# 12 СТАНДАРТНЫЕ ВСПОМОГАТЕЛЬНЫЕ СРЕДСТВА

Пакет java.util содержит ряд стандартных вспомогательных интерфейсов и классов.

# Содержание

# Пакет **java.util** содержит:

***Коллекции:***

- **BitSet**: битовый вектор с динамическим изменением размера.
- **Enumeration**: интерфейс, который возвращает объект, используемый для перечисления набора объектов (например, элементов, содержащихся в конкретной хеш-таблице).
- **Vector**: вектор, состоящий из элементов типа **Object**, с динамическим изменением размера.
- **Stack**: расширение класса **Vector**, в котором добавлены методы для работы с простейшим стеком **LIFO** (“последним пришел, первым вышел”).
- **Dictionary**: абстрактный класс, содержащий алгоритмы для работы с парами ключ/значение.
- **Hashtable**: реализация **Dictionary**, в которой для сопоставления ключа со значением используется хеш-код.
- **Properties**: расширение **Hashtable**, в котором строковые ключи сопоставляются со строковыми значениями.

***Концепции проектирования:***

- **Observer/Observable**: с помощью этой пары интерфейс/класс вы можете сделать свой объект “наблюдаемым” (Observable) — закрепить за ним один или более объектов-наблюдателей (Observer), которые будут извещаться в том случае, если с наблюдаемым объектом происходит что-то интересное.

***Прочее:***

- **Date**: работа с датами с точностью до одной секунды.
- **Random**: объекты, генерирующие последовательности псевдослучайных чисел.
- **StringTokenizer**: деление строки на лексемы с учетом символов-ограничителей. По умолчанию ими считаются разделители **(whitespace**).

# 12.1. Класс BitSet

Класс **BitSet** позволяет создать битовый вектор, размер которого изменяется динамически.

**BitSet** представляет собой набор битов со значениями true или false размером до 232–1, причем изначально все биты равны false. Для хранения набора выделяется объем памяти, необходимый для хранения вектора вплоть до старшего бита, который устанавливался или сбрасывался в программе — все превышающие его биты считаются равными false.

При создании объекта **BitSet** можно явно задать исходный размер набора или воспользоваться безаргументным конструктором для установки размера по умолчанию.

public void **set(int bit)**

Устанавливает бит в позиции bit, присваивая ему значение true.

public void **clear(int bit)**

Сбрасывает бит в позиции bit, присваивая ему значение false.

public boolean **get(int bit)**

Возвращает значение бита в позиции bit.

public void **and(BitSet other)**

Выполняет операцию логического И над данным набором и other и присваивает результат данному набору.

public void **or(BitSet other)**

Выполняет операцию логического ИЛИ над данным набором и other и присваивает результат данному набору.

public void **xor(BitSet other)**

Выполняет операцию исключающего логического ИЛИ над данным набором и other и присваивает результат данному набору.

# Класс BitSet

public int **size()**

Возвращает позицию старшего бита в наборе, который может быть установлен или сброшен без необходимости увеличения набора.

public int **hashCode()**

Возвращает хеш-код для набора, определяемый значениями битов.

public boolean **equals(BitSet other)**

Возвращает true, если все биты other совпадают с битами в данном наборе.

В приведенном ниже **примере**

С помощью объекта BitSet происходит пометка символов, встречающихся в строке. Объект можно распечатать и посмотреть, какие же символы входят в строку:

# Реализация

```
import java.util.*;
public class WhichChars {
    private BitSet used = new BitSet();
    public WhichChars(String str) {
     for (int i = 0; i < str.length(); i++)
         used.set(str.charAt(i));
   // установить бит, соответствующий символу
     }
     public String toString() {
         String desc = "[";
         int size = used.size();
         System.out.println(size);
          for (int i = 0; i < size; i++) {
               if (used.get(i))  {desc += ~(char)i;
        System.out.println((char)i);}  }
               return desc + "]";     }}
class WhichCars_Test{
public static void main(String[] args){
    WhichChars  wc= new WhichChars(args[0]);
System.out.print(wc.toString());
    }
}
```

# 12.2. Интерфейс Enumeration

Большинство классов-коллекций использует интерфейс Enumeration в качестве средства для перебора объектов, входящих в коллекцию. Интерфейс Enumeration объявляет два метода:

public abstract boolean **hasMoreElements()**

Возвращает true, если перебор элементов перечисления еще не закончен. Метод может многократно вызываться между последовательными вызовами **nextElement**.

public abstract Object **nextElement()**

Возвращает следующий элемент перечисления. Вызовы этого метода осуществляют последовательный перебор всех элементов. Если следующего элемента не существует, возбуждается исключение **NoSuchElementException.**

Цикл, в котором Enumeration используется для перебора элементов класса-коллекции, в данном случае — элементов хеш-таблицы:

```
Enumeration e = table.elements();
while (e.hasMoreElements())
   doSomethingWith(e.nextElement());
```

**Контракт Enumeration** не гарантирует *фиксации исходного состояния (snapshot guarantee)*. Другими словами, если содержимое коллекции изменяется во время итерации, это может отразиться на значениях, возвращаемых методами. Например, если в реализации **nextElement** используется содержимое исходной коллекции, то удаление объектов из списка во время перебора может иметь разрушительные последствия.

