Пакет Java.util

Эта лабораторная работа посвящена пакету **Java**.util, в котором содержится множество вспомогательных классов и интерфейсов. Они настолько удобны, что практически любая программа использует эту библиотеку. Центральную часть в изложении занимает тема контейнеров, или коллекций, – классов, хранящих упорядоченные ссылки на ряд объектов. Они были существенно переработаны в ходе создания версии Java2. Также рассматриваются классы для работы с датой, для генерации случайных чисел, обеспечения поддержки многих национальных языков в приложении и др.

Работа с датами и временем

Класс Date

Класс **Date** изначально предоставлял набор функций для работы с датой – для получения текущего года, месяца и т.д. Однако сейчас все перечисленные методы не рекомендованы к использованию и практически всю функциональность для этого предоставляет класс **Calendar**.

Существует несколько конструкторов класса **Date**, однако рекомендовано к использованию два:

Date() и **Date**(**long Date**)

Второй конструктор принимает в качестве параметра значение типа **long**, указывающее на количество миллисекунд, прошедших с 1 января 1970 г., 00:00:00 по Гринвичу. Первый конструктор создает экземпляр, соответствующий текущему моменту. Фактически это эквивалентно второму варианту new **Date**(System.currentTimeMillis()). Можно уже после создания экземпляра класса **Date** использовать метод setTime(**long** time) для того, чтобы задать нужное время.

Для сравнения дат служат методы **after**(**Date Date**) и **before**(**Date Date**), которые возвращают булевское значение, в зависимости от того, выполнено условие или нет. Метод **compareTo**(**Date** anotherDate) возвращает значение типа **int**, которое равно -1, если дата меньше сравниваемой, 1 – если больше и 0 – если даты равны. Метод **toString**() возвращает строковое описание даты. Однако для более понятного и удобного преобразования даты в текст рекомендуется пользоваться классом **SimpleDateFormat**, определенным в пакете **Java.text**.

Классы Calendar и GregorianCalendar

Более развитые средства для работы с датами представляет класс **Calendar**. **Calendar** является абстрактным классом. Для различных платформ реализуются конкретные подклассы календаря. На данный момент существует реализация Григорианского календаря – **GregorianCalendar**. Экземпляр этого класса получается путем вызова статического метода getInstance(), который возвращает экземпляр класса **GregorianCalendar**. Подклассы класса **Calendar** должны интерпретировать объект **Date** по-разному. В будущем предполагается реализовать также лунный календарь, используемый в некоторых странах.

Calendar обеспечивает набор методов, позволяющих манипулировать различными "частями" даты, т.е. получать и устанавливать дни, месяцы, недели и т.д.

Если при задании параметров календаря некоторые параметры упущены, то для них будут использованы значения по умолчанию для начала отсчета, т.е.

```
YEAR = 1970, MONTH = JANUARY, DATE = 1 и т.д.
```

Для считывания и установки различных "частей" даты используются методы **get**(**int** field), **set**(**int** field, **int** value), **add**(**int** field, **int** amount), roll(**int** field, **int** amount), переменная типа **int** с именем field указывает на номер поля, с которым нужно произвести операцию. Для удобства все эти поля определены в **Calendar** как статические константы типа **int**.

Рассмотрим подробнее порядок выполнения перечисленных методов.

Метод set(int field,int value).

Как уже говорилось, данный метод производит установку какого-либо поля даты. На самом деле после вызова этого метода немедленного пересчета даты не производится. Пересчет даты будет осуществлен только после вызова методов **get**(), **getTime**() или **getTimeInMillis**(). Таким образом, последовательная установка нескольких полей не вызовет ненужных вычислений. Помимо этого, появляется еще один интересный эффект. Рассмотрим следующий пример. Предположим, что дата установлена на последний день августа. Необходимо перевести ее на последний день сентября. Если бы внутреннее представление даты изменялось после вызова метода **set**, то при последовательной установке полей мы получили бы вот такой эффект:

```
public class Test {
  public Test() {
  }
  public static void main(String[] args) {
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMMM dd HH:mm:ss");
    Calendar cal = Calendar.getInstance();
    cal.set(Calendar.YEAR,2002);
    cal.set(Calendar.MONTH,Calendar.AUGUST);
    cal.set(Calendar.DAY_OF_MONTH,31);
    System.out.println(" Initialy set Date: " + sdf.format(cal.getTime()));
    cal.set(Calendar.MONTH,Calendar.SEPTEMBER);
    System.out.println(" Date with month changed : " +
sdf.format(cal.getTime()));
    cal.set(Calendar.DAY_OF_MONTH,30);
    System.out.println(" Date with day changed : " +
sdf.format(cal.getTime()));

  }
}
```

Пример 1.

