплементировать методы public Object get(int index) и public int size(). Для реализации модифицируемого списка необходимо также реализовать метод public void set(int index,Object element) (в противном случае при его вызове будет возбуждено исключение UnsupportedOperationException).

В отличие от AbstractCollection, в этом случае нет необходимости реализовывать метод iterator, так как он уже реализован поверх методов доступа к элементам списка get, set, add, remove.

java.util.AbstractSet - данный класс расширяет AbstractCollection и реализует основную функциональность, определенную в интерфейсе Set. Следует отметить, что этот класс не

переопределяет функциональность, реализованную в классе AbstractCollection.

java.util.AbstractMap - этот класс расширяет основную функциональность, определенную в интерфейсе Map. Для реализации немодифицируемого класса, унаследованного от AbstractMap, достаточно определить метод entrySet, который должен возвращать объект, приводимый к типу AbstractSet. Этот набор (Set) не должен обеспечивать методов для добавления и удаления элементов из набора. Для реализации модифицируемого класса Map необходимо также переопределить метод put и добавить в итератор, возвращаемый entrySet().iterator(), поддержку метода remove.

java.util.AbstractSequentialList - этот класс расширяет AbstractList и является основой для класса LinkedList. Основное отличие от AbstractList заключается в том, что этот класс обеспечивает не только последовательный, но и произвольный доступ к элементам списка, с помощью методов get(int index), set(int index, Object element), add(int index, Object element) и remove(int index). Для того, чтобы реализовать данный класс, необходимо переопределить методы listIterator и size. Причем, если реализуется немодифицируемый список, для итератора достаточно реализовать методы hasNext, next, hasPrevious, previous и index. Для модифицируемого списка необходимо дополнительно реализовать метод set, а для списков переменной длины еще и add, и remove.

2.3 Конкретные классы коллекций

java.util.ArrayList - этот класс расширяет AbstractList и весьма похож на класс Vector. Он также динамически расширяется, как Vector, однако его методы не являются синхронизированными, вследствие чего операции с ним выполняются быстрее. Для того, чтобы воспользоваться синхронизированной версией ArrayList, можно применить вот такую конструкцию:

```
List l = Collections.synchronizedList(new ArrayList(...));
public class Test {
  public Test() {
  }
  public static void main(String[] args) {
    Test t = new Test();
    ArrayList al = new ArrayList();
    al.add("First element");
    al.add("Second element");
    al.add("Third element");
    Iterator it = al.iterator();
    while(it.hasNext()) {
      System.out.println((String)it.next());
    }
    System.out.println("\n");
    al.add(2,"Insertion");
    it = al.iterator();
    while(it.hasNext()){
      System.out.println((String)it.next());
    }
  }
}
```

Результатом будет:

```
First element
Second element
Third element
Firts element
```

```
Second element
Insertion
Third element
```

java.util.LinkedList - представляет собой реализацию интерфейса List. Он реализует все методы интерфейса List, помимо этого добавляются еще новые методы, которые позволяют добавлять, удалять и получать элементы в конце и начале списка. LinkedList является двухсвязным списком и позволяет перемещаться как от начала в конец списка, так и наоборот. LinkedList удобно использовать для организации стека.

```
public class Test {
   public Test() {
   }
   public static void main(String[] args) {
      Test test = new Test();
      LinkedList ll = new LinkedList();
      ll.add("Element1");
      ll.addFirst("Element2");
      ll.addFirst("Element3");
      ll.addLast("Element4");
      test.dumpList(ll);
      ll.remove(2);
      test.dumpList(ll);
      String element = (String)ll.getLast();
      ll.remove(element);
      test.dumpList(ll);
   }
   private void dumpList(List list){
      Iterator it = list.iterator();
      System.out.println();
      while(it.hasNext()){
        System.out.println((String)it.next());
      }
   }
}
```

Результатом будет:

```
Element3
Element2
Element1
Element4
Element3
Element2
Element4
Element3
Element2
```

Классы LinkedList и ArrayList имеют схожую функциональность. Однако с точки зрения производительности они отличаются. Так, в ArrayList заметно быстрей (примерно на порядок) осуществляются операции прохода по всему списку (итерации) и получения данных. LinkedList почти на порядок быстрее выполняет операции удаления и добавления новых элементов.

java.util.Hashtable - расширяет абстрактный класс Dictionary. В JDK 1.2 класс Hashtable также реализует интерфейс Map. Hashtable предназначен для хранения объектов в виде пар ключ/значение. Из самого названия следует, что Hashtable использует алгоритм хэширования для увеличения скорости доступа к данным. Для того, чтобы выяснить принципы работы

данного алгоритма, рассмотрим несколько примеров.

