Глава 10

КОЛЛЕКЦИИ

Программирование заставило дерево зацвести.

Алан Дж. Перлис

Общие определения

Коллекции — это хранилища или контейнеры, поддерживающие различные способы накопления и упорядочения объектов с целью обеспечения возможностей эффективного доступа к ним. Они представляют собой реализацию абстрактных структур данных, поддерживающих три основные операции:

- добавление нового элемента в коллекцию;
- удаление элемента из коллекции;
- изменение элемента в коллекции.

В качестве других операций могут быть реализованы следующие: заменить, просмотреть элементы, подсчитать их количество и др.

Применение коллекций обусловливается возросшими объемами обрабатываемой информации. Когда счет используемых объектов идет на сотни тысяч, массивы не обеспечивают ни должной скорости, ни экономии ресурсов. Современные информационные системы тестируются на примере электронного магазина, содержащего около 40 тысяч товаров и 125 миллионов клиентов, сделавших 400 миллионов заказов.

Примером коллекции является стек (структура LIFO — Last In First Out), в котором всегда удаляется объект, вставленный последним. Для очереди (структура FIFO — First In First Out) используется другое правило удаления: всегда удаляется элемент, вставляемый первым. В абстрактных типах данных существует несколько видов очередей: двусторонние очереди, кольцевые очереди, обобщенные очереди, в которых запрещены повторяющиеся элементы. Стеки и очереди могут быть реализованы как на базе массива, так и на базе связного списка.

Коллекции в языке Java объединены в библиотеке классов **java.util** и представляют собой контейнеры для хранения и манипулирования объектами. До появления Java 2 эта библиотека содержала классы только для работы с простейшими структурами данных: **Vector**, **Stack**, **Hashtable**, **BitSet**, а также интерфейс **Enumeration** для работы с элементами этих классов. Коллекции, появившиеся в Java 2, представляют собой общую технологию хранения и доступа к объектам. Скорость обработки коллекций повысилась по сравнению с предыдущей

версией языка за счет отказа от их потокобезопасности. Поэтому, если объект коллекции может быть доступен из различных потоков, что наиболее естественно для распределенных приложений, необходимо использовать коллекции из Java 1.

Так как в коллекциях при практическом программировании хранится набор ссылок на объекты одного типа, следует обезопасить коллекцию от появления ссылок на другие, не разрешенные логикой приложения типы. Такие ошибки при использовании нетипизированных коллекций выявляются на стадии выполнения, что повышает трудозатраты на исправление и верификацию кода. Поэтому, начиная с версии Java SE 5, коллекции стали типизированными.

Более удобным стал механизм работы с коллекциями, а именно:

- предварительное сообщение компилятору о типе ссылок, которые будут храниться в коллекции, при этом проверка осуществляется на этапе компиляции;
- отсутствие необходимости постоянно преобразовывать возвращаемые по ссылке объекты (тип **Object**) к требуемому типу.

Структура коллекций характеризует способ, с помощью которого программы Java обрабатывают группы объектов. Так как **Object** — суперкласс для всех классов, то в коллекции можно хранить объекты любого типа, кроме базовых. Коллекции — это динамические массивы, связные списки, деревья, множества, хэш-таблицы, стеки, очереди.

Интерфейсы коллекций:

Map<K, V> — карта отображения вида «ключ-значение»;

Collection<E> — вершина иерархии коллекций **List, Set**;

List<E> — специализирует коллекции для обработки списков;

Set<E> — специализирует коллекции для обработки множеств, содержащих уникальные элементы.

Все классы коллекций реализуют также интерфейсы **Serializable**, **Cloneable** (кроме **WeakHashMap**). Кроме того, классы, реализующие интерфейсы **List<E>** и **Set<E>**, реализуют также интерфейс **Iterable<E>**.

В интерфейсе **Collection<E>** определены методы, которые работают на всех коллекциях:

boolean add(E obj) — добавляет **obj** к вызывающей коллекции и возвращает **true**, если объект добавлен, и **false**, если **obj** уже элемент коллекции;

boolean remove(Object obj) — удаляет **obj** из коллекции;

boolean addAll(Collection<? extends E> c) — добавляет все элементы коллекции к вызывающей коллекции;

void clear() — удаляет все элементы из коллекции;

boolean contains(Object obj) — возвращает **true**, если вызывающая коллекция содержит элемент **obj**;

boolean equals(Object obj) — возвращает **true**, если коллекции эквивалентны;

boolean isEmpty() — возвращает **true**, если коллекция пуста;

Iterator<E> iterator() — извлекает итератор;

int size() — возвращает количество элементов в коллекции;

Object[] toArray() — копирует элементы коллекции в массив объектов;

<T> T[] toArray(T a[]) — копирует элементы коллекции в массив объектов определенного типа.

При работе с элементами коллекции применяются следующие интерфейсы:

Comparator<T>, Comparable<T> — для сравнения объектов;

Iterator<E>, **ListIterator<E>**, **Map.Entry<K, V>** — для перечисления и доступа к объектам коллекции.

Интерфейс **Iterator<E>** используется для построения объекта, который обеспечивает доступ к элементам коллекции. К этому типу относится объект, возвращаемый методом **iterator().** Такой объект позволяет просматривать содержимое коллекции последовательно, элемент за элементом. Позиции итератора располагаются в коллекции между элементами. В коллекции, состоящей из N элементов, существует N+1 позиций итератора.

Методы интерфейса **Iterator<E>:**

boolean hasNext() — проверяет наличие следующего элемента, а в случае его отсутствия (завершения коллекции) возвращает **false**. Итератор при этом остается неизменным;

E next() — возвращает ссылку на объект, на который указывает итератор, и передвигает текущий указатель на следующий, предоставляя доступ к следующему элементу. Если следующий элемент коллекции отсутствует, то метод **next()** генерирует исключение **NoSuchElementException**;

void remove() — удаляет объект, возвращенный последним вызовом метода **next().** Если метод **next()** до вызова **remove()** не вызывался, то будет сгенерировано исключение **IllegalStateException**.

Интерфейс **Map.Entry** предназначен для извлечения ключей и значений карты с помощью методов **K getKey()** и **V getValue()** соответственно. Вызов метода **V setValue(V value)** заменяет значение, ассоциированное с текущим ключом.

Списки

Класс **ArrayList<E>** — динамический массив объектных ссылок. Реализует интерфейсы **List<E>**, **Collection<E>**, **Iterable<E>**.

```
java.util.AbstractCollection<E>
   └ java.util.AbstractList<E>
      └ java.util.ArrayList<E>
```

В классе объявлены конструкторы:

```
ArrayList()
ArrayList(Collection <? extends E> c)
ArrayList(int capacity)
```

Практически все методы класса являются реализацией абстрактных методов из суперклассов и интерфейсов. Методы интерфейса **List<E>** позволяют

вставлять и удалять элементы из позиций, указываемых через отсчитываемый от нуля индекс:

void add(int index, E element) — вставляет **element** в позицию, указанную в **index**;

E remove(int index) — удаляет объект из позиции **index**;

E set(int index, E element) — заменяет объект в позиции **index**, возвращает при этом удаляемый элемент;

void addAll(int index, Collection<? extends E> c) — вставляет в вызывающий список все элементы коллекции **c**, начиная с позиции **index**;

E get(int index) — возвращает элемент в виде объекта из позиции **index**;

int indexOf(Object ob) — возвращает индекс указанного объекта;

List<E> subList(int fromIndex, int toIndex) — извлекает часть коллекции в указанных границах.

Удаление и добавление элементов для такой коллекции представляет собой ресурсоемкую задачу, поэтому объект **ArrayList<E>** лучше всего подходит для хранения списков с малым числом подобных действий. С другой стороны, навигация по списку осуществляется очень быстро, поэтому операции поиска производятся за более короткое время.