Результатом будет:

```
Initialy set Date: 2002 August 31 22:57:47
Date with month changed : 2002 October 01 22:57:47
Date with day changed : 2002 October 30 22:57:47
```

Как мы видим, в данном примере при изменении месяца день месяца остался неизменным и было унаследовано его предыдущее значение. Но поскольку в сентябре 30 дней, дата

автоматически была переведена на 1 октября, и когда было бы установлено 30 число, оно относилось бы уже к октябрю. В следующем примере считывание даты не производится, соответственно, ее вычисление не выполняется до тех пор, пока все поля не установлены:

```
public class Test {
  public Test() {
  }
  public static void main(String[] args) {
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMMM dd HH:mm:ss");

    Calendar cal = Calendar.getInstance();
    cal.set(Calendar.YEAR,2002);
    cal.set(Calendar.MONTH,Calendar.AUGUST);
    cal.set(Calendar.DAY_OF_MONTH,31);
    System.out.println(" Initialy set Date: " + sdf.format(cal.getTime()));
   cal.set(Calendar.MONTH,Calendar.SEPTEMBER);
   cal.set(Calendar.DAY_OF_MONTH,30);
   System.out.println(" Date with day AND month changed : " +
sdf.format(cal.getTime()));
  }
}
```

Пример 3.

Результатом будет:

```
Initialy set Date: 2002 August 31 23:03:51
Date with day AND month changed: 2002 September 30 23:03:51
```

Метод add(int field,int delta).

Добавляет некоторое смещение к существующей величине поля. В принципе, то же самое можно сделать с помощью **set**(f, **get**(f) + delta).

В случае использования метода **add** следует помнить о двух правилах:

1. Если величина поля изменения выходит за диапазон возможных значений данного поля, то производится деление по модулю данной величины, частное суммируется со следующим по старшинству полем.
2. Если изменяется одно из полей, причем, после изменения младшее по отношению к изменяемому полю принимает некорректное значение, то оно изменяется на то, которое максимально близко к "старому".

```
public class Test {

   public Test() {
   }
   public static void main(String[] args) {
      SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMMM dd HH:mm:ss");
      Calendar cal = Calendar.getInstance();
      cal.set(Calendar.YEAR,2002);
      cal.set(Calendar.MONTH,Calendar.AUGUST);
      cal.set(Calendar.DAY_OF_MONTH,31);
      cal.set(Calendar.HOUR_OF_DAY,19);
      cal.set(Calendar.MINUTE,30);
      cal.set(Calendar.SECOND,00);
      System.out.println("Current Date: " + sdf.format(cal.getTime()));
      cal.add(Calendar.SECOND,75);
      System.out.println("Current Date: " + sdf.format(cal.getTime()));
      cal.add(Calendar.MONTH,1);
```

```
      System.out.println("Current Date: " + sdf.format(cal.getTime()));
   }
}
```

Пример 5.

Результатом будет:

```
Current Date: 2002 August 31 19:30:00
Rule 1: 2002 August 31 19:31:15
Rule 2: 2002 September 30 19:31:15
```

Метод roll(int field,int delta).

Добавляет некоторое смещение к существующей величине поля и не производит изменения старших полей. Рассмотрим приведенный ранее пример, но с использованием метода roll.

```
public class Test {

   public Test() {
   }
   public static void main(String[] args) {
      SimpleDateFormat sdf = new SimpleDateFormat("yyyy MMMM dd HH:mm:ss");
      Calendar cal = Calendar.getInstance();
      cal.set(Calendar.YEAR,2002);
      cal.set(Calendar.MONTH,Calendar.AUGUST);
      cal.set(Calendar.DAY_OF_MONTH,31);
      cal.set(Calendar.HOUR_OF_DAY,19);
      cal.set(Calendar.MINUTE,30);
      cal.set(Calendar.SECOND,00);
      System.out.println("Current Date: " + sdf.format(cal.getTime()));
      cal.roll(Calendar.SECOND,75);
      System.out.println("Rule 1: " + sdf.format(cal.getTime()));
      cal.roll(Calendar.MONTH,1);
      System.out.println("Rule 2: " + sdf.format(cal.getTime()));
   }
}
```

Пример 7.