# 12.3. Реализация интерфейса Enumeration

При разработке новых классов-коллекций может возникнуть необходимость в собственной реализации **Enumeration**. Следующий класс реализует интерфейс **Enumeration** для того, чтобы возвращать символы, представленные объектом BitSet в классе WhichChars:

```
class EnumerateWhichChars implements Enumeration {
    private BitSet bits;
   private int pos;      // следующая проверяемая позиция
   private int setSize;  // количество бит (для оптимизации)
    EnumerateWhichChars(BitSet whichBits) {
        bits = whichBits;
        setSize = whichBits.size();
        pos = 0;
    }     public boolean hasMoreElements() {
        while (pos < setSize && !bits.get(pos))
           pos++;
       return (pos < setSize);     }
   public Object nextElement() {
        if (hasMoreElements())
            return new Character((char)pos++);
       else         return null;     }}
```

# Реализация интерфейса Enumeration

Класс перебирает биты, входящие в **BitSet**, и возвращает объекты **Character** со значениями символов, которым соответствуют установленные биты в объекте **BitSet**. Метод **hasMoreElements** перемещает текущую позицию к следующему возвращаемому элементу. Он написан так, чтобы его можно было многократно использовать для каждого вызова **nextElement**.

Теперь в класс **WhichChars** необходимо включить метод, который возвращает объект-перечисление:

```
public Enumeration characters() {
    return new EnumeratrWhichChars(used);}
```

Обратите внимание: метод **characters** возвращает объект класса **Enumeration**, а не EnumerateWhichChars. Класс EnumerateWhichChars не предназначен для открытого использования, поэтому реализацию перечисления можно скрыть. Если только вы не захотите возвращать объект-перечисление с новыми открытыми возможностями, следует скрывать тип объекта, чтобы оставить для себя возможность изменить его реализацию по своему усмотрению.

# 12.4. Класс Vector

Класс Vector предназначен для работы с массивом переменного размера, состоящим из элементов **Object**. Новые элементы могут добавляться в начало, середину или конец вектора, и к любому элементу можно обратиться посредством индекса. Массивы в языке Java имеют фиксированный размер, так что объекты **Vector** оказываются полезными в тех случаях, когда в момент создания массива неизвестно максимальное количество сохраняемых элементов или это количество велико, а достигается оно редко.

Методы класса **Vector** делятся на три категории:

- Методы для модификации вектора.
- Методы для получения объектов, хранящихся в векторе.
- Методы, управляющие процессом расширения вектора, когда его емкости оказывается недостаточно.

**Безаргументный конструктор** создает объект **Vector**, размер которого регулируется в соответствии с принятыми по умолчанию правилами.

Многие методы класса изменяют содержимое вектора. Все они, кроме **setElements,** при необходимости осуществляют динамическое изменение размера вектора в соответствии со своими потребностями.

# Класс Vector

public final synchronized void **setElementAt(Object obj, int index)**

Присваивает obj элементу вектора с индексом index. Старое значение этого элемента пропадает. При задании индекса, превышающего текущий размер вектора, возбуждается исключение **IndexOutOfBoundsException**. Чтобы убедиться в корректности индекса перед его применением, используйте метод **setSize** (см. ниже).

public final synchronized void **removeElementAt(Object obj, int index)**

Удаляет элемент вектора с индексом **index**. Элементы, находящиеся после удаленного, сдвигаются к началу, а размер вектора уменьшается на 1.

public final synchronized void **insertElementAt(Object obj, int index)**

Вставляет элемент obj в позицию index. Элементы, следующие после удаленного, сдвигаются, чтобы освободить место для вставки.

public final synchronized void **addElement(Object obj)**

Добавляет элемент obj к концу вектора.

# Класс Vector

public final synchronized boolean **removeElement(Object obj)**

Эквивалентен методу indexOf(obj) и — в случае удачного поиска — вызову **removeElementAt** для найденного индекса. Если объект не является элементом вектора, removeElement возвращает false (метод indexOf описывается ниже).

public final synchronized void **removeAllElements()**

Удаляет все элементы вектора. Вектор становится пустым.

Класс **Polygon** предназначен для хранения списка объектов **Point**, которые представляют собой вершины многоугольника:

```
import java.util.Vector;
public class Polygon {
   private Vector vertices = new Vector();
     public void add(Point p) {
          vertices.addElements(p);     }
     public void remove(Point p) {
         vertices. RemoveElement(p);     }
   public int numVertices() {
       return vertices.size();     }     // ... другие методы ...}
```

# Класс Vector

Существует ряд методов, предназначенных для просмотра содержимого вектора. При задании недопустимого индекса возбуждается исключение Index OutOfBoundsException. Все методы, которые ищут элемент в векторе, используют метод Object.equals для сравнения искомого объекта с элементами Vector.

public final synchronized Object **elementAt(int index)**

Возвращает элемент с индексом index.

public final boolean **contains(Object obj)**

Возвращает true, если obj является элементом вектора.

public final synchronized int **indexOf(Object obj, int index)**

Ищет первое вхождение заданного объекта obj начиная с позиции index, и возвращает его индекс или –1, если объект не найден.

public final int **indexOf(Object obj)**

Эквивалентен indexOf(obj,0).

public final synchronized int **lastIndexOf(Object obj,**
**int index)**

Осуществляет поиск obj в обратном направлении от позиции index и возвращает его индекс или –1, если объект не найден.

public final int **lastIndexOf(Object obj)**

Эквивалентен lastIndexOf(obj,size()-1).

public final synchronized void **copyInto(Object[] anArray)**

Копирует элементы вектора в заданный массив.