Предположим, имеется массив строк, содержащий названия городов. Для того, чтобы найти элемент массива, содержащий название города, в общем случае требуется просмотреть весь массив, а если необходимо найти все элементы массива, то для поиска каждого, в среднем, потребуется просматривать половину массива. Такой подход может оказаться приемлемым только для небольших массивов.

Как уже отмечалось ранее, для того, чтобы увеличить скорость поиска, используется алгоритм хэширования. Каждый объект в Java унаследован от Object. Как уже отмечалось ранее, hash определено как целое число, которое уникально идентифицирует экземпляр класса Object и, соответственно, все экземпляры классов, унаследованных от Object. Это число возвращает метод hashCode(). Именно оно используется при сохранении ключа в Hashtable следующим образом: разделив длину массива, предназначенного для хранения ключей, на код, получаем некое целое число, которое служит индексом для хранения ключа в массиве array.length % hashCode().

Далее, если необходимо добавить новую пару ключ/значение, вычисляется новый индекс, и если этот индекс совпадает с уже имеющимся, то создается список ключей, на который указывает элемент массива ключей. Таким образом, при обратном извлечении ключа необходимо вычислить индекс массива по тому же алгоритму и получить его. Если ключ в массиве единственный, то используется значение элемента массива, если хранится несколько ключей, то необходимо обойти список и выбрать нужный.

Есть несколько соображений, относящихся к производительности классов, использующих для хранения данных алгоритм хэширования. В частности, размер массива. Если массив окажется слишком мал, то связанные списки будут слишком длинными и скорость поиска станет существенно снижаться, так как просмотр элементов списка будет такой же, как в обычном массиве. Чтобы этого избежать, задается некий коэффициент заполнения. При заполнении элементов массива, в котором хранятся ключи (или списки ключей) на эту величину, происходит увеличение массива и производится повторное реиндексирование. Таким образом, если массив окажется слишком мал, то он будет быстро заполняться и будет производиться операция повторного индексирования, которая отнимает достаточно много ресурсов. С другой стороны, если массив сделать большим, то при необходимости просмотреть последовательно все элементы коллекции, использующей алгоритм хэширования, придется обрабатывать большое количество пустых элементов массива ключей.

Начальный размер массива и коэффициент загрузки коллекции задаются при конструировании. Например:

```
Hashtable ht = new Hashtable(1000,0.60)
```

Существует также конструктор без параметров, который использует значения по умолчанию 101 для размера массива (в последней версии значение уменьшено до 11) и 0.75 для коэффициента загрузки.

Использование алгоритма хэширования позволяет гарантировать, что скорость доступа к элементам коллекции такого типа будет увеличиваться не линейно, а логарифмически. Таким образом, при частом поиске каких-либо значений по ключу имеет смысл задействовать коллекции, применяющие алгоритм хэширования.

java.util.HashMap - этот класс расширяет AbstractMap и весьма похож на класс Hashtable. HashMap предназначен для хранения пар объектов ключ/значение. Как для ключей, так и для элементов допускаются значения типа null. Порядок хранения элементов в этой коллекции не совпадает с порядком их добавления. Порядок элементов в коллекции также может меняться во времени. HashMap обеспечивает постоянное время доступа для операций get и put.

Итерация по всем элементам коллекции пропорциональна ее емкости. Поэтому имеет смысл не делать размер коллекций чрезмерно большим, если достаточно часто придется осуществлять итерацию по элементам.