Результатом будет:

```
Current Date: 2002 August 31 19:30:00
Rule 1: 2002 August 31 19:30:15
Rule 2: 2002 September 30 19:30:15
```

Как видно из результатов работы приведенного выше кода, действие правила 1 изменилось по сравнению с методом **add**, а правило 2 действует так же.

Класс TimeZone

Класс **TimeZone** предназначен для совместного использования с классами **Calendar** и DateFormat. Класс абстрактный, поэтому от него порождать объекты нельзя. Вместо этого определен статический метод **getDefault**(), который возвращает экземпляр наследника **TimeZone** с настройками, взятыми из операционной системы, под управлением которой работает JVM. Для того, чтобы указать произвольные параметры, можно воспользоваться статическим методом **getTimeZone**(**String** ID), в качестве параметра которому передается наименование конкретного временного пояса, для которого необходимо получить объект

TimeZone. Набор полей, определяющих возможный набор параметров для **getTimeZone**, нигде явно не описывается. Вместо этого определен статический метод **String**[] **getAvailableIds**(), который возвращает возможные значения для параметра **getTimeZone**. Так можно определить набор возможных параметров для конкретного временного пояса (рассчитывается относительно Гринвича) **String**[] **getAvailableIds**(**int** offset).

Рассмотрим пример, в котором на консоль последовательно выводятся:

- временная зона по умолчанию;
- список всех возможных временных зон;
- список временных зон, которые совпадают с текущей временной зоной.

```
public class Test {
   public Test() {
   }
   public static void main(String[] args) {
      Test test = new Test();
      TimeZone tz = TimeZone.getDefault();
      int rawOffset = tz.getRawOffset();
      System.out.println("Current TimeZone" + tz.getDisplayName() +
tz.getID() + "\n\n");
      // Dispaly all available TimeZones
      System.out.println("All Available TimeZones \n");
      String[] idArr = tz.getAvailableIds();
      for(int cnt=0;cnt < idArr.length;cnt++){
      tz = TimeZone.getTimeZone(idArr[cnt]);
      System.out.println(test.padr(tz.getDisplayName() +
         tz.getID(),64) + " raw offset=" + tz.getRawOffset() +
         ";hour offset=(" + tz.getRawOffset()/ (1000 * 60 * 60 ) + ")");
      }
      // Dispaly all available TimeZones same as for Moscow
      System.out.println("\n\n TimeZones same as for Moscow \n");
      idArr = tz.getAvailableIds(rawOffset);
      for(int cnt=0;cnt < idArr.length;cnt++){
         tz = TimeZone.getTimeZone(idArr[cnt]);
         System.out.println(test.padr(tz.getDisplayName()+
            tz.getID(),64) + " raw offset=" + tz.getRawOffset() +
            ";hour offset=(" + tz.getRawOffset()/ (1000 * 60 * 60 ) + ")");
      }
   }
   String padr(String str,int len){
      if(len - str.length() > 0){
         char[] buf = new char[len - str.length()];
         Arrays.fill(buf,' ');
         return str + new String(buf);
      }else{
         return str.substring(0,len);
      }
   }
}
```

Пример 9.

Результатом будет:

```
Current TimeZone Moscow Standard TimeEurope/Moscow
TimeZones same as for Moscow
Eastern African TimeAfrica/Addis_Aba raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Asmera raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Dar_es_Sa raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Djibouti raw offset=10800000;hour offset=(3)
```

```
Eastern African TimeAfrica/Kampala raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Khartoum raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Mogadishu raw offset=10800000;hour offset=(3)
Eastern African TimeAfrica/Nairobi raw offset=10800000;hour offset=(3)
Arabia Standard TimeAsia/Aden raw offset=10800000;hour offset=(3)
Arabia Standard TimeAsia/Baghdad raw offset=10800000;hour offset=(3)
Arabia Standard TimeAsia/Bahrain raw offset=10800000;hour offset=(3)
Arabia Standard TimeAsia/Kuwait raw offset=10800000;hour offset=(3)
Arabia Standard TimeAsia/Qatar raw offset=10800000;hour offset=(3)
Arabia Standard TimeAsia/Riyadh raw offset=10800000;hour offset=(3)
Eastern African TimeEAT raw offset=10800000;hour offset=(3)
Moscow Standard TimeEurope/Moscow raw offset=10800000;hour offset=(3)
Eastern African TimeIndian/Antananar raw offset=10800000;hour offset=(3)
Eastern African TimeIndian/Comoro raw offset=10800000;hour offset=(3)
Eastern African TimeIndian/Mayotte raw offset=10800000;hour offset=(3)
```