# Класс Vector

public final synchronized Enumeration **elements()**

Возвращает **Enumeration** для текущего состава элементов. Для последовательной выборки элементов возвращаемого объекта применяются методы **Enumeration**. Исходное состояние при этом не фиксируется, поэтому для получения “фотографии” содержимого вектора пользуйтесь методом **copyInto**.

public final synchronized Object **firstElement()**

Возвращает первый элемент вектора. Если вектор пуст, возбуждается исключение NoSuchElementException.

public final synchronized Object **lastElement()**

Возвращает последний элемент вектора. Если вектор пуст, возбуждается исключение NoSuchElementException. Пара методов firstElement/ lastElement может использоваться для перебора элементов вектора, но существует риск изменения вектора во время выполнения цикла. Для получения “фотографии” содержимого вектора пользуйтесь методом **copyInto.**

Размер вектора равен количеству элементов, содержащихся в нем. Чтобы изменить размер вектора, можно добавлять или удалять элементы либо вызвать метод setSize или trimSize:

# Класс Vector

public final int **size()**

Возвращает количество элементов, содержащихся в векторе. Обратите внимание, что эта величина отличается от емкости вектора.

public final boolean **isEmpty()**

Возвращает true, если вектор не содержит ни одного элемента.

public final synchronized void **trimToSize()**

Сокращает емкость вектора до текущего размера. Этот метод используется для минимизации объема памяти, когда вектор находится в устойчивом состоянии. Последующие добавления элементов к вектору приведут к его увеличению.

public final synchronized void **setSize(int newSize)**

Устанавливает размер вектора равным **newSize**. Если при этом вектор сокращается, то элементы за его концом теряются; если вектор увеличивается, то новые элементы равны null.

Правильное управление емкостью вектора существенно влияет на эффективность работы

Если вы знаете, сколько элементов будет добавлено в вектор, используйте метод **ensureCapacity** для однократного увеличения емкости вектора.

# Класс Vector

Параметры управления емкостью задаются при конструировании вектора. Для создания объектов Vector применяются следующие конструкторы:

public **Vector(int initialCapacity, int capacityIncrement)**

Создает пустой вектор с заданной исходной емкостью и запоминает ее приращение. Приращение, равное 0, означает удвоение емкости буфера при каждом его увеличении; в противном случае буфер увеличивается на capacityIncrement элементов.

public **Vector(int initialCapacity)**

Эквивалентен Vector(initialCapacity, 0).

public **Vector()**

Конструирует пустой вектор со значениями исходной емкости и приращения, заданными по умолчанию.

public final int **capacity()**

Возвращает текущую емкость вектора — количество элементов, которые могут храниться в векторе без увеличения его размера.

public final synchronized void **ensureCapacity** (int minCapacity)

Метод гарантирует, что емкость вектора будет не ниже заданной, и при необходимости увеличивает текущую емкость.

# Класс Vector

Приведем метод класса Polygon, который присоединяет к объекту вершины другого многоугольника:

```
public void merge(Polygon other) {
    int otherSize = other.vertices.size();
     vertices.enusreCapacity(vertices.size() + otherSize);
    for (int i = 0; i <otherSize; i++)
        vertices.addElement(other.vertices.elementAt(i));}
```

В этом примере метод ensureCapacity используется для того, чтобы с добавлением новых вершин емкость вектора увеличивалась не более одного раза.

Реализация vector.toString выдает строку с полным описанием вектора, включающую результат вызова **toString** для каждого из содержащихся в нем элементов

# Класс Vector

Помимо этих открытых методов, подклассы Vector могут использовать защищенные поля класса. Соблюдайте осторожность в работе с ними — например, методы Vector предполагают, что размер буфера превышает количество элементов вектора.

public Object **elementData[]**

Буфер, в котором хранятся элементы вектора.

public int **elementCount**

Текущее количество элементов в буфере.

public int **capacityIncrement**

Количество элементов, которое добавляется к емкости вектора при заполнении буфера elementData. Если значение этого поля равно 0, то размер буфера удваивается при каждой необходимости его увеличения.

**Упражнение 12.1**

Напишите программу, которая открывает файл и читает из него строки (по одной), сохраняя каждую строку в объекте Vector, сортируемом методом String.compareTo. В этом вам может пригодиться класс для чтения строк из файла, созданный в упражнении 11.2.

# 12.5. Класс Stack

Класс Stack расширяет **Vector**, добавляя методы для реализации простейшего стека объектов **Object**, построенного по принципу **LIFO** (“последним пришел, первым вышел”). Метод **push** заносит объект в стек, а **pop** выталкивает верхний элемент стека. Метод **peek** возвращает значение верхнего элемента стека, при этом сам элемент остается в стеке. Метод **empty** возвращает true, если стек пуст. Попытка вызова pop или peek для пустого объекта-стека приводит к возбуждению исключения **EmptyStackException**.