Методы HashMap не являются синхронизированными. Для того, чтобы обеспечить нормальную работу в многопоточном варианте, следует использовать либо внешнюю синхронизацию потоков, либо синхронизированный вариант коллекции.

```
public class Test {
   private class TestObject{
      String text = "";
      public TestObject(String text){
         this.text = text;
      };
      public String getText(){
         return this.text;
      }
      public void setText(String text){
         this.text = text;
      }
   }
   public Test() {
   }
   public static void main(String[] args) {
      Test t = new Test();
      TestObject to = null;
      HashMap hm = new HashMap();
      hm.put("Key1",t.new TestObject("Value 1"));
      hm.put("Key2",t.new TestObject("Value 2"));
      hm.put("Key3",t.new TestObject("Value 3"));
      to = (TestObject)hm.get("Key1");
      System.out.println("Object value for Key1 = " + to.getText() +
"\n");
      System.out.println("Iteration over entrySet");
      Map.Entry entry = null;
      Iterator it = hm.entrySet().iterator();
      // Итератор для перебора всех точек входа в Map
      while(it.hasNext()){
         entry = (Map.Entry)it.next();
         System.out.println("For key = " + entry.getKey() +
            " value = " + ((TestObject)entry.getValue()).getText());
      }
      System.out.println();
      System.out.println("Iteration over keySet");
      String key = "";
      // Итератор для перебора всех ключей в Map
      it = hm.keySet().iterator();
      while(it.hasNext()){
         key = (String)it.next();
         System.out.println( "For key = " + key + " value = " +
            ((TestObject)hm.get(key)).getText());
      }
   }
}
```

Результатом будет:

```
Object value for Key1 = Value 1

Iteration over entrySet
For key = Key3 value = Value 3
For key = Key2 value = Value 2
For key = Key1 value = Value 1

Iteration over keySet
For key = Key3 value = Value 3
For key = Key2 value = Value 2
For key = Key1 value = Value 1
```

java.util.TreeMap - расширяет класс AbstractMap и реализует интерфейс SortedMap. TreeMap содержит ключи в порядке возрастания. Используется либо натуральное сравнение ключей, либо должен быть реализован интерфейс Comparable. Реализация алгоритма поиска обеспечивает логарифмическую зависимость времени выполнения основных операций (containsKey, get, put и remove). Запрещено применение null значений для ключей. При использовании дубликатов ключей ссылка на объект, сохраненный с таким же ключом, будет утеряна. Например:

```
public class Test {

   public Test() {
   }
   public static void main(String[] args) {
      Test t = new Test();
      TreeMap tm = new TreeMap();
      tm.put("key","String1");
      System.out.println(tm.get("key"));
      tm.put("key","String2");
      System.out.println(tm.get("key"));
   }
}
```

Результатом будет:

```
String1
String2
```

2.4. Обзор некоторых вспомогательных классов

2.4.1 Класс Collections

Класс Collections является классом-утилитой и содержит несколько вспомогательных методов для работы с классами, обеспечивающими различные интерфейсы коллекций. Например, для сортировки элементов списков, для поиска элементов в упорядоченных коллекциях и т.д. Но, пожалуй, наиболее важным свойством этого класса является возможность получения синхронизированных вариантов классов-коллекций. Например, для получения синхронизированного варианта Map можно использовать следующий подход:

```
HashMap hm = new HashMap();
:
Map syncMap = Collections.synchronizedMap(hm);
:
```

Как уже отмечалось ранее, начиная с JDK 1.2, класс Vector реализует интерфейс List. Рассмотрим пример сортировки элементов, содержащихся в классе Vector.

```
public class Test {
   private class TestObject {
      private String name = "";
      public TestObject(String name) {
         this.name = name;
      }
   }
   private class MyComparator implements Comparator {
      public int compare(Object l,Object r) {
         String left = (String)l;
         String right = (String)r;
         return -1 * left.compareTo(right);
      }
   }
   public Test() {
   }

   public static void main(String[] args) {
      Test test = new Test();
      Vector v = new Vector();
      v.add("bbbbb");
      v.add("aaaaa");
      v.add("ccccc");
      System.out.println("Default elements order");
      test.dumpList(v);
      Collections.sort(v);
      System.out.println("Default sorting order");
      test.dumpList(v);
      System.out.println("Reverse sorting order with providing
imlicit comparator");
      Collections.sort(v,test.new MyComparator());
      test.dumpList(v);
   }
   private void dumpList(List l) {
      Iterator it = l.iterator();
      while(it.hasNext()) {
         System.out.println(it.next());
      }
   }
}
```

2.4.2 Класс Properties

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.