/ # 1 # создание параметризованной коллекции # DemoGeneric.java */*

```
package by.bsu.collect;
import java.util.ArrayList;
public class DemoGeneric {
   public static void main(String args[ ]) {
      ArrayList<String> list = new ArrayList<>();
      // ArrayList<int> b = new ArrayList<int>(); // ошибка компиляции
      list.add("Java"); /* компилятор "знает"
      допустимый тип передаваемого значения */
      list.add("JavaFX 2");
      String res = list.get(0); /* компилятор "знает"
      допустимый тип возвращаемого значения */
      // list.add(new StringBuilder("C#")); // ошибка компиляции
      // компилятор не позволит добавить "посторонний" тип
      System.out.print(list); // удобный вывод
   }
}
```

В результате будет выведено:

[Java, JavaFX 2]

В данной ситуации не создается новый класс для каждого конкретного типа и сама коллекция не меняется, просто компилятор снабжается информацией о типе элементов, которые могут храниться в **list**. При этом параметром коллекции может быть только объектный тип.

Следует отметить, что указывать тип следует при создании ссылки, иначе будет позволено добавлять объекты всех типов. На этом основан принцип типобезопасности, обеспечиваемый параметризацией коллекций. Пусть в системе онлайн-торговли используются понятия **Order** и **Item**. Заказы могут собираться в списки, например, список заказов за день. Возможна ситуация, когда по каким-либо причинам в список заказов будет добавлен экземпляр товара. В этой ситуации даже свести баланс действительно сделанных заказов простым определением размера списка будет невозможно. Придется извлекать каждый объект, определять его тип и проч.

/ # 2 # некорректная(raw) и корректная коллекция # UncheckCheckRun.java */*

```
package by.bsu.collection;
import java.util.ArrayList;
public class UncheckCheckRun {
          public static void main(String[ ] args) {
                    ArrayList raw = new ArrayList() { // "сырая" коллекция – raw type
                              { // логический блок анонимного класса
                                        add(new Order(231, 12.f));
                                        add(new Item(23154, 120.f, "Xerox"));
                                        add(new Order(217, 1.7f));
                              }
                    };
                    // при извлечении требуется приведение типов
                    Order or1 = (Order) raw.get(0);
                    Item or2 = (Item) raw.get(1);
                    Order or3 = (Order) raw.get(2);
                    for (Object ob : raw) {
                              System.out.println("raw " + ob);
                    }
                    ArrayList<Order> orders = new ArrayList<Order>() {
                              {
                                        add(new Order(231, 12.f));
                                        add(new Order(389, 2.9f));
                                        add(new Order(217, 1.7f));
                                        // add(new Item(23154, 120.f, "Xerox"));
                                        // ошибка компиляции: список параметризован
                              }
                    };
                    for (Order ob : orders) {
                              System.out.println("Order: " + ob);
                    }
          }
}
```

В результате будет выведено:

raw Order [idOrder=231, amount=12.0]
raw Item [idItem=231, price=12.0, name=Xerox]

raw Order [idOrder=217, amount=1.7]
Order: Order [idOrder=231, amount=12.0]
Order: Order [idOrder=389, amount=2.9]
Order: Order [idOrder=217, amount=1.7]

где классы **Order** и **Item** представляют собой сущности *Заказ* и *Товар* в следующем виде:

/ # 3 # класс используется здесь и далее для наполнения коллекций # Order.java */*

```
package by.bsu.collection;
public class Order {
          private int orderId;
          private float amount;
          // поля и методы описания подробностей заказа
          public Order(int orderId, float amount) {
                    this.orderId = orderId;
                    this.amount = amount;
          }
          public int getOrderId () {
                    return orderId;
          }
          public float getAmount() {
                    return amount;
          }
          @Override
          public String toString() {
                    return "Order [orderId =" + orderId + ", amount=" + amount + "]";
          }
}
```

/ # 4 # класс используется здесь и далее для наполнения коллекций # Item.java */*

```
package by.bsu.collection;
public class Item {
          private int itemId;
          private float price;
          private String name;
          public Item(int itemId, float price, String name) {
                    this itemId = itemId;
                    this.price = price;
                    this.name = name;
          }
          public int getItemId () {
                    return itemId;
          }
          public float getPrice() {
                    return price;
          }
          public String getName() {
```

```
                return name;
        }
        @Override
        public String toString() {
                return "Item [itemId =" + itemId + ", price=" + price + ", name=" + name + "]\n";
        }
}
```

Чтобы параметризация коллекции была полной, необходимо указывать параметр и при объявлении ссылки, и при создании объекта.

Интерфейс **Iterator** используется для последовательного доступа к элементам коллекции. Ниже приведен пример поиска с помощью итератора заказов, сумма которых превышает заданную.

/ # 5 # методы класса ArrayList и интерфейса Iterator # RunIterator.java # FindOrder.java*/*

```
package by.bsu.collection;
import java.util.Iterator;
import java.util.List;
import java.util.ArrayList;
public class RunIterator {
        public static void main(String[ ] args) {
                ArrayList<Order> orders = new ArrayList<Order>() {
                        {
                                add(new Order(231, 12.f));
                                add(new Order(389, 2.9f));
                                add(new Order(217, 1.7f));
                        }
                };
                FindOrder fo = new FindOrder();
                List<Order> res = fo.findBiggerAmountOrder(10f, orders);
                System.out.println(res);
        }
}
package by.bsu.collection;
import java.util.List;
import java.util.Iterator;
public class FindOrder {
        public List<Order> findBiggerAmountOrder(float bigAmount, List<Order> orders) {
                ArrayList<Order> bigPrices = new ArrayList();
                Iterator<Order> it = orders.iterator();
                while (it.hasNext()) {
                        Order current = it.next();
                        if(current.getAmount() >= bigAmount) {
                                bigPrices.add(current);
                        }
                }
        return bigPrices;
        }
}
```

В результате на консоль будет выведено:

[Order [idOrder=231, amount=12.0]]

Метасимвол в коллекциях

Метасимволы используются при параметризации коллекций для расширения возможностей самой коллекции и обеспечения ее типобезопасности. Например, если параметр метода предыдущего примера изменить с **List<Order>** на **List<? extends Order>**, то в метод можно будет передавать коллекции, параметризованные любым допустимым типом, а именно классом **Order** и любым его подклассом, что невозможно при записи без анонимного символа.

Но в методе нельзя будет добавить к коллекции новый элемент, пусть даже и допустимого типа, так как компилятору неизвестен заранее тип параметризации списка.

```
List<Order> findBiggerAmountOrder(float bigAmount, List<? extends Order> orders) {
          // orders.add(new Order(231, 12.f)); // ошибка компиляции
          orders.remove(0); // удалять можно
          // some code here
}
```

Объясняется это тем, что список, передаваемый в качестве параметра метода, может быть параметризован классом **DiscountOrder**, а в методе будет совершаться попытка добавления экземпляра самого класса **Order**, как показано выше, что недопустимо определением самой параметризации передаваемого в метод списка, а именно:

```
findBiggerAmountOrder(10.f, new ArrayList<DiscountOrder>());
```

где

```
public class DiscountOrder extends Order {
          // поля
          public DiscountOrder(int idOrder, float amount) {
                    super(idOrder, amount);
          }
          // методы
}
```

Поэтому добавление к спискам, параметризованным метасимволом с применением **extends**, запрещено всегда.

Параметризация **List<? super Order>** утверждает, что параметр метода или возвращаемое значение может получить список типа **Order** или любого из его суперклассов, в то же время разрешает добавлять туда экземпляры класса **Order** и любых его подклассов.

```
List<Order> findBiggerAmountOrder(float bigAmount, List<? super Order> orders) {
                    orders.add(new Order(231, 12.f));
                    // some code here
}
```

Или, если использовать **super** для определения типа возвращаемого значения:

```
List<? super Order> findBiggerAmountOrder(float bigAmount, List<? extends Order> orders) {
        // some code here
}
```

В этом случае к списку, возвращенному методом, можно будет добавлять экземпляры класса **Order** и его подклассов.

Интерфейс ListIterator

Для доступа к элементам списка может также использоваться интерфейс **ListIterator<E>**, который позволяет получить доступ сразу в необходимую программисту позицию списка вызовом метода **listIterator(int index).** Интерфейс **ListIterator<E>** расширяет интерфейс **Iterator<E>** и предназначен для обработки списков и их вариаций. Наличие методов **E previous(), int previousIndex()** и **boolean hasPrevious()** обеспечивает обратную навигацию по списку. Метод **int nextIndex()** возвращает номер следующего итератора. Метод **void add(E obj)** позволяет вставлять элемент в список текущей позиции. Вызов метода **void set(E obj)** производит замену текущего элемента списка на объект, передаваемый методу в качестве параметра.