Чтобы выяснить, насколько далеко расположен тот или иной элемент от вершины стека, применяется метод **search**; 1 соответствует вершине стека. Если объект не найден, возвращается –1. Для проверки совпадения искомого объекта с объектами в стеке применяется метод **Object.equals**.

В приведенном ниже примере класс **Stack** используется для слежения за тем, у кого в данный момент находится некоторый предмет. Имя исходного владельца попадает в стек первым. Когда кто-нибудь одалживает у него предмет, имя должника также заносится в стек. При возврате предмета имя должника выталкивается из стека. Последнее имя должно всегда оставаться в стеке, так как в противном случае будет утрачена информация о владельце.

# Class Borrow

```
import java.util.Stack;public class Borrow {
     private String itemName;
    private Stack hasIt = new Stack();
    public Borrow(String name, String owner) {
          itemName = name;
         hasIt.push(owner);
     // первым следует имя владельца     }
     public void borrow(String borrower) {
         hasIt.push(borrower);     }
    public String currentHolder() {
        return (String)hasIt.peek();     }
    public String returnIt() {
       String ret = (String)hasIt.pop();
        if (hasIt.empty())
    // случайно вытолкнутый владелец
             hasIt.push(ret);   // вернуть его обратно
        return ret;     }}
```

**Упражнение 12.2**

Добавьте метод, который использует метод search, чтобы определить количество должников.

# 12.6. Класс Dictionary

Абстрактный класс **Dictionary** представляет собой интерфейс.

В нем определен ряд абстрактных методов, предназначенных для хранения ***элемента*** с некоторым ***ключом*** и последующей выборки элемента по ключу. Этот интерфейс является базовым для класса **Hashtable**, однако класс **Dictionary** определен отдельно, чтобы другие реализации могли использовать разные алгоритмы для сопоставления ключа с элементом.

Класс **Dictionary** возвращает **null**, сигнализируя о таких событиях, как отсутствие элемента с заданным ключом; следовательно, ни ключ, ни элемент не могут быть равны null. Если задать значение null для аргумента-ключа или элемента, возбуждается исключение **NullPointerException**.

Класс **Dictionary** содержит следующие методы:

public abstract Object **put(Object key, Object element)**

Заносит element в словарь с ключом key. Возвращает старый элемент, хранившийся с ключом key, или null, если такого элемента нет.

public abstract Object **get(Object key)**

Возвращает объект, занесенный в словарь с ключом key, или null, если ключ не определен.

public abstract Object **remove(Object key)**

Удаляет из словаря элемент с ключом key и возвращает значение удаленного элемента или null, если ключ не определен.

# Класс Dictionary

public abstract int **size()**

Возвращает количество элементов в словаре.

public abstract boolean **isEmpty()**

Возвращает true, если словарь не содержит ни одного элемента.

public abstract Enumeration **keys()**

Возвращает объект-перечисление для всех ключей, входящих в словарь.

public abstract Enumeration **elements()**

Возвращает объект-перечисление для всех элементов, входящих в словарь.

Объекты-перечисления, возвращаемые методами keys и elements, не гарантируют фиксации исходного состояния, однако при создании класса, в котором используется Dictionary, можно осуществить такую гарантию в вашей собственной реализации этих методов.

# 12.7. Класс Hashtable

***Хеш-таблицы*** представляют собой распространенный механизм для хранения пар ключ/элемент.

Класс **Hashtable** реализует интерфейс **Dictionary**. Он обладает определенной емкостью и средствами, определяющими момент увеличения таблицы. Расширение хеш-таблицы требует повторного хеширования всех ее элементов в соответствии с их новым положением в увеличенной таблице, так что важно обеспечить однократное изменение таблицы.

Другой фактор, влияющий на эффективность хеш-таблицы, — процесс генерации **хеш-кода** по ключу.

Конфликты хеш-кодов должны происходить как можно реже. Хеш-коды обязаны равномерно распределяться по диапазону возможных значений, который для класса **Hashtable** совпадает с полным диапазоном **типа int**..

Значение **хеш-кода** возвращается методом **hashCode** для объекта, являющегося ключом. По умолчанию каждый объект имеет уникальный хеш-код. Использование в качестве ключей случайно выбранных объектов приводит к порождению различных хеш-кодов.

Классы **String, BitSet** и большинство других, переопределяющих метод **equal**, обычно переопределяют и **hashCode**.

Это важно, поскольку **класс Hashtable** использует **хеш-код** для нахождения набора ключей, которые могут совпадать с заданным, и вызывает **equal** для каждого из таких объектов, пока не будет найден совпадающий.

Если для некоторых объектов **equal** и **hashCode** окажутся несовместимыми, то при использовании объектов этого типа в качестве ключей **Hastable** их поведение окажется непредсказуемым.

# Класс Hashtable

Кроме методов, входящих в класс **Dictionary (get, put, remove, size, isEmpty, keys и elements), Hastable** содержит следующие методы:

public synchronized boolean **containsKey(Object key)**

Возвращает true, если хеш-таблица содержит элемент с заданным ключом.

public synchronized boolean **contains(Object element)**

Возвращает true, если заданный **element** является элементом **хеш-таблицы**. Данная операция является более сложной, чем метод **containsKey**, поскольку хеш-таблица спроектирована с расчетом на эффективный поиск ключей, а не элементов.

public synchronized void **clear()**

Делает хеш-таблицу пустой.

public synchronized Object **clone()**

Создает дубликат хеш-таблицы. Ключи и элементы при этом *не* дублируются.