Метод **load(InputStream inStream)** позволяет загрузить набор свойств из входного потока (потоки данных подробно рассматриваются в лекции 15). Как правило, это текстовый файл, в котором хранятся параметры. Параметры - это строки, которые представляют собой пары ключ/значение. Предполагается, что по умолчанию используется кодировка ISO 8859-1. Каждая строка должна оканчиваться символами \r,\n или \r\n. Строки из файла будут

считываться до тех пор, пока не будет достигнут его конец. Строки, состоящие из одних пробелов, или начинающиеся со знаков ! или #, игнорируются, т.е. их можно трактовать как комментарии. Если строка оканчивается символом /, то следующая строка считается ее продолжением. Первый символ с начала строки, отличный от пробела, считается началом ключа. Первый встретившийся пробел, двоеточие или знак равенства считается окончанием ключа. Все символы окончания ключа при необходимости могут быть включены в название ключа, но при этом перед ними должен стоять символ \. После того, как встретился символ окончания ключа, все аналогичные символы будут проигнорированы до начала значения. Оставшаяся часть строки интерпретируется как значение. В строке, состоящей только из символов \t, \n, \r, \\, \", \', \ и \uxxxx, они все распознаются и интерпретируются как одиночные символы. Если встретится сочетание \ и символа конца строки, то следующая строка будет считаться продолжением текущей, также будут проигнорированы все пробелы до начала строки-продолжения.

Метод **save(OutputStream inStream,String header)** сохраняет набор свойств в выходной поток в виде, пригодном для вторичной загрузки с помощью метода load. Символы, считающиеся служебными, кодируются так, чтобы их можно было считать при вторичной загрузке. Символы в национальной кодировке будут приведены к виду \uxxxx. При сохранении используется кодировка ISO 8859-1. Если указан header, то он будет помещен в начало потока в виде комментария (т.е. с символом # в начале), далее будет следовать комментарий, в котором будет указано время и дата сохранения свойств в потоке.

В классе Properties определен еще метод list(PrintWriter out), который практически идентичен save. Отличается лишь заголовок, который изменить нельзя. Кроме того, строки усекаются по ширине. Поэтому данный метод для сохранения Properties не годится.

```
public class Test {
   public Test() {
   }
   public static void main(String[] args) {
      Test test = new Test();
      Properties props = new Properties();
      StringWriter sw = new StringWriter();
      sw.write("Key1 = Value1 \n");
      sw.write(" Key2 : Value2 \r\n");
      sw.write(" Key3 Value3 \n ");
      InputStream is = new
ByteArrayInputStream(sw.toString().getBytes());

      try {
         props.load(is);
      }
      catch (IOException ex) {
         ex.printStackTrace();
      }
      props.list(System.out);
      props.setProperty("Key1","Modified Value1");
      props.setProperty("Key4","Added Value4");
      props.list(System.out);
   }
}
```

Результатом будет:

```
-- listing properties --
Key3=Value3
```

```
Key2=Value2
Key1=Value1

-- listing properties --
Key4=Added Value4
Key3=Value3
Key2=Value2
Key1=Modified Value1
```

2.4.3 Интерфейс Comparator

В коллекциях многие методы сортировки или сравнения требуют передачи в качестве одного из параметров объекта, который реализует интерфейс Comparator. Этот интерфейс определяет единственный метод compare(Object obj1,Object obj2), который на основании определенного пользователем алгоритма сравнивает объекты, переданные в качестве параметров. Метод compare должен вернуть:

```
-1 если obj1 < obj2
0 если obj1 = obj2
1 если obj1 > obj2
```

2.4.4 Класс Arrays

Статический класс Arrays обеспечивает набор методов для выполнения операций над массивами, таких, как поиск, сортировка, сравнение. В Arrays также определен статический метод public List aList(a[] arr), который возвращает список фиксированного размера, основанный на массиве. Изменения в List можно внести, изменив данные в массиве.

```
public class Test {
   public Test() {
   }
   public static void main(String[] args) {
      Test test = new Test();
      String[] arr = {"String 1","String 4",
                      "String 2","String 3"};
      test.dumpArray(arr);
      Arrays.sort(arr);
      test.dumpArray(arr);
      int ind = Arrays.binarySearch(arr,
                  "String 4");
      System.out.println(
        "\nIndex of \"String 4\" = " + ind);
   }
   void dumpArray(String arr[]){
      System.out.println();
      for(int cnt=0;cnt < arr.length;cnt++) {
         System.out.println(arr[cnt]);
      }
   }
}
```

2.4.5 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 (исключающее или).