Класс SimpleTimeZone

Класс **SimpleTimeZone**, как потомок **TimeZone**, реализует его абстрактные методы и предназначен для применения в настройках, использующих Григорианский календарь. В большинстве случаев нет необходимости создавать экземпляр данного класса с помощью конструктора. Вместо этого лучше использовать статические методы, которые возвращают тип **TimeZone**, рассмотренные в предыдущем параграфе. Единственная, пожалуй, причина для использования конструктора – необходимость задания нестандартных правил перехода на зимнее и летнее время.

В классе **SimpleTimeZone** определено три конструктора. Рассмотрим наиболее полный с точки зрения функциональности вариант, который, помимо временной зоны, задает летнее и зимнее время.

```
public SimpleTimeZone(int rawOffset,
      String ID,
      int startMonth,
      int startDay,
      int startDayOfWeek,
      int startTime,
      int endMonth,
      int endDay,
      int endDayOfWeek,
   int endTime)
```

rawOffset – временное смещение относительно гринвича;

ID – идентификатор временной зоны (см. пред.параграф);

startMonth – месяц перехода на летнее время;

startDay – день месяца перехода на летнее время*;

startDayOfWeek – день недели перехода на летнее время*;

startTime – время перехода на летнее время (указывается в миллисекундах);

endMonth – месяц окончания действия летнего времени;

endDay – день окончания действия летнего времени*;

endDayOfWeek– день недели окончания действия летнего времени*;

endTime – время окончания действия летнего времени (указывается в миллисекундах).

Перевод часов на зимний и летний вариант исчисления времени определяется специальным правительственным указом. Обычно переход на летнее время происходит в 2 часа в последнее воскресенье марта, а переход на зимнее время – в 3 часа в последнее воскресенье октября.

Алгоритм расчета таков:

- если startDay=1 и установлен день недели, то будет вычисляться первый день недели startDayOfWeek месяца startMonth (например, первое воскресенье);
- если startDay=-1 и установлен день недели, то будет вычисляться последний день недели startDayOfWeek месяца startMonth (например, последнее воскресенье);
- если день недели startDayOfWeek установлен в 0, то будет вычисляться число startDay конкретного месяца startMonth;
- для того, чтобы установить день недели после конкретного числа, специфицируется отрицательное значение дня недели. Например, чтобы указать первый понедельник после 23 февраля, используется вот такой набор: startDayOfWeek=-MONDAY, startMonth=FEBRUARY, startDay=23
- для того, чтобы указать последний день недели перед каким-либо числом, указывается отрицательное значение этого числа и отрицательное значение дня недели. Например, для того, чтобы указать последнюю субботу перед 23 февраля, необходимо задать такой набор параметров: startDayOfWeek=-SATURDAY, startMonth=FEBRUARY, startDay=-23;
- все вышеперечисленное относится также и к окончанию действия летнего времени.

Рассмотрим пример получения текущей временной зоны с заданием перехода на зимнее и летнее время для России по умолчанию.

```
public class Test {
   public Test() {
   }
   public static void main(String[] args) {
      Test test = new Test();
      SimpleTimeZone stz = new SimpleTimeZone(
         TimeZone.getDefault().getRawOffset()
         ,TimeZone.getDefault().getID()
         ,Calendar.MARCH
         ,-1
         ,Calendar.SUNDAY
         ,test.getTime(2,0,0,0)
         ,Calendar.OCTOBER
         ,-1
         ,Calendar.SUNDAY
         ,test.getTime(3,0,0,0)
         );
   System.out.println(stz.toString());
   }
   int getTime(int hour,int min,int sec,int ms){
      return hour * 3600000 + min * 60000 + sec * 1000 + ms;
   }
}
```

Пример 11.