Объекты **Hashtable** автоматически увеличиваются, когда они становятся слишком заполненными. Под выражением “слишком заполненными” понимается превышение *показателя загрузки* таблицы, который представляет собой отношение количества элементов к текущей емкости таблицы.

Когда таблица увеличивается, ее новая емкость примерно вдвое превышает текущую. Для повышения эффективности следует выбирать емкость, представленную простым числом, чтобы при увеличении объекта **Hastable** также было выбрано ближайшее простое число. Исходная емкость хеш-таблицы и показатель загрузки могут задаваться в конструкторах **Hashtable**:

# Класс Hashtable

public **Hashtable()**

Конструирует новую, пустую хеш-таблицу с принятой по умолчанию исходной емкостью и показателем загрузки, равным 0, 75.

public **Hashtable(int initialCapacity)**

Конструирует новую, пустую хеш-таблицу с заданной емкостью initial Capacity и принятым по умолчанию показателем загрузки, равным 0,75.

public **Hashtable(int initialCapacity, float loadFactor)**

Конструирует новую, пустую хеш-таблицу с заданной емкостью и показателем загрузки loadFactor, который представляет собой число, лежащее в диапазоне 0,0–1,0 и определяющее момент увеличения хеш-таблицы. Если количество элементов хеш-таблицы превышает текущую емкость, умноженную на показатель загрузки, то хеш-таблица автоматически увеличивается.

При увеличении объекта Hashtable повторное хеширование осуществляется методом **rehash**.

Метод **rehash** является защищенным, так что расширенные классы могут вызывать его по своему усмотрению, когда они решат, что наступило время увеличить емкость таблицы. Задать новый размер при этом невозможно — он всегда вычисляется методом rehash.

При реализации метод **Hashtable.toString** возвращает строку, которая полностью описывает содержимое таблицы, включая результаты вызова **toString** для всех ключей и элементов, входящих в нее.

**Упражнение 12.3**

Напишите программу, которая пользуется объектом **StreamTokenizer** для разбиения входного файла на слова и подсчета количества слов в файле, с выводом результата.

# 12.8. Класс Properties

Класс **Properties** является расширением **Hashtable**.

Практически для всех манипуляций со списками свойств используются методы **Hashtable**, однако для получения свойств применяется один из двух методов **getProperty**:

public String **getProperty(String key)**

Возвращает элемент для заданного ключа key. Если ключ отсутствует в списке свойств, просматривается список свойств по умолчанию (если он существует). Метод возвращает null, если свойство не найдено.

public String **getProperty(String key, String defaultElement)**

Возвращает элемент для заданного ключа key. Если ключ отсутствует в списке свойств, просматривается список свойств по умолчанию (если он существует). Если элемент отсутствует в обоих списках, возвращается строка defaultElement.

Класс **Properties** содержит два конструктора: один вызывается без аргументов, а второму передается объект Properties, который представляет вспомогательный список свойств по умолчанию. Если поиск в основном списке свойств оказывается неудачным, то просматривается вспомогательный объект Properties, который, в свою очередь, может иметь собственный вспомогательный объект со свойствами по умолчанию, и так далее. Цепочка основных и вспомогательных списков свойств может иметь произвольную длину.

# Класс Properties

public **Properties()**

Создает пустой список свойств.

public **Properties(Properties defaults)**

Создает пустой список свойств с заданным вспомогательным объектом Properties для поиска свойств, отсутствующих в основном списке.

Если список свойств состоит только из строковых ключей и элементов, можно записывать или считывать его из файла или иного потока ввода/вывода с помощью следующих методов:

public void **save(OutputStream out, String header)**

Сохраняет содержимое списка свойств в OutputStream. Строка header записывается в выходной поток в виде комментария, состоящего из одной строки.

public synchronized void **load(InputStream in)** throws

IOException

Загружает список свойств из InputStream. Предполагается, что список свойств был ранее сохранен методом save. Метод загружает свойства только в основной список, но не во вспомогательный.

Для получения объекта Enumeration, представляющего собой “фотографию” ключей в списке свойств, применяется метод propertyNames:

public Enumeration **propertyNames()**

Создает объект-перечисление с перечнем всех ключей. Метод гарантирует фиксацию исходного состояния.

public void **list(PrintStream out)**

Выводит свойства из списка в заданный поток PrintStream. Метод полезен во время отладки.

# 12.9. Классы Observer/Observable

Типы **Observer/Observable** предоставляют протокол, в соответствии с которым произвольное количество объектов-наблюдателей **Observer** получают уведомления о каких-либо изменениях или событиях, относящихся к произвольному количеству объектов **Observable**.