При внесении изменений в коллекцию после извлечения итератора гарантирует генерацию исключения **java.util.ConcurrentModificationException** при попытке использовать итератор позднее.

```
ArrayList<Order> orders = new ArrayList<>();
// добавление элементов
Iterator<Order> it = orders.iterator();
orders.add(new Order(12, 12.1f)); // или remove(0);
while (it.hasNext()) {
        System.out.println(it.next());
}
```

Параллельная или конкурентная модификация коллекции различными активностями (самой коллекцией и ее итератором) приводит к неразрешимой проблеме и заканчивается генерацией исключения практически всегда. Проблема заключается в том, что итератор извлек N число позиций, а коллекция изменила число своих экземпляров и итератор перестал соответствовать коллекции. Если модификацию коллекции и работу с итератором нужно выполнять из различных потоков, то для решения такой задачи используются ограниченно потокобезопасные решения для коллекций из пакета **java.util.concurrent**.

При создании класса, содержащего коллекцию элементов, возможны два способа: агрегация коллекции в качестве поля класса или отношение **HAS-A** и наследование от класса, представляющего коллекцию или отношение **IS-A**. Последнее встречается значительно реже.

/ # 6 # класс, агрегирующий список, с реализацией интерфейса Iterable # Order.java*/*

```
import java.util.Iterator;
import java.util.List;
import java.util.Collections;
public class Order implements Iterable<Item> {
          private int orderId;
          private List<Item> listItems;
          // private float amount; // не нужен, т.к. сумму можно вычислить
          public Order(int orderId, List<Item> listItems) {
                    this.orderId = orderId;
                    this.listItems = listItems;
          }
          public int getOrderId () {
                    return orderId;
          }
          public List<Item> getListItems() {
                    return Collections.unmodifiableList(listItems);
          }
          // некоторые делегированные методы интерфейсов List и Collection
          public boolean add(Item e) {
                    return listItems.add(e);
          }
          public Item get(int index) {
                    return listItems.get(index);
          }
          public Item remove(int index) {
                    return listItems.remove(index);
          }
          // создание итератора
          @Override
          public Iterator<Item> iterator() {
                    return listItems.iterator();
          }
}
```

Но можно эту задачу решить еще проще, унаследовав класс коллекции, например, **ArrayList**:

/ # 7 # класс, наследующий список и приобретающий его свойства # Order.java */*

```
package by.bsu.collection;
import java.util.ArrayList;
public class Order extends ArrayList<Item> {
          private int orderId;
          // private float amount; // по прежнему не нужен, т.к. сумму можно вычислить
          public Order(ArrayList<Item> c) {
                    super(c);
          }
          public Order(int orderId, ArrayList<? extends Item> c) {
                    super(c);
```

```
                this.orderId = orderId;
        }
        public int getOrderId() {
                return orderId;
        }
        public void setOrderId (int orderId) {
                this. orderId = orderId;
        }
}
```

При такой реализации нет явной необходимости в переопределении метода класса **ArrayList**, так как можно просто воспользоваться его стандартными методами.

Интерфейс Comparator

Реализация метода **equals()** класса **Object** предоставляет возможность проверить, эквивалентен один экземпляр другому или нет. На практике возникает необходимость сравнения объектов на больше/меньше либо равно.

При реализации интерфейса **java.util.Comparator<T>** появляется возможность сортировки списка объектов конкретного типа по правилам, определенным для этого типа. Контракт интерфейса подразумевает реализацию метода **int compare(T ob1, T ob2),** принимающего в качестве параметров два объекта, для которых должно быть определено возвращаемое целое значение, знак которого и определяет правило сортировки. Метод **public abstract boolean equals(Object obj),** также объявленный в интерфейсе **Comparator<T>**, очень рекомендуется переопределять, если экземпляр класса будет использоваться для хранения информации. Это необходимо для исключения противоречивой ситуации, когда для двух объектов метод **compare()** возвращает **0**, т. е. сообщает об их эквивалентности, в то же время метод **equals()** для этих же объектов возвращает **false**, так как данный метод не был никем определен и была использована его версия из класса **Object**. Кроме того, наличие метода **equals()** обеспечивает корректную работу метода семантического поиска и проверки на идентичность **contains(Object o),** определенного в интерфейсе **java.util.Collection**, а следовательно, реализованного в любой коллекции.

Метод **compare()** автоматически вызывается при сортировке списка методом: **static <T> void sort(List<T> list, Comparator<? super T> c)** класса **Collections**, в качестве первого параметра принимающий коллекцию, в качестве второго — объект-comparator, из которого извлекается и применяется правило сортировки.

С помощью анонимного типа можно привести простейшую реализацию компаратора.

/ # 8 # сортировка списка по полю, определенному в классе comparator-е # SortItemRunner.java */*

```
package by.bsu.comparison;
import java.util.ArrayList;
```

```
import java.util.Collections;
import java.util.Comparator;
import by.bsu.collection.Item;
public class SortItemRunner {
          public static void main(String[ ] args) {
                    ArrayList<Item> p = new ArrayList<Item>() {
                              {
                                        add(new Item(52201, 9.75f, "T-Shirt"));
                                        add(new Item(52127, 13.99f, "Dress"));
                                        add(new Item(47063, 45.95f, "Jeans"));
                                        add(new Item(90428, 60.9f, "Gloves"));
                                        add(new Item(53295, 31f, "Shirt"));
                                        add(new Item(63220, 14.9f, "Tie"));
                              }
                    };
                    // создание компаратора
                    Comparator<Item> comp = new Comparator<Item>() {
                              // сравнение для сортировки по убыванию цены товара
                              public int compare(Item one, Item two) {
                                        return Double.compare(two.getPrice(), one.getPrice());
                              }
                              // public boolean equals(Object ob) { /* реализация */ }
                    };
                    // сортировка списка объектов
                    Collections.sort(p, comp);
                    System.out.println(p);
          }
}
```

В результате будет выведено:

[Item [itemId=90428, price=60.9, name=Gloves]
, Item [itemId =47063, price=45.95, name=Jeans]
, Item [itemId =53295, price=31.0, name=Shirt]
, Item [itemId =63220, price=14.9, name=Tie]
, Item [itemId =52127, price=13.99, name=Dress]
, Item [itemId =52201, price=9.75, name=T-Shirt]
]

Если в процессе использования необходимы сортировки по различным полям класса, то реализацию компаратора следует вынести в отдельный класс.

Для обеспечения возможности сортировки по любому полю класса **Item** следует создать перечисление со значениями, соответствующими полям этого класса

```
public enum ItemEnum {
          ITEM_ID, PRICE, NAME
}
```

и отдельный класс, реализующий параметризованный интерфейс **Comparator**.

/ # 9 # возможность сортировки по всем полям класса # ItemComparator.java */*

```
package by.bsu.collection;
import java.util.Comparator;
public class ItemComparator implements Comparator<Item> {
          private ItemEnum sortingIndex;
          public ItemComparator(ItemEnum sortingIndex) {
             setSortingIndex(sortingIndex);
          }
          public final void setSortingIndex(ItemEnum sortingIndex) {
             if (sortingIndex == null) {
                throw new IllegalArgumentException();
             }
             this.sortingIndex = sortingIndex;
          }
          public ItemEnum getSortingIndex() {
             return sortingIndex;
          }
          @Override
          public int compare(Item one, Item two) {
             switch (sortingIndex) {
             case ITEM_ID:
                return one.getItemId() - two.getItemId();
             case PRICE:
                return Double.compare(two.getPrice() - one.getPrice());
             case NAME:
                return one.getName().compareTo(two.getName());
             default:
                throw new EnumConstantNotPresentException(ItemEnum.class,sortingIndex.name());
             }
          }
}
```

При необходимости сортировки по полю **itemId** следует изменить значение статической переменной **sortingIndex** и в качестве второго параметра методу **sort()** передать объект класса **ItemComparator**:

```
Collections.sort (p, new ItemComparator (ItemEnum.ITEM_ID));
```

Достаточно легко реализовать возможность сортировки по возрастанию и убыванию для каждого из полей класса, добавив в перечисление, например, поле **ascend** типа **boolean**, от значения которого поставить в зависимость знак числа, возвращаемого методом **compare().** Перечисление **ItemEnum** будет выглядеть следующим образом:

/ # 10 # перечисление, предоставляющее возможность сортировки по убыванию\ возрастанию для всех полей класса # FullItemEnum.java # */*

```
package by.bsu.collection;
public enum FullItemEnum {
          ITEM_ID(true), PRICE(false), NAME(true);
```

```
        private boolean ascend;
        private FullItemEnum(boolean ascend) {
                this.ascend = ascend;
        }
        public void setAscend(boolean ascend) {
                this.ascend = ascend;
        }
        public boolean getAscend() {
                return ascend;
        }
}
```

Класс **ItemComparator** также следует переписать, так как с учетом новых возможностей число кодов блоков **case** увеличится.