Результатом будет:

```
Java.util.SimpleTimeZone[id=Europe/Moscow,offset=10800000,dstSavings=3600000,
useDaylight=true,
startYear=0,startMode=2,startMonth=2,startDay=-
1,startDayOfWeek=1,startTime=7200000,startTimeMode=0,
endMode=2,endMonth=9,endDay=-1,endDayOfWeek=1,endTime=10800000,endTimeMode=0]
```

Интерфейс Observer и класс Observable

Интерфейс **Observer** определяет всего один метод, **update** (**Observable** o, **Object** arg), который вызывается, когда обозреваемый объект изменяется.

Класс **Observable** предназначен для поддержки обозреваемого объекта в парадигме MVC (model-view-controller), которая, как и другие проектные решения и шаблоны, описана в специальной литературе. Этот класс должен быть унаследован, если возникает необходимость в том, чтобы отслеживать состояние какого-либо объекта. Обозреваемый объект может иметь несколько обозревателей. Соответственно, они должны реализовать интерфейс **Observer**.

После того, как в состоянии обозреваемого объекта что-то меняется, необходимо вызвать метод **notifyObservers**, который, в свою очередь, вызывает методы **update** у каждого обозревателя.

Порядок, в котором вызываются методы **update** обозревателей, заранее не определен. Реализация по умолчанию подразумевает их вызов в порядке регистрации. Регистрация осуществляется с помощью метода **addObserver**(**Observer** o). Удаление обозревателя из списка выполняется с помощью **deleteObserver**(**Observer** o). Перед вызовом **notifyObservers** необходимо вызвать метод **setChanged**, который устанавливает признак того, что обозреваемый объект был изменен.

Рассмотрим пример организации взаимодействия классов:

```
public class TestObservable extends Java.util.Observable {
   private String name = "";
   public TestObservable(String name) {
      this.name = name;
   }

   public void modify() {
      setChanged();
   }

   public String getName() {
      return name;
   }
}

public class TestObserver implements Java.util.Observer {
   private String name = "";

   public TestObserver(String name) {
      this.name = name;
   }
   public void update(Java.util.Observable o,Object arg) {
      String str = "Called update of " + name;
      str += " from " + ((TestObservable)o).getName();
```

```
        str += " with argument " + (String)arg;
        System.out.println(str);
    }
}
public class Test {
    public Test() {
    }
    public static void main(String[] args) {
        Test test = new Test();
        TestObservable to = new TestObservable("Observable");
        TestObserver o1 = new TestObserver("Observer 1");
        TestObserver o2 = new TestObserver("Observer 2");
        to.addObserver(o1);
        to.addObserver(o2);
        to.modify();
        to.notifyObservers("Notify argument");
    }
}
```

Пример 13.

В результате работы на консоль будет выведено:

```
Called update of Observer 2 from Observable with argument Notify argument
Called update of Observer 1 from Observable with argument Notify argument
```

На практике использовать **Observer** не всегда удобно, так как в **Java** отсутствует множественное наследование и **Observer** необходимо наследовать в самом начале построения иерархии классов. Как вариант, можно предложить определить интерфейс, задающий функциональность, сходную с **Observer**, и реализовать его в подходящем классе.

Коллекции

Зачастую в программе работа идет не с одним объектом, а с целой группой более или менее однотипных экземпляров (например, автопарк организации). Проще всего сделать это с помощью массивов. Однако, несмотря на то, что это достаточно эффективное решение для многих случаев, оно имеет некоторые ограничения. Так, обращаться к элементу массива можно только по его номеру (индексу). Также необходимо заранее задать длину массива и больше ее не менять.

Массивы существовали в **Java** изначально. Кроме того, было определено два класса для организации более эффективной работы с наборами объектов: **Hashtable** и **Vector**. В JDK 1.2 набор классов, поддерживающих работу с коллекциями, был существенно расширен.

Существует несколько различных типов классов-коллекций. Все они разрабатывались, по возможности, в соответствии с единой логикой и определенными интерфейсами и там, где это возможно, работа с ними унифицирована. Однако все коллекции отличаются внутренними механизмами хранения, скоростью доступа к элементам, потребляемой памятью и другими деталями. Например, в некоторых коллекциях объекты (также называемые элементами коллекций), могут быть упорядочены, в некоторых – нет. В некоторых типах коллекций допускается дублирование ссылок на объект, в некоторых – нет. Далее мы рассмотрим каждый из классов-коллекций.