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

```
public class Test {

   public Test() {
   }
   public static void main(String[] args) {
      Test test = new Test();
      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);
   }
}
```

Результатом будет:

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

Проанализировав первую строку вывода на консоль, можно сделать вывод, что для внутреннего представления BitSet использует значения типа long.

3. Дополнительная литература

1. Вязовик Н.А. **Программирование на JAVA. Курс лекций на intuit.ru**
2. Хорстманн К., Корнелл Г. **Java 2. Том 2. Тонкости программирования**
3. Герберт Шилдт, Джеймс Холмс **Искусство программирования на JAVA**.
4. Патрик Нотон, Герберт Шилдт **Полный справочник по Java**.

Рекумендуется самостоятельно изучить современные средства многопоточного программирования на J2SE 6.0, которые храняться в пакете java.util.concurrent.

4. Порядок выполнения работы

В соответствии с **вариантом** выполните следующее основное задание:

1. Создайте новый проект в среде Eclipse.
2. Создать классы и интерфейсы в соответствии с заданием.
3. Поместить созданные классы и интерфейсы в пакет.
4. Создайте вне пакета управляющий класс, предусмотрев в нем точку входа (main), который будет подключать пакет и осуществлять тестирование разработанных классов и интерфейса.
5. Подготовьте отчет о выполнении лабораторной работы:

Для успешной сдачи лабораторной работы необходимо:

1. представить преподавателю отлаженный код программы для указанного варианта задания;
2. подготовить отчет по работе.

5. Порядок оформления отчета

Отчет о выполнении лабораторной работы должен содержать:

1) титульный лист;
2) задание;
3) текст программы;
4) результаты работы программы.

6. Варианты заданий

Создать программу реализующую работу с объектами типа Collection.

Каждая программа должна иметь **интерфейс** для тестирования основных функций. Задания:

1. Создать программу, реализующую **телефонный справочник** (человек, номера телефонов (домашний, сотовый), адрес) Использовать класс-коллекцию **Hashtable**. Программа должна уметь вводить и выводить данные, осуществлять поиск и сортировку данных по различным поля справочника с использованием интерфейса Comparator.

2. Создать программу реализующую **каталог автомобилей**. (марка, модель, объем двигателя, количество л/с). Использовать класс-коллекцию **ArrayList**. Программа должна уметь вводить и выводить данные, осуществлять поиск и сортировку данных по различным полям с использованием интерфейса Comparator.

3. Создать программу реализующую **каталог товаров** (категория, товар, цена). Использовать класс-коллекцию **LinkedList**. Программа должна уметь вводить и выводить данные, осуществлять поиск и сортировку данных по различным полям с использованием интерфейса Comparator.

4. Создать программу реализующую **список абитуриентов** (ФИО, год поступления, специальность, каждый абитуриент имеет оценки по математике, русскому физике). Использовать класс-коллекцию **Vector**. Программа должна уметь вводить и выводить данные, осуществлять поиск и сортировку данных по различным полям с использованием интерфейса Comparator.

5. Создать программу реализующую **каталог книг** (автор, название, год, издательство). Использовать класс-коллекцию **HashMap**. Программа должна уметь вводить и выводить данные, осуществлять поиск и сортировку данных по различным полям с использованием интерфейса Comparator.

6. Создать программу реализующую **список студентов** (ФИО, номер зачетки, год поступления, группа). Использовать класс-коллекцию **TreeMap**. Программа должна уметь вводить и выводить данные, осуществлять поиск и сортировку данных по различным полям с использованием интерфейса Comparator.

7. Создать программу реализующую **расписание** (группа, предмет, аудитория, преподаватель, время). Использовать класс-коллекцию **HashSet**. Программа должна уметь вводить и выводить данные, осуществлять поиск и сортировку данных по различным полям с использованием интерфейса Comparator.

8. Создать программу реализующую **список задач ОС Windows** (Название процесса, процессорное время, занимаемая память, приоритет). Использовать класс-коллекцию **Queue**.

Программа должна уметь вводить и выводить данные, осуществлять поиск и сортировку данных по различным полям с использованием интерфейса Comparator.