Класс-компаратор, являясь логической частью класса-сущности, может быть его частью и представлен в виде статического внутреннего класса

/ # 11 # класс-сущность с внутренними классами-компараторами # Item.java # */*

```
package by.bsu.collection;
import java.util.Comparator;
public class Item {
        private int itemId;
        private float price;
        private String name;
        public Item(int itemId, float price, String name) {
                super();
                this.itemId = itemId;
                this.price = price;
                this.name = name;
        }
        public int getItemId() {
                return itemId;
        }
        public float getPrice() {
                return price;
        }
        public String getName() {
                return name;
        }
        public static class IdComparator implements Comparator<Item> {
                @Override
                public int compare(Item one, Item two) {
                        return one.getItemId() - two.getItemId();
                }
        }
        public static class PriceComparator implements Comparator<Item> {
                @Override
                public int compare(Item one, Item two) {
```

```
                    return Double.compare(two.getPrice(), one.getPrice());
                }
        }
}
```

Экземпляр компаратора создается обычным для внутренних классов способом

```
Item.IdComparator comp = new Item.IdComparator();
```

Объявление внутри класса позволяет манипулировать доступом к его функциональности.

Интерфейс **Comparator** не рекомендуется имплементировать классу-сущности, для сортировки экземпляров которого он предназначается.

Класс LinkedList и интерфейс Queue

Коллекция **LinkedList<E>** реализует связанный список.

```
java.util.AbstractCollection<E>
      └ java.util.AbstractList<E>
          └ java.util.AbstractSequentialList<E>
              └ java.util.LinkedList<E>
```

Реализует кроме интерфейсов, указанных при описании **ArrayList**, также интерфейсы **Queue<E>** и **Deque<E>**.

Связанный список хранит ссылки на объекты отдельно вместе со ссылками на следующее и предыдущее звенья последовательности, поэтому часто называется двунаправленным списком. Операции добавления и удаления выполняются достаточно быстро, в отличие от операций поиска и навигации.

В этом классе объявлены методы, позволяющие манипулировать им как очередью, двунаправленной очередью и т. д. Двунаправленный список кроме обычного имеет особый «нисходящий» итератор, позволяющий двигаться от конца списка к началу, и извлекается методом **descendingIterator().**

Для манипуляций с первым и последним элементами списка в **LinkedList<E>** реализованы методы:

void addFirst(E ob), void addLast(E ob) — добавляющие элементы в начало и конец списка;

E getFirst(), E getLast() — извлекающие элементы;

E removeFirst(), E removeLast() — удаляющие и извлекающие элементы;

E removeLastOccurrence(E elem), E removeFirstOccurrence(E elem) — удаляющие и извлекающие элемент, первый или последний раз встречаемый в списке.

Класс **LinkedList<E>** реализует интерфейс **Queue<E>**, что позволяет списку придать свойства очереди. В компьютерных науках очередь — структура данных, в основе которой лежит принцип FIFO (first in, first out). Элементы добавляются в конец и вынимаются из начала очереди. Но существует возможность

не только добавлять и удалять элементы, также можно просмотреть, что находится в очереди. К тому же специализированные методы интерфейса **Queue<E>** по манипуляции первым и последним элементами такого списка **E element(), boolean offer(E o), E peek(), E poll(), E remove()** работают немного быстрее, чем соответствующие методы класса **LinkedList<E>**.

Методы интерфейса **Queue<E>**:

boolean add(E o) — вставляет элемент в очередь, если же очередь полностью заполнена, то генерирует исключение **IllegalStateException**;

boolean offer(E o) — вставляет элемент в очередь, если возможно;

E element() — возвращает, но не удаляет головной элемент очереди;

E peek() — возвращает, но не удаляет головной элемент очереди, возвращает **null**, если очередь пуста;

E poll() — возвращает и удаляет головной элемент очереди, возвращает **null**, если очередь пуста;

E remove() — возвращает и удаляет головной элемент очереди.

Методы **element()** и **remove()** отличаются от методов **peek()** и **poll()** тем, что генерируют исключение **NoSuchElementException**, если очередь пуста.

```
Queue<Item> queue = new LinkedList<>();
```

Создается очередь простым присваиванием связанного списка ссылке типа **Queue**.

/ # 12 # двунаправленный список и очередь # OrderQueueAction.java */*

```
package by.bsu.collection;
import java.util.LinkedList;
import java.util.Queue;
public class OrderQueueAction {
          public static void main(String[ ] args) {
                    LinkedList<Order> orders = new LinkedList<Order>() {
                              {
                                        add(new Order(231, 12.f));
                                        add(new Order(389, 2.9f));
                                        add(new Order(217, 1.7f));
                              }
                    };
                    Queue<Order> queueA = orders; // создание очереди
                    queueA.offer(new Order(222, 9.7f)); // элемент не добавится
                    orderProcessing(queueA); // обработка очереди
                    if (queueA.isEmpty()) {
                              System.out.println("Queue of Orders is empty");
                    }
          }
          public static void orderProcessing(Queue<Order> queue) { // заменить void -> boolean
                    Order ob = null;
                    // заменить while -> do{} while
                    while ((ob = queue.poll()) != null) { // извлечение с удалением
```

```
                    System.out.println("Order #" + ob.getIdOrder() + " is processing");
                    // verifying and processing
                }
        }
}
```

В результате будет выведено:

Order #231 is processing
Order #389 is processing
Order #217 is processing
Queue of Orders is empty

При всей схожести списков **ArrayList** и **LinkedList** существуют серьезные отличия, которые необходимо учитывать при использовании коллекций в конкретных задачах. Если необходимо осуществлять быструю навигацию по списку, то следует применять **ArrayList**, так как перебор элементов в **LinkedList** осуществляется на порядок медленнее. С другой стороны, если требуется часто добавлять и удалять элементы из списка, то уже класс **LinkedList** обеспечивает значительно более высокую скорость переиндексации. То есть если коллекция формируется в начале процесса и в дальнейшем используется только для доступа к информации, то применяется **ArrayList**, если же коллекция подвергается изменениям на всем протяжении функционирования приложения, то выгоднее **LinkedList**.

Интерфейс **Queue<E>** также реализует класс **PriorityQueue<E>**.

```
java.util.AbstractCollection<E>
      └ java.util.AbstractQueue<E>
          └ java.util.PriorityQueue<E>
```

В такую очередь элементы добавляются не в порядке «живой очереди», а в порядке, определяемом в классе, экземпляры которого содержатся в очереди. Сам порядок следования задан реализацией интерфейсов **Comparator<E>** или **Comparable<E>**.

По умолчанию компаратор размещает элементы в очереди в порядке возрастания. Если порядок в классе не определен, а именно, не реализован ни один из указанных интерфейсов, то генерируется исключительная ситуация **ClassCastException** при добавлении второго элемента в очередь. При добавлении первого элемента компаратор не вызывается.

```
PriorityQueue<Order> orders = new PriorityQueue();
orders.add(new Order(546, 53.f)); // нормально
orders.add(new Order(146, 13.f)); // ошибка времени выполнения
```

С другой стороны класс **String** реализует интерфейс **Comparable**:

```
PriorityQueue<String> q = new PriorityQueue<String>();
orders.add(new String("Oracle")); // нормально
orders.add(new String("Google")); // нормально
```

Если попытаться заменить тип **String** на **StringBuilder** или **StringBuffer**, то создать очередь **PriorityQueue** так просто не удастся, как и в случае добавления объектов класса **Order**. Решением такой задачи будет создание нового класса-оболочки с полем типа **StringBuilder** или **Order** и реализацией интерфейса **Comparable<T>**.