Классы, обеспечивающие манипулирование коллекциями объектов, объявлены в пакете **Java.util.**

Интерфейсы

Интерфейс Collection

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

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

Интерфейс Set

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

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

Интерфейс List

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

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

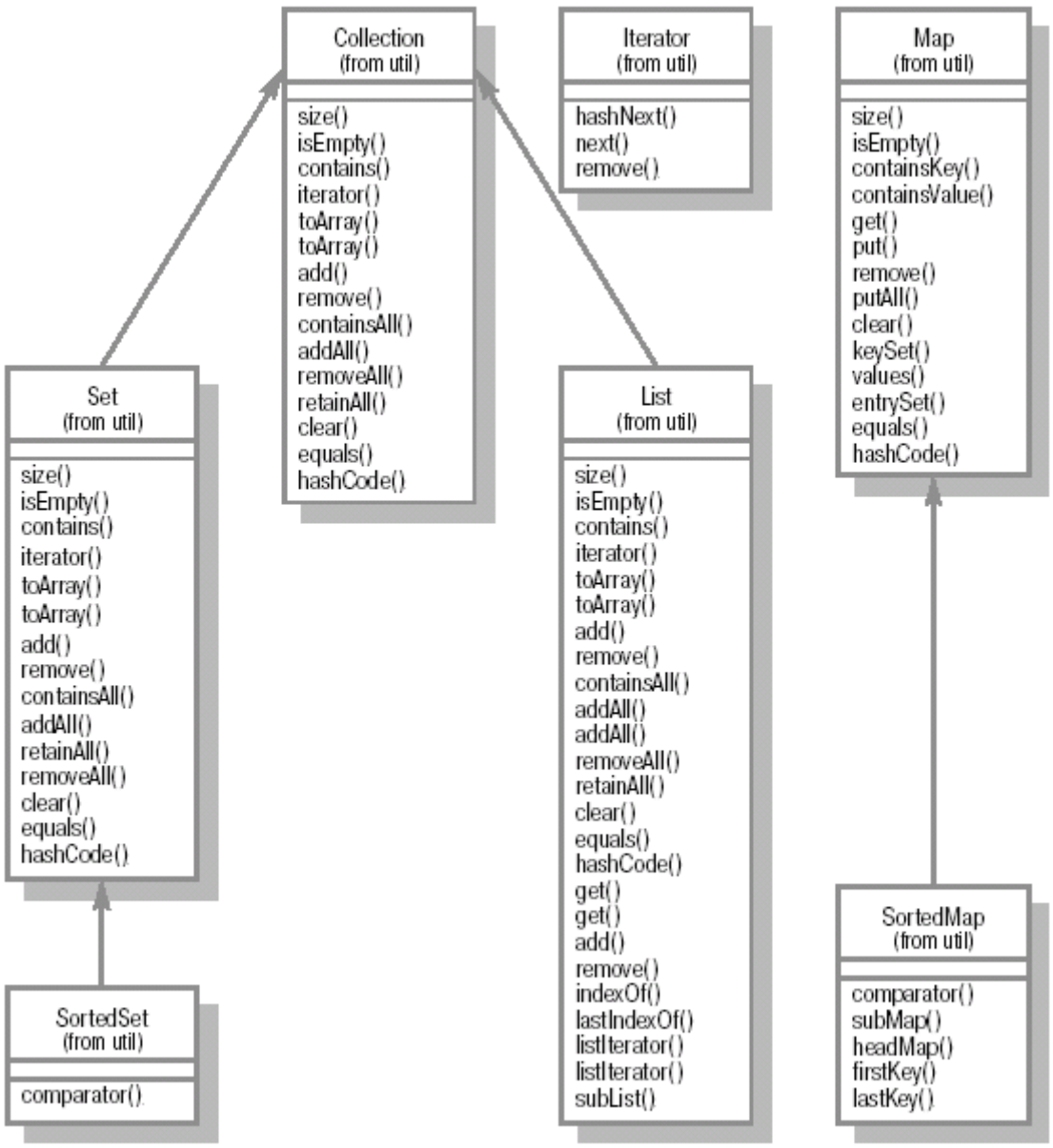

Рис. 1. Основные типы для работы с коллекциями.