Объект **Observable** производится от подкласса **Observable**, благодаря чему можно вести список объектов **Observer**, уведомляемых об изменениях в объекте **Observable**. Все объекты- "наблюдатели", входящие в список, должны реализовывать интерфейс **Observer**. Когда с наблюдаемым объектом происходят изменения, заслуживающие внимания, или случаются некоторые события, которые представляют интерес для **Observer**, вызывается метод **notifyObservers** объекта **Observable**, который обращается к методу **update** для каждого из объектов **Observer**. Метод **update** интерфейса **Observable** выглядит следующим образом:

public abstract void **update(Observable obj, Object arg)**

Метод вызывается, когда объект **Observable** должен сообщить наблюдателям об изменении или некотором событии. Параметр arg дает возможность передачи произвольного объекта, содержащего описание изменения или события в объекте Observer.

Класс **Observable** реализует методы для ведения списка объектов **Observer**, для установки флага, сообщающего об изменении объекта, а также для вызова метода **update** любого из объектов **Observer**.

# Классы Observer/Observable

Для ведения списка объектов **Observer** используются следующие методы:

public synchronized void **addObserver(Observer o)**

Добавляет аргумент o типа Observer к списку объектов-наблюдателей.

public synchronized void **deleteObserver(Observer o)**

Удаляет аргумент o типа Observer из списка объектов-наблюдателей.

public synchronized void **deleteObservers()**

Удаляет все объекты Observer из списка наблюдателей.

public synchronized int **countObservers()**

Возвращает количество объектов-наблюдателей.

Следующие методы извещают объекты Observer о произошедших изменениях:

public synchronized void **notifyObservers(Object arg)**

Уведомляет все объекты Observer о том, что с наблюдаемым объектом что-то произошло, после чего сбрасывает флаг изменения объекта. Для каждого объекта-наблюдателя, входящего в список, вызывается его метод update, первым параметром которого является объект Observable, а вторым — arg.

public void **notifyObservers()**

Эквивалентен notifyObservers(null).

# Классы Observer/Observable

Приведенный пример показывает, как протокол **Observer/Observable** может применяться для наблюдения за пользователями, зарегистрированными в системе.

Сначала определяется класс **Users**, расширяющий Observable:

```
import java.util.*;
public class Users extends Observable {
    private Hashtable loggedIn = new Hashtable();
    public void login(String name, String password) throws BadUserException
    {
        // метод возбуждает исключение BadUserException
       if (!passwordValid(name, password) throw new
    BadUserException(name);
        UserState state = new UserState(name);
       loggedIn.put(name, state);
        setChanged();
       notifyObservers(state);     }
   public void logout(UserState state) {
        loggedIn.remove(state.name());
        setChanged();
       notifyObservers(state);     }     // ...}
```

# Классы Observer/Observable

Объект **Users** содержит список активных пользователей и для каждого из них заводит объект **UserState**. Когда кто-либо из пользователей входит в систему или прекращает работу, то всем объектам **Observer** передается его объект **UserState**.

Метод **notifyObservers** рассылает сообщения наблюдателям лишь в случае изменения состояния наблюдаемого объекта, так что мы должны также вызвать метод **setChanged** для **Users**, иначе **notifyObservers** ничего не сделает.

Кроме метода **setChanged**, существует еще два метода для работы с флагом изменения состояния:

**clearChanged** помечает объект **Observable** как неизменявшийся, а

**hasChanged** возвращает логическое значение флага.

# Классы Observer/Observable

Реализация **update** для объекта **Observer**, постоянно следящего за составом зарегистрированных пользователей:

```
import java.util.*;
public class Eye implements Observer {
    Users watching;
     public Eye(Users users) {
          watching = users;
        watching.addObserver(this);     }
     public void update(Observable users, Object whichState)     {
         if (users != watching)  throw new IllegalArgumentException();
         UserState state = (UserState)whichState;
        if (watching.loggedIn(state))    // вход в систему
      addUser(state);
          // внести в список
        else     removeUser(state);
        // удалить из списка     }}
```

# Классы Observer/Observable

Каждый объект **Eye** наблюдает за конкретным объектом Users.

Когда пользователь входит в систему или прекращает работу, объект Eye извещается об этом, поскольку в его конструкторе вызывается метод **addObserver** для объекта **User**, в котором объект **Eye** указывается в качестве объекта-наблюдателя.

При вызове метода **update** происходит проверка на правильность параметров и изменение выводимой информации в зависимости от того, вошел ли данный пользователь в систему или вышел.

Механизм **Observer/Observable** отчасти напоминает механизм **wait/ notify** для потоков, однако он отличается большей гибкостью и меньшим количеством ограничений.

Механизм потоков гарантирует, что синхронный доступ защитит программу от нежелательных эффектов многозадачности.

Механизм наблюдения позволяет организовать между участниками любую связь, не зависящую от используемых потоков. В обоих механизмах предусмотрен поставщик информации (**Observable** и объект, вызывающий **notify**) и ее потребитель (**Observer** и объект, вызывающий **wait**), однако они удовлетворяют различные потребности.

**Упражнение 12.6**

Создайте реализацию интерфейса **Attributed**, в которой механизм **Observer/Observable** используется для уведомления наблюдателей об изменениях, происходящих с объектами

# 12.10. Класс Date

Класс **Date** предоставляет в распоряжение программиста механизм для вычислений, связанных с датами и временем, а также для вывода их результатов

Вы можете установить дату и определить ее, при необходимости учитывая локальный часовой пояс.

Предполагается, что класс **Date** работает в соответствии со стандартом UTC (Coordinated Universal Time — координированное универсальное время), однако это не всегда возможно.