/ # 13 # пользовательский класс, обьект которого может быть добавлен в очередь PriorityQueue и множество TreeSet # Order.java */*

```
package by.bsu.collection;
public class Order implements Comparable<Order> {
          private int orderId;
          private float amount;
          // поля и методы описания подробностей заказа
          public Order(int orderId, float amount) {
                    super();
                    this.orderId = orderId;
                    this.amount = amount;
          }
          public int getOrderId() {
                    return orderId;
          }
          public float getAmount() {
                    return amount;
          }
          // реализация метода интерфейса Comparable
          @Override
          public int compareTo(Order ob) {
                    return this.orderId - ob.orderId;
          }
          @Override
          public String toString() {
                    return "Order [orderId =" + orderId + ", amount=" + amount + "]";
          }
}
```

Предлагаемое решение универсально для любых пользовательских типов. Предложенное решение применимо для создания упорядоченных множеств вида **TreeSet**, которые используют компаратор для сортировки при добавлении экземпляра в множество.

Интерфейс Deque и класс ArrayDeque

Интерфейс **Deque** определяет «двунаправленную» очередь и, соответственно, методы доступа к первому и последнему элементам двусторонней очереди. Реализацию этого интерфейса можно использовать для моделирования стека. Методы обеспечивают удаление, вставку и обработку элементов. Каждый из этих

методов существует в двух формах. Одни методы создают исключительную ситуацию в случае неудачного завершения, другие возвращают какое-либо из значений (**null** или **false** в зависимости от типа операции). Вторая форма добавления элементов в очередь сделана специально для реализаций **Deque**, имеющих ограничение по размеру. В большинстве реализаций операции добавления заканчиваются успешно. Методы **addFirst(), addLast()** вставляют элементы в начало и в конец очереди соответственно. Метод **add()** унаследован от интерфейса **Queue** и абсолютно аналогичен методу **addLast()** интерфейса **Deque**. Объявить двуконечную очередь на основе связанного списка можно, например, следующим образом:

```
Deque<String> dq = new LinkedList<>();
```

Класс **ArrayDeque** быстрее, чем **Stack**, если используется как стек, и быстрее, чем **LinkedList<E>,** если используется в качестве очереди.

Множества

Интерфейс **Set<E>** объявляет поведение коллекции, не допускающей дублирования элементов. Интерфейс **SortedSet<E>** наследует **Set<E>** и объявляет поведение набора, отсортированного в возрастающем порядке, заранее определенном для класса. Интерфейс **NavigableSet<E>** существенно облегчает поиск элементов, например, расположенных рядом с заданным.

Класс **HashSet<E>** наследуется от абстрактного суперкласса **AbstractSet<E>** и реализует интерфейс **Set<E>**, используя хэш-таблицу для хранения коллекции.

```
java.util.AbstractCollection<E>
      └ java.util.AbstractSet<E>
          └ java.util.HashSet<E>
```

Ключ (хэш-код) используется в качестве индекса хэш-таблицы для доступа к объектам множества, что значительно ускоряет процессы поиска, добавления и извлечения элемента. Скорость указанных процессов становится заметной для коллекций с большим количеством элементов. Множество **HashSet** не является сортированным. В таком множестве могут храниться элементы с одинаковыми хэш-кодами в случае, если эти элементы не эквивалентны при сравнении. Для грамотной организации **HashSet** следует следить, чтобы реализации методов **hashCode()** и **equals()** соответствовали контракту.

Конструкторы класса:

```
HashSet()
HashSet(Collection <? extends E> c)
HashSet(int capacity)
HashSet(int capacity, float loadFactor),
```

где **capacity** — число ячеек для хранения хэш-кодов.

/ # 14 # использование множества для вывода всех уникальных слов из файла # DemoHashSet.java */*

```
package by.bsu.set;
import java.io.*;
import java.util.*;
public class DemoHashSet {
          public static void main(String[ ] args) {
                    HashSet<String> words = new HashSet<>(100);
                    long callTime = System.nanoTime();
                    Scanner scan = null;
                    try {
                       scan = new Scanner(new File("texts\\nabokov.txt"));
                              scan.useDelimiter("[^A-я]+");
                              while (scan.hasNext()) {
                                        String word = scan.next();
                                        words.add(word.toLowerCase());
                              }
                    } catch (FileNotFoundException e) {
                              e.printStackTrace();
                    }
                    Iterator<String> it = words.iterator();
                    while (it.hasNext()) {
                              System.out.println(it.next());
                    }
                    TreeSet<String> ts = new TreeSet<>(words);
                    System.out.println(ts);
                    long totalTime = System.nanoTime() - callTime;
                    System.out.println("различных слов: " + words.size() + ", "
                              + totalTime + " наносекунд");
          }
}
```

Класс **TreeSet<E>** для хранения объектов использует бинарное дерево.

```
java.util.AbstractCollection<E>
      └ java.util.AbstractSet<E>
          └ java.util.TreeSet<E>
```

При добавлении объекта в дерево он сразу же размещается в необходимую позицию с учетом сортировки. Сортировка происходит благодаря тому, что класс реализует интерфейс **SortedSet**, где правило сортировки добавляемых элементов определяется в самом классе, сохраняемом в множестве, который в большинстве случаев реализует интерфейс **Comparable**. Обработка операций удаления и вставки объектов происходит медленнее, чем в хэш-множествах, но быстрее, чем в списках.

Конструкторы класса:

```
TreeSet()
TreeSet(Collection <? extends E> c)
TreeSet(Comparator <? super E> c)
TreeSet(SortedSet <E> s)
```

Класс **TreeSet<E>** содержит методы по извлечению первого и последнего (наименьшего и наибольшего) элементов **E first()** и **E last().** Методы **subSet(E from, E to)**, **tailSet(E from)** и **headSet(E to)** предназначены для извлечения определенной части множества. Метод **Comparator <? super E> comparator()** возвращает объект **Comparator**, используемый для сортировки объектов множества или **null**, если выполняется обычная сортировка.

/ # 15 # создание множества из списка и его методы # DemoTreeSet.java */*

```
package by.bsu.set;
import java.util.*;
public class DemoTreeSet {
          public static void main(String[ ] args) {
                    ArrayList<String> list = new ArrayList<>();
                    boolean b;
                    for (int i = 0; i < 6; i++) {
                              list.add((int) (Math.random() * 10) + "Y ");
                    }
                    System.out.println(list + "список");
                    TreeSet<String> set = new TreeSet(list);
                    System.out.println(set + "множество");

                    System.out.println(set.comparator()); // null – String реализует Comparable

                    // извлечение наибольшего и наименьшего элементов
                    System.out.println(set.last() + " " + set.first());

                    HashSet<String> hSet = new HashSet<>(set);
                    for (String str : hSet) {
                              System.out.println(str + "    " + str.hashCode());
                    }
          }
}
```

В результате может быть выведено:

[6Y , 5Y , 2Y , 5Y , 4Y , 8Y]список
[2Y , 4Y , 5Y , 6Y , 8Y]множество
null
8Y 2Y
2Y 50841
6Y 54685
8Y 56607
4Y 52763
5Y 53724

Множество инициализируется списком и сортируется сразу же в процессе создания. С помощью итератора элемент может быть найден и удален из множества. Для множества, состоящего из обычных строк, используется лексикографическая

сортировка, задаваемая реализацией интерфейса **Comparable**, поэтому метод **comparator()** возвращает **null**.

Абстрактный класс **EnumSet<E extends Enum<E>>** наследуется от абстрактного класса **AbstractSet**.

```
java.util.AbstractCollection<E>
      └ java.util.AbstractSet<E>
         └ java.util.EnumSet<E>
```

Класс специально реализован для работы с типами **enum**. Все элементы такой коллекции должны принадлежать единственному типу **enum**, определенному явно или неявно. Внутренне множество представимо в виде вектора битов, обычно единственного **long**. Множества нумераторов поддерживают перебор по диапазону из нумераторов. Скорость выполнения операций над таким множеством очень высока, даже если в ней участвует большое количество элементов.