9. Реализовать сравнительный тест для различных реализаций **List** (**LinkedList, ArrayList)** Программа должна выполнять сравнение скорости выполнения операций: запись, чтение, поиск значения по ключу и выводить результаты замеров таблицу. Чтобы адекватно оценить значения скорость выполнения операции нужно выполнить эту операцию много раз (1000, 10000) на различных наборах данных (например, коллекция инкапсулирует int, Time или String , класс из ЛР №2, содержащий несколько полей и тд.). Коллекция должна хранить большое количество элементов (от 1000).

10. Реализовать сравнительный тест для различных реализаций **Set** (**HashSet, TreeSet)** Программа должна выполнять сравнение скорости выполнения операций: запись, чтение, поиск значения по ключу и выводить результаты замеров таблицу. Чтобы адекватно оценить значения скорость выполнения операции нужно выполнить эту операцию много раз (1000, 10000) на различных наборах данных (например, коллекция инкапсулирует int, Time или String , класс из ЛР №2, содержащий несколько полей и тд.). Коллекция должна хранить большое количество элементов (от 1000).

11. Реализовать сравнительный тест для различных реализаций **Map** (**HashMap, TreeMap)** Программа должна выполнять сравнение скорости выполнения операций: запись, чтение, поиск значения по ключу и выводить результаты замеров таблицу. Чтобы адекватно оценить значения скорость выполнения операции нужно выполнить эту операцию много раз (1000, 10000) на различных наборах данных (например, коллекция инкапсулирует float, Time или String , класс из ЛР №2, содержащий несколько полей и тд). Коллекция должна хранить большое количество элементов (от 1000).

12. Реализовать сравнительный тест для **TreeSe**t и **ArrayList** Программа должна выполнять сравнение скорости выполнения операций: запись, чтение, поиск значения по ключу и выводить результаты замеров таблицу. Чтобы адекватно оценить значения скорость выполнения операции нужно выполнить эту операцию много раз (1000, 10000) на различных наборах данных (например, коллекция инкапсулирует double, Time или String , класс из ЛР №2, содержащий несколько полей и тд). Коллекция должна хранить большое количество элементов (от 1000).

13. Реализовать сравнительный тест для **HashMap** и **Vector** Программа должна выполнять сравнение скорости выполнения операций: запись, чтение, поиск значения по ключу и выводить результаты замеров таблицу. Чтобы адекватно оценить значения скорость выполнения операции нужно выполнить эту операцию много раз (1000, 10000) на различных наборах данных (например, коллекция инкапсулирует byte, Time или String , класс из ЛР №2, содержащий несколько полей и тд). Коллекция должна хранить большое количество элементов (от 1000).

14. Реализовать сравнительный тест для **HashSet** и **HashMap** Программа должна выполнять сравнение скорости выполнения операций: запись, чтение, поиск значения по ключу и выводить результаты замеров таблицу. Чтобы адекватно оценить значения скорость выполнения операции нужно выполнить эту операцию много раз (1000, 10000) на различных наборах данных (например, коллекция инкапсулирует boolean, Time или String , класс из ЛР №2, содержащий несколько полей и тд). Коллекция должна хранить большое количество элементов (от 1000).

15. Реализовать сравнительный тест для **TreeSe**t и **TreeMap** Программа должна выполнять сравнение скорости выполнения операций: запись, чтение, поиск значения по ключу и выводить результаты замеров таблицу. Чтобы адекватно оценить значения скорость выполнения операции нужно выполнить эту операцию много раз (1000, 10000) на различных наборах данных (например, коллекция инкапсулирует float, Time или String , класс из ЛР №2,

содержащий несколько полей и тд). Коллекция должна хранить большое количество элементов (от 1000).

16. Реализовать сравнительный тест для **ArrayList** и **Stack** Программа должна выполнять сравнение скорости выполнения операций: запись, чтение, поиск значения по ключу и выводить результаты замеров таблицу. Чтобы адекватно оценить значения скорость выполнения операции нужно выполнить эту операцию много раз (1000, 10000) на различных наборах данных (например, коллекция инкапсулирует long, Time или String , класс из ЛР №2, содержащий несколько полей и тд). Коллекция должна хранить большое количество элементов (от 1000).