Неточности возникают из-за механизмов обращения со временем, используемых в операционной системе. /Почти все современные системы временного исчисления предполагают, что одни сутки состоят из 24*60*60 секунд. В системе UTC примерно раз в год к суткам прибавляется дополнительная секунда, называемая "переходной".

Большинство компьютерных часов не обладает необходимой точностью, чтобы отражать этот факт, поэтому класс **Date** также не учитывает его. Некоторые компьютерные стандарты определены в GMT - это название является общеупотребительным, тогда как UT представляет собой "научное" название того же самого стандарта.

Различие между UTC и UT состоит в том, что стандарт UT основан на атомных часах, а UTC - на астрономических наблюдениях. На практике отличие оказывается пренебрежимо малым.

Компоненты дат задаются в единицах, принятых в стандарте UTC, и принадлежат соответствующим диапазонам. Значение, выходящее за пределы диапазона, интерпретируется правильно — например, 32 января эквивалентно 1 февраля. Диапазоны определяются следующим образом:

год год после 1900, со всеми цифрами

месяц 0–11  дата день месяца, 1–31 час 0–23 минуты 0–59

секунды 0–61 (с учетом переходной секунды).

# Класс Date

Класс **Date** прост в использовании, но содержит много методов:

public **Date()**

Создает объект Date, соответствующий текущей дате/времени.

public **int Date(int year, int month, int date, int hrs, int min, sec)**

Создает объект Date, соответствующий заданной дате/времени.

public **Date(int year, int month, int date, int hrs, int min)**

Эквивалентно Date(year, month, date, hrs, min, 0), то есть началу текущей минуты.

public **Date(int year, int month, int date)**

Эквивалентно Date(year, month, date, 0, 0, 0), то есть полуночи заданной даты.

public **Date(String s)**

Создает дату из строки в соответствии с синтаксисом, принятым в методе parse

public static long **int hrs, int min, UTC(int year, int month, int date, int sec)**

Вычисляет значение в стандарте UTC для указанной даты.

public static long **parse(String s)**

Анализирует строку, представляющую время, и возвращает полученное значение. Метод может работать со многими форматами, но важнее всего, что он воспринимает даты в стандарте IETF: “Sat, 12 Aug 1995 13:30:00 GMT”.

# Класс Date

public **Date(long date)**
Создает объект-дату. Перед созданием объекта Date происходит нормализация полей. Метод воспринимает в качестве параметра значение, возвращаемое методами parse и UTC.
public int **getYear()**
Возвращает год, всегда следующий после 1900.
public int **getMonth()**
Возвращает значение месяца в диапазоне 0–11 (с января по декабрь соответственно).
public int **getDate()**
Возвращает число месяца.
public int **getDay()**
Возвращает день недели в диапазоне 0–6 (с воскресенья до субботы соответственно).
public int **getHours()**
Возвращает час в диапазоне 0–23 (значение 0 соответствует полуночи).
public int **getMinutes()**
Возвращает минуты в диапазоне 0–59.
public int **getSeconds()**
Возвращает секунды в диапазоне 0–61.
public long **getTime()**
Возвращает время в формате UTC.
public int **getTimezoneOffset()**
Возвращает смещение часового пояса в минутах. Результат учитывает время суток, и на него может влиять летнее время — если оно учитывается, то в зависимости от времени года может присутствовать дополнительное смещение часового пояса

# Класс Date

public void **setYear(int year)**
Устанавливает значение года. Год должен быть после 1900.
public void **setMonth(int month)**
Устанавливает месяц.
public void **setDate(int date)**
Устанавливает число месяца.
public void **setDay(int day)**
Устанавливает день недели.
public void **setHours(int hours)**
Устанавливает час.
public void **setMinutes(int minutes)**
Устанавливает минуты.
public void **setSeconds(int seconds)**
Устанавливает секунды.
public boolean **before(Date other)**
Возвращает true, если дата объекта наступает раньше даты other.
public boolean **after(Date other)**
Возвращает true, если дата объекта наступает после даты other.
public boolean **equals(Object other)**
Возвращает true, если дата объекта представляет в стандарте UTC ту же дату, что и other.

# Класс Date

public int **hashCode()**

Вычисляет хеш-код, чтобы объекты Date могли использоваться в качестве ключей в хеш-таблицах.

public String **toString()**

Преобразует дату в String, например: “Fri Oct 13 14:33:57 EDT 1995". /Формат строки совпадает с форматом, используемым в функции ctime в соответствии со стандартом ANSI C./

public String **toLocaleString()**

Преобразует дату в String с использованием национального формата. Другими словами, дата будет представлена в виде, принятом в локализованной операционной системе. Например, жители США привыкли видеть месяц перед числом (“June 13"), тогда как в Европе обычно используется обратный порядок (”13 June ").

public String **toGMTString()**

Преобразует дату в String с использованием конвенции Internet GMT, в форме

d mon yyyy hh:mm:ss GMT

где d — число месяца (одна или две цифры), mon — первые три буквы месяца, yyyy — год из четырех цифр, hh — часы (0–23), mm — минуты, а ss — секунды. Информация о местном часовом поясе при этом игнорируется.