Создать объект этого класса можно только с помощью статических методов. Метод **EnumSet<E> noneOf(Class<E> elemType)** создает пустое множество нумерованых констант с указанным типом элемента, метод **allOf(Class<E> elementType)** создает множество нумерованых констант, содержащее все элементы указанного типа. Метод **of(E first, E… rest)** создает множество, первоначально содержащее указанные элементы. С помощью метода **complementOf(EnumSet<E> s)** создается множество, содержащее все элементы, которые отсутствуют в указанном множестве. Метод **range(E from, E to)** создает множество из элементов, содержащихся в диапазоне, определенном двумя элементами. При передаче указанным методам в качестве параметра **null** будет сгенерирована исключительная ситуация **NullPointerException**.

/ # 16 # использование множества enum-типов # CarManufacturer.java # UseEnumSet.java */*

```
package by.bsu.set;
public enum CarManufacturer {
   AUDI, BMW, VW, TOYOTA, HONDA, ISUZU, SUZUKI, VOLVO, RENAULT
}
package by.bsu.set;
import java.util.EnumSet;
public class UseEnumSet {
   public static void main(String[ ] args) {
      /* множество japanAuto содержит элементы типа
      enum из интервала, определенного двумя элементами */
      EnumSet <CarManufacturer> japanAuto =
      EnumSet.range(CarManufacturer.TOYOTA, CarManufacturer.SUZUKI);
/* множество other будет содержать все элементы, не содержащиеся в множестве japanAuto */
                  EnumSet <CarManufacturer> other = EnumSet.complementOf(japanAuto);
                  System.out.println(japanAuto);
                  System.out.println(other);
                           action("audi", japanAuto);
```

```
                    action("suzuki", japanAuto);
            }
public static boolean action(String auto, EnumSet <CarManufacturer> set) {
   CarManufacturer cm = CarManufacturer.valueOf(auto.toUpperCase());
   boolean ok = false;
   if(ok = set.contains(cm)) {
   // обработка
   System.out.println("Обработан: " + cm);
    } else {
    System.out.println("Обработка невозможна: " + cm);
    }
    return ok;
    }
}
```

В результате будет выведено:

[TOYOTA, HONDA, ISUZU, SUZUKI]
[AUDI, BMW, VW, VOLVO, RENAULT]
Обработка невозможна: AUDI
Обработан: SUZUKI

Карты отображений

Карта отображений — это объект, который хранит пару «ключ–значение». Поиск объекта (значения) облегчается по сравнению с множествами за счет того, что его можно найти по его уникальному ключу. Уникальность объектов-ключей должна обеспечиваться переопределением методов **hashCode()** и **equals()** или реализацией интерфейсов **Comparable**, **Comparator** пользовательским классом. Классы карт отображений:

AbstractMap<K, V> — реализует интерфейс **Map<K, V>**, является суперклассом для всех перечисленных карт отображений;

HashMap<K, V> — использует хэш-таблицу для работы с ключами;

TreeMap<K, V> — использует дерево, где ключи расположены в виде дерева поиска в определенном порядке;

WeakHashMap<K, V> — позволяет механизму сборки мусора удалять из карты значения по ключу, ссылка на который вышла из области видимости приложения;

LinkedHashMap<K, V> — образует дважды связанный список ключей. Этот механизм эффективен, только если превышен коэффициент загруженности карты при работе с кэш-памятью и др.

Для класса **IdentityHashMap<K, V>** хэш-коды объектов-ключей вычисляются методом **System.identityHashCode()** по адресу объекта в памяти в отличие от обычного значения **hashCode(),** вычисляемого сугубо по содержимому самого объекта.

Интерфейсы карт:

Map<K, V> — отображает уникальные ключи и значения;

Map.Entry<K, V> — описывает пару «ключ–значение»;

SortedMap<K, V> — содержит отсортированные ключи и значения;

NavigableMap<K, V> — добавляет новые возможности навигации и поиска по ключу.

Интерфейс **Map<K, V>** содержит следующие методы:

V get(Object obj) — возвращает значение, связанное с ключом **obj.** Если элемент с указанным ключом отсутствует в карте, то возвращается значение **null**;

V put(K key, V value) — помещает ключ **key** и значение **value** в вызывающую карту. При добавлении в карту элемента с существующим ключом произойдет замена текущего элемента новым. При этом метод возвратит заменяемый элемент;

void putAll(Map <? extends K,? extends V> t) — помещает коллекцию **t** в вызывающую карту;

V remove(Object key) — удаляет пару «ключ–значение» по ключу **key**;

void clear() — удаляет все пары из вызываемой карты;

boolean containsKey(Object key) — возвращает **true**, если вызывающая карта содержит **key** как ключ;

boolean containsValue(Object value) — возвращает **true**, если вызывающая карта содержит **value** как значение;

Set<K> keySet() — возвращает множество ключей;

Set<Map.Entry<K, V>> entrySet() — возвращает множество, содержащее значения карты в виде пар «ключ–значение»;

Collection<V> values() — возвращает коллекцию, содержащую значения карты.

В коллекциях, возвращаемых тремя последними методами, можно только удалять элементы, добавлять нельзя. Данное ограничение обуславливается параметризацией возвращаемого методами значения.

Интерфейс **Map.Entry<K, V>** представляет пару «ключ–значение» и содержит следующие методы:

K getKey() — возвращает ключ текущего входа;

V getValue() — возвращает значение текущего входа;

V setValue(V obj) — устанавливает значение объекта **obj** в текущем входе.

В примере показаны способы создания хэш-карты и доступа к ее элементам.

/ # 17 # создание хэш-карты и замена элемента по ключу # DemoHashMap.java */*

```
package by.bsu.maps;
public class DemoHashMap {
          public static void main(String[] args) {
                    HashMap<String, Integer> hm = new HashMap<String, Integer>(3) {
                              {
                                        this.put("Сырок", 3);
                                        this.put("Пряник", 5);
```

```
                                        this.put("Молоко", 1);
                                        this.put("Хлеб", 1);
                                }
                        };
                        System.out.println(hm);
                        hm.put("Пряник", 4); // замена или добавление при отсутствии ключа
                        System.out.println(hm + "после замены элемента");
                        Integer a = hm.get("Хлеб");
                        System.out.println(a + " - найден по ключу 'Хлеб'");
                        // вывод хэш-таблицы с помощью методов интерфейса Map.Entry<K,V>
                        Set<Map.Entry<String, Integer>> setv = hm.entrySet();
                        System.out.println(setv);
                        Iterator<Map.Entry<String, Integer>> i = setv.iterator();
                        while (i.hasNext()) {
                                Map.Entry<String, Integer> me = i.next();
                                System.out.println(me.getKey() + " : " + me.getValue());
                        }
                        Set<Integer> val = new HashSet<Integer>(hm.values());
                        System.out.println(val);
        }
}
```

{Пряник=5, Хлеб=1, Сырок=3, Молоко=1}
{Пряник=4, Хлеб=1, Сырок=3, Молоко=1}после замены элемента
1 - найден по ключу 'Хлеб'
[Пряник=4, Хлеб=1, Сырок=3, Молоко=1]
Пряник : 4
Хлеб : 1
Сырок : 3
Молоко : 1
[1, 3, 4]

Метод **put()** не проверяет наличия пары с таким же ключом, как и у добавляемой пары и просто заменяет значение по ключу на новое. Если критично наличие всех ранее добавленных элементов, следует перед добавлением пары выполнять проверку на наличие идентичных ключей. В простейшем варианте это выглядит:

```
HashMap<String, Integer> hm = new HashMap<String, Integer>() {
                        {
                                this.put("Сырок", 3);
                                this.put("Пряник", 5);
                                this.put("Молоко", 1);
                                this.put("Хлеб", 1);
                        }
        };
        if( !hm.containsKey("Пряник") ) { // замена не произойдет, если ключ существует
                        hm.put("Пряник", 4);
        }
```

Класс **EnumMap<K extends Enum<K>, V>** в качестве ключа может принимать только объекты, принадлежащие одному типу **enum**, который должен быть определен при создании коллекции. Специально организован для обеспечения максимальной скорости доступа к элементам коллекции.

/ # 18 # пример работы с классом EnumMap # GASStation.java # Gases.java */*

```
package by.bsu.enummap;
enum GASStation {
          DT(50), A80(30), A92(30), A95(50), GAS(40);
          private Integer maxVolume;
          private GASStation (int maxVolume) {
                    this.maxVolume = maxVolume;
          }
          public Integer getMaxVolume() {
                    return maxVolume;
          }
}
import java.util.EnumMap;
public class Gases {
   public static void main(String[ ] args) {
      EnumMap<GASStation, Integer> station1 =
          new EnumMap<GASStation, Integer>(GASStation.class);
          station1.put(GASStation.DT, 10);
          station1.put(GASStation.A80, 5);
          station1.put(GASStation.A92, 30);
      EnumMap<GASStation, Integer> station2 =
                    new EnumMap<GASStation, Integer>(GASStation.class);
          station2.put(GASStation.DT, 25);
          station2.put(GASStation.A95, 25);
          System.out.println(station1);
          System.out.println(station2);
    }
}
```

В результате будет выведено:

{DT=10, A80=5, A92=30}
{DT=25, A95=25}

Унаследованные коллекции

В ряде распределенных приложений, например с использованием сервлетов, до сих пор применяются коллекции, более медленные в обработке, но при этом потокобезопасные (thread-safety), существовавшие в языке Java с момента его создания, а именно карта **Hashtable<K, V>**, список **Vector<E>** и перечисление (аналог итератора) **Enumeration<E>**. Все они также были параметризованы, но сохраняют возможность одновременного доступа из конкурирующих потоков.

Класс **Hashtable<K, V>** реализует интерфейс **Map**,

```
java.util.Dictionary<K,V>
      └ java.util.Hashtable<K,V>
```

обладает также несколькими специфичными по сравнению с другими коллекциями методами:

Enumeration<V> elements() — возвращает перечисление для значений карты;
Enumeration<K> keys() — возвращает перечисление для ключей карты.

/ # 19 # создание хэш-таблицы и поиск элемента по ключу # HashTableDemo.java */*

```
package by.bsu.legacy;
import java.util.*;
public class HashTableDemo {
          public static void main(String[ ] args) {
                    Hashtable<Integer, Double> ht = new Hashtable<Integer, Double>() {
                        {
                              for (int i = 0; i < 5; i++) {
                                        ht.put(i, Math.atan(i));
                              }
                        }
                    };
                    Enumeration<Integer> ek = ht.keys();
                    int key;
                    while (ek.hasMoreElements()) {
                              key = ek.nextElement();
                              System.out.printf("%4d  ", key);
                    }
                    System.out.println("");
                    Enumeration<Double> ev = ht.elements();
                    double value;
                    while (ev.hasMoreElements()) {
                              value = ev.nextElement();
                              System.out.printf("%.2f  ", value);
                    }
          }
}
```

В результате в консоль будет выведено:

```
   4     3     2     1     0
1,33  1,25  1,11  0,79  0,00
```

Принципы работы с коллекциями, в отличие от их структуры, со сменой версий языка существенно не изменились.

Класс **Properties** предназначен для хранения карты свойств, где и ключ, и значения являются экземплярами класса **String**. Значения пары можно загружать и хранить в файле.

/ # 20 # создание экземпляра и файла properties # PropertiesDemoWriter.java */*

```
package by.bsu.collection;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Properties;
public class PropertiesDemoWriter {
   public static void main(String[ ] args) {
      Properties props = new Properties();
      try {
          // установка значений экземпляру
          props.setProperty("db.driver", "com.mysql.jdbc.Driver");
          // props.setProperty("db.url", "jdbc:mysql://127.0.0.1:3306/testphones");
          props.setProperty("db.user", "root");
          props.setProperty("db.password", "pass");
          props.setProperty("db.poolsize", "5");
          // запись properties-файла в папку prop проекта
          props.store(new FileWriter("prop" + File.separator + "database.properties"),null);
      } catch (IOException e) {
          e.printStackTrace();
      }
   }
}
```

В результате в файле **database.properties** будет размещена следующая информация:

#Wed Aug 01 03:40:01 FET 2012
db.password=pass
db.user=root
db.poolsize=5
db.driver=com.mysql.jdbc.Driver

Символ «=» служит по умолчанию разделителем ключа и значения в файле.**properties**, также в этом качестве можно использовать символ «:». Эти два специальных символа при записи в файл в качестве части ключа или значения получают впереди символ «\», чтобы в дальнейшем при чтении не быть воспринятым как разделитель «ключа–значения». Например, при задании свойства вида

```
props.setProperty("db.url", "jdbc:mysql://127.0.0.1:3306/testphones");
```

в файл будет записано

db.url=jdbc\:mysql\://127.0.0.1\:3306/testphones

Извлечь информацию из файла достаточно просто.

/ # 21 # загрузка файла properties в экземпляр и доступ к содержимому # PropertiesDemo.java */*

```
package by.bsu.collection;
import java.io.FileReader;
import java.io.IOException;
import java.util.Properties;
public class PropertiesDemo {
          public static void main(String[ ] args) {
                    Properties props = new Properties();
                    try {
                              // загрузка пар ключ-значение через поток ввода-вывода
                              props.load(new FileReader("prop\\database.properties"));
                    } catch (IOException e) {
                              e.printStackTrace();
                    }
                    String driver = props.getProperty("db.driver");
                    // следующих двух ключей в файле нет
                    String maxIdle = props.getProperty("db.maxIdle"); // будет присвоен null
                    // значение "20" будет присвоено ключу, если он не будет найден в файле
                    String maxActive = props.getProperty("db.maxActive", "20");
                    System.out.println(driver);
                    System.out.println(maxIdle);
                    System.out.println(maxActive);
          }
}
```

В результате будет выведено:

com.mysql.jdbc.Driver
null
20

Алгоритмы класса Collections

Класс **java.util.Collections** содержит большое количество статических методов, предназначенных для манипулирования коллекциями.

С применением предыдущих версий языка было разработано множество коллекций, в которых никаких проверок нет, следовательно, при их использовании нельзя гарантировать, что в коллекцию не будет помещен «посторонний» объект. Для этого в класс **Collections** был добавлен новый метод:

```
static <E> Collection <E> checkedCollection(Collection<E> c, Class<E> type)
```

Этот метод создает коллекцию, проверяемую на этапе выполнения, т. е. в случае добавления «постороннего» объекта генерируется исключение **ClassCastException**:

/ # 22 # проверяемая коллекция # SafeSetRun.java */*

```
package by.bsu.check;
import java.util.*;
public class SafeSetRun {
          public static void main(String args[ ]) {
                    HashSet orders;
                    // orders = new HashSet();  // заменяемый код на jdk1.4 и ниже
                    orders = Collections.checkedSet(new HashSet<Order>(), Order.class);
                    orders.add(new Order(параметры));
                    // some code here
                    orders.add(new Item(параметры)); // runtime error
          }
}
```

В этот же класс добавлен целый ряд статических методов, специализированных для проверки конкретных типов коллекций, а именно: **checkedList(), checkedSortedMap(), checkedMap(), checkedSortedSet(), checkedCollection(),** а также

<T> void copy(List<? super T> dest, List<? extends T> src) — копирует все элементы из одного списка в другой;

boolean disjoint(Collection<?> c1, Collection<?> c2) — возвращает **true**, если коллекции не содержат одинаковых элементов;

<T> List <T> emptyList(), <K, V> Map <K, V> emptyMap(), <T> Set <T> emptySet() — возвращают пустой список, карту отображения и множество соответственно;

<T> void fill(List<? super T> list, T obj) — заполняет список заданным элементом;

int frequency(Collection<?> c, Object o) — возвращает количество вхождений в коллекцию заданного элемента;

<T> boolean replaceAll(List<T> list, T oldVal, T newVal) — заменяет все заданные элементы новыми;

void reverse(List<?> list) — «переворачивает» список;

void rotate(List<?> list, int distance) — сдвигает список циклически на заданное число элементов;

void shuffle(List<?> list) — перетасовывает элементы списка;

singleton(T o), singletonList(T o), singletonMap(K key, V value) — создают множество, список и карту отображения, позволяющие добавлять только один элемент;

<T extends Comparable<? super T>> void sort(List<T> list),

<T> void sort(List<T> list, Comparator<? super T> c) — сортировка списка естественным порядком и с использованием **Comparable** или **Comparator** соответственно;

void swap(List<?> list, int i, int j) — меняет местами элементы списка, стоящие на заданных позициях;

<T> List<T> unmodifiableList(List<? extends T> list) — возвращает ссылку на список с запрещением его модификации. Аналогичные методы есть и для других видов коллекций.