# 12.11. Класс Random

Объекты класса Random предназначены для работы с независимыми последовательностями псевдослучайных чисел. Если вам нужна последовательность типа double и вас не интересует порядок следования чисел, можно воспользоваться методом java.lang.Math.random — он создает объект Random при первом вызове и в дальнейшем возвращает псевдослучайные числа из этого объекта.

Чтобы иметь больше средств для контроля за последовательностью создайте объект Random и получайте числа от него.

public **Random()**

Создает новый генератор случайных чисел. Стартовое значение определяется на основании текущего времени.

public **Random(long seed)**

Создает новый генератор случайных чисел с заданным стартовым значением. Два объекта Random, созданные с одинаковым seed, будут порождать совпадающие последовательности псевдослучайных чисел.

public synchronized void **setSeed(long seed)**

Устанавливает стартовое значение генератора случайных чисел равным seed. Метод может быть вызван в любой момент — в результате произойдет сброс последовательности и последующее ее порождение на основе стартового значения.

# Класс Random

public int **nextInt()**

Возвращает псевдослучайное значение типа int, равномерно распределенное между величинами Integer.MIN_VALUE и Integer.MAX_VALUE включительно.

public long **nextLong()**

Возвращает псевдослучайное значение типа long, равномерно распределенное между величинами Long.MIN_VALUE и Long.MAX_VALUE включительно.

public float **nextFloat()**

Возвращает псевдослучайное значение типа float, равномерно распределенное между величинами Float.MIN_VALUE и Float.MAX_VALUE включительно.

public double **nextDouble()**

Возвращает псевдослучайное значение типа double, равномерно распределенное между величинами Double.MIN_VALUE и Double.MAX_VALUE включительно.

public synchronized double **nextGaussian()**

Возвращает псевдослучайное значение типа double, подчиняющееся распределению Гаусса, с математическим ожиданием 0,0 и стандартным отклонением 1,0.

# 12.12. Класс String Tokenizer

**Класс StringTokenizer** делит строку на части, используя для этого символы-разделители. Последовательность лексем, выделенных из строки, фактически представляет собой упорядоченный объект-перечисление, поэтому класс **StringTokenizer** реализует интерфейс **Enumeration**.

**StringTokenizer** также предоставляет ряд методов с более конкретной типизацией. Перечисление **StringTokenizer** не гарантирует фиксации исходного состояния, но это не имеет значения, поскольку объекты **String** доступны только для чтения. Например, для деления строки на лексемы, отделяемые запятыми и пробелами, может использоваться следующий цикл:

```
String str = "Gone, and forgotten";
StringTokenizer tokens = new StringTokenizer(str, " ,");
while (tokens.hasMoreTokens())
    System.out.println(tokens.nextToken());
```

Запятая включена в список разделителей в конструкторе **StringTokenizer** для того, чтобы анализатор “поглощал” запятые вместе с пробелами, оставляя только слова, которые возвращаются по одному

# Класс String Tokenizer

Класс **StringTokenizer** содержит несколько методов, которые определяют, что считать словом, следует ли отдельно обрабатывать строки и числа, и так далее:

public **StringTokenizer(String str, String delim, boolean returnTokens)**

Конструирует объект StringTokenizer для строки str с использованием символов из строки delim в качестве разделителей. Логическое значение returnTokens определяет, следует ли возвращать разделители как лексемы или же пропускать их. В первом случае каждый символ-разделитель возвращается отдельно.

public **StringTokenizer(String str, String delim)**

Эквивалентен StringTokenizer(str, delim, false), то есть разделители пропускаются.

public **StringTokenizer(String str)**

Эквивалентен StringTokenizer(str, " \t\n\r"), то есть используются стандартные символы-разделители.

public boolean **HasMoreTokens()**

Возвращает true, если в строке еще остаются лексемы.

public String **nextToken()()**

Возвращает следующую лексему в строке. Если лексем больше нет, возбуждается исключение NoSuchElementException.

public String **nextToken(String delim)**

Заменяет набор символов-разделителей на символы из строки delim и возвращает следующую лексему.

# Класс String Tokenizer

public int countTokens()

Возвращает количество лексем, остающихся в строке при использовании текущего набора разделителей. Оно равно числу возможных вызовов **next Token** перед тем, как будет возбуждено исключение.

Если понадобилось узнать количество лексем, то этот метод работает быстрее циклического вызова **nextToken**, поскольку строки-лексемы только подсчитываются, без расходов на конструирование и возврат значения.

Два метода класса **StringTokenizer,** унаследованные от интерфейса **Enumeration (hasMoreElements и nextElement),** эквивалентны методам **hasMoreTokens и nextToken** соответственно.

Если вам понадобится более мощный механизм для деления строки или другого входного значения на лексемы, обратитесь к разделу "**Класс Stream Tokenizer",** в котором описывается класс с большими возможностями по части распознавания ввода. Чтобы воспользоваться классом **Stream Tokenizer** для строки, создайте для нее объект **StringBufferInputStream.** Тем не менее во многих случаях бывает достаточно и простого класса **String Tokenizer.**

**Упражнение 12.9**

Напишите метод, который получает строку, делит ее на лексемы с использованием стандартных символов-разделителей и возвращает новую строку, в которой первая буква каждого слова преобразована в заглавный регистр