/ # 23 # применение некоторых алгоритмов # AlgorithmDemo.java */*

```
package by.bsu.collect;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class AlgoritmDemo {
          public static void main(String[ ] args) {
                    Comparator<Integer> comp = new Comparator() {
                              public int compare(Integer n, Integer m) {
                                        return m.intValue() - n.intValue();
                              }
                    };
                    ArrayList<Integer> list = new ArrayList();

                    Collections.addAll(list, 1, 2, 3, 4, 5);
                    Collections.shuffle(list);
                    print(list);
                    Collections.sort(list, comp);
                    print(list);
                    Collections.reverse(list);
                    print(list);
                    Collections.rotate(list, 3);
                    print(list);
                    System.out.println("min: " + Collections.min(list, comp));
                    System.out.println("max: " + Collections.max(list, comp));

                    List<Integer> singl = Collections.singletonList(71);
                    print(singl);
                    // singl.add(21); // ошибка времени выполнения
          }
          private static void print(List<Integer> c) {
                    for (int i : c) {
                              System.out.print(i + " ");
                    }
                    System.out.println();
          }
}
```

В результате будет выведено:

```
4 3 5 1 2
5 4 3 2 1
1 2 3 4 5
3 4 5 1 2
min: 5
max: 1
71
```

Задания к главе 10

Вариант А

1. Ввести строки из файла, записать в список. Вывести строки в файл в обратном порядке.
2. Ввести число, занести его цифры в стек. Вывести число, у которого цифры идут в обратном порядке.
3. Создать в стеке индексный массив для быстрого доступа к записям в бинарном файле.
4. Создать список из элементов каталога и его подкаталогов.
5. Создать стек из номеров записи. Организовать прямой доступ к элементам записи.
6. Занести стихотворения одного автора в список. Провести сортировку по возрастанию длин строк.
7. Задать два стека, поменять информацию местами.
8. Определить множество на основе множества целых чисел. Создать методы для определения пересечения и объединения множеств.
9. Списки (стеки, очереди) I(1..n) и U(1..n) содержат результаты n-измерений тока и напряжения на неизвестном сопротивлении R. Найти приближенное число R методом наименьших квадратов.
10. С использованием множества выполнить попарное суммирование произвольного конечного ряда чисел по следующим правилам: 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.
11. Сложить два многочлена заданной степени, если коэффициенты многочленов хранятся в объекте **HashMap**.
12. Умножить два многочлена заданной степени, если коэффициенты многочленов хранятся в различных списках.
13. Не используя вспомогательных объектов, переставить отрицательные элементы данного списка в конец, а положительные — в начало списка.
14. Ввести строки из файла, записать в список **ArrayList**. Выполнить сортировку строк, используя метод **sort()** из класса **Collections**.
15. Задана строка, состоящая из символов «(», «)», «[», «]», «{», «}». Проверить правильность расстановки скобок. Использовать стек.
16. Задан файл с текстом на английском языке. Выделить все различные слова. Слова, отличающиеся только регистром букв, считать одинаковыми. Использовать класс **HashSet**.
17. Задан файл с текстом на английском языке. Выделить все различные слова. Для каждого слова подсчитать частоту его встречаемости. Слова, отличающиеся регистром букв, считать различными. Использовать класс **HashMap**.

Вариант В

1. В кругу стоят *N* человек, пронумерованных от 1 до *N*. При ведении счета по кругу вычеркивается каждый второй человек, пока не останется один. Составить две программы, моделирующие процесс. Одна из программ должна использовать класс **ArrayList**, а вторая — **LinkedList**. Какая из двух программ работает быстрее? Почему?
2. Задан список целых чисел и число *X*. Не используя вспомогательных объектов и не изменяя размера списка, переставить элементы списка так, чтобы сначала шли числа, не превосходящие *X*, а затем числа, больше *X*.
3. Написать программу, осуществляющую сжатие английского текста. Построить для каждого слова в тексте оптимальный префиксный код по алгоритму Хаффмена. Использовать класс **PriorityQueue**.
4. Реализовать класс Graph, представляющий собой неориентированный граф. В конструкторе класса передается количество вершин в графе. Методы должны поддерживать быстрое добавление и удаление ребер.
5. На базе коллекций реализовать структуру хранения чисел с поддержкой следующих операций:
   - добавление/удаление числа;
   - поиск числа, наиболее близкого к заданному (т. е. модуль разницы минимален).
6. Реализовать класс, моделирующий работу *N*-местной автостоянки. Машина подъезжает к определенному месту и едет вправо, пока не встретится свободное место. Класс должен поддерживать методы, обслуживающие приезд и отъезд машины.
7. Во входном файле хранятся две разреженные матрицы — A и B. Построить циклически связанные списки CA и CB, содержащие ненулевые элементы соответственно матриц A и B. Просматривая списки, вычислить: а) сумму S = A + B; б) произведение P = A × B.
8. Во входном файле хранятся наименования некоторых объектов. Построить список C1, элементы которого содержат наименования и шифры данных объектов, причем элементы списка должны быть упорядочены по возрастанию шифров. Затем «сжать» список C1, удаляя дублирующие наименования объектов.
9. Во входном файле расположены два набора положительных чисел; между наборами стоит отрицательное число. Построить два списка C1 и C2, элементы которых содержат соответственно числа 1-го и 2-го набора таким образом, чтобы внутри одного списка числа были упорядочены по возрастанию. Затем объединить списки C1 и C2 в один упорядоченный список, изменяя только значения полей ссылочного типа.
10. Во входном файле хранится информация о системе главных автодорог, связывающих г. Минск с другими городами Беларуси. Используя эту информацию,

построить дерево, отображающее систему дорог республики, а затем, продвигаясь по дереву, определить минимальный по длине путь из г. Минска в другой заданный город. Предусмотреть возможность сохранения дерева в виртуальной памяти.

11. Один из способов шифрования данных, называемый «двойным шифрованием», заключается в том, что исходные данные при помощи некоторого преобразования последовательно шифруются на некоторые два ключа — K1 и K2. Разработать и реализовать эффективный алгоритм, позволяющий находить ключи K1 и K2 по исходной строке и ее зашифрованному варианту. Проверить, оказался ли разработанный способ действительно эффективным, протестировав программу для случая, когда оба ключа являются 20-битными (время ее работы не должно превосходить одной минуты).
12. На плоскости задано *N* точек. Вывести в файл описания всех прямых, которые проходят более чем через одну точку из заданных. Для каждой прямой указать, через сколько точек она проходит. Использовать класс **HashMap**.
13. На клетчатой бумаге нарисован круг. Вывести в файл описания всех клеток, целиком лежащих внутри круга, в порядке возрастания расстояния от клетки до центра круга. Использовать класс **PriorityQueue**.
14. На плоскости задано *N* отрезков. Найти точку пересечения двух отрезков, имеющую минимальную абсциссу. Использовать класс **TreeMap**.
15. На клетчатом листе бумаги закрашена часть клеток. Выделить все различные фигуры, которые образовались при этом. Фигурой считается набор закрашенных клеток, достижимых друг из друга при движении в четырех направлениях. Две фигуры являются различными, если их нельзя совместить поворотом на угол, кратный 90 градусам, и параллельным переносом. Используйте класс **HashSet**.
16. Дана матрица из целых чисел. Найти в ней прямоугольную подматрицу, состоящую из максимального количества одинаковых элементов. Использовать класс **Stack**.
17. Реализовать структуру «черный ящик», хранящую множество чисел и имеющую внутренний счетчик K, изначально равный нулю. Структура должна поддерживать операции добавления числа в множество и возвращение *K*-го по минимальности числа из множества.
18. На прямой гоночной трассе стоит *N* автомобилей, для каждого из которых известны начальное положение и скорость. Определить, сколько произойдет обгонов.
19. На прямой гоночной трассе стоит *N* автомобилей, для каждого из которых известны начальное положение и скорость. Вывести первые *K* обгонов.