Интерфейс Map

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

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

Интерфейс SortedSet

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

Интерфейс SortedMap

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

Интерфейс Iterator

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

Таким образом, подводя итог, перечислим интерфейсы, используемые при работе с коллекциями:

```
Java.util.Collection
Java.util.Set
Java.util.List
Java.util.Map
Java.util.SortedSet
Java.util.SortedMap
Java.util.Iterator
```

Абстрактные классы, используемые при работе с коллекциями

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

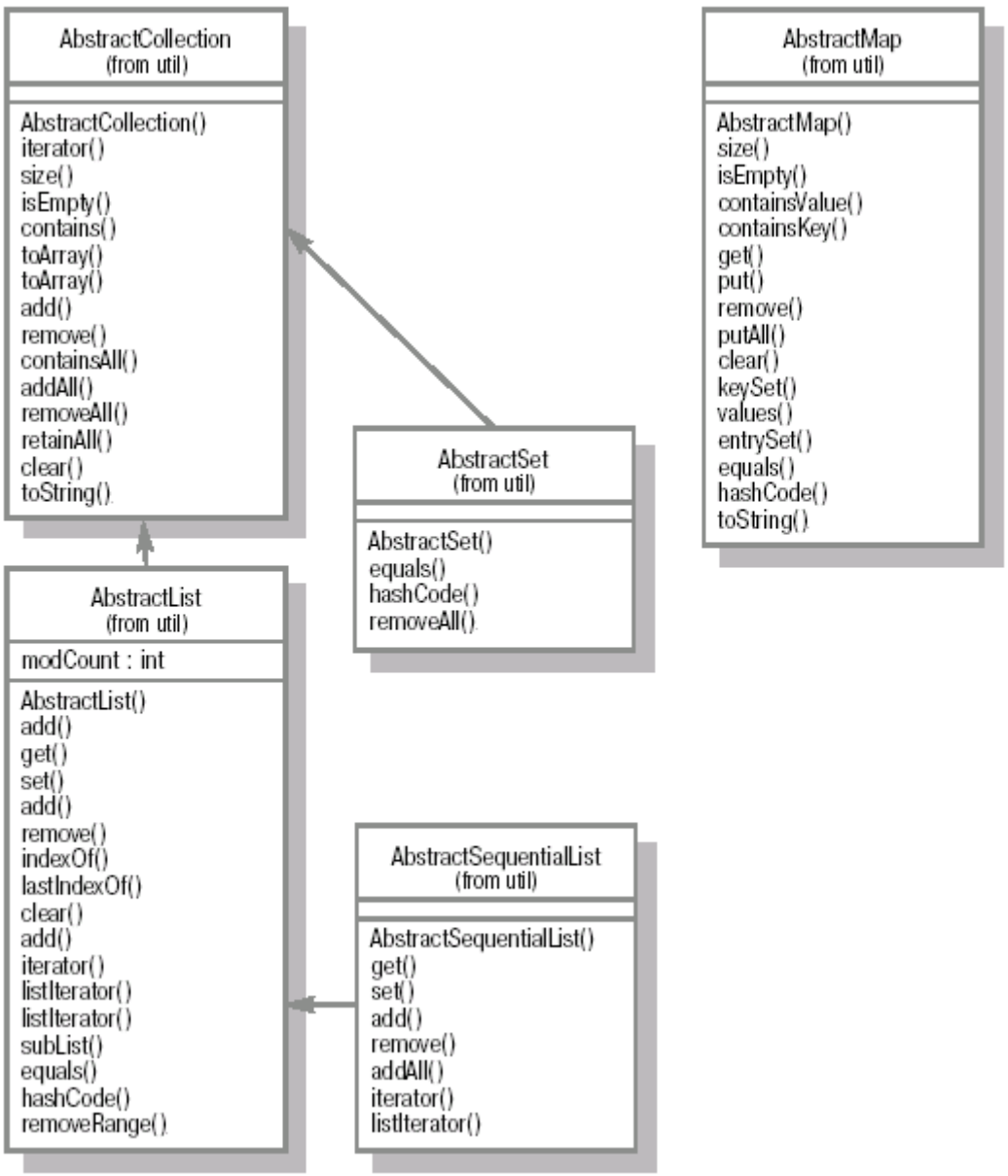

Рис. 2. Базовые абстрактные классы.

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

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

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

Java.util.AbstractSet – данный класс расширяет **AbstractCollection** и реализует основную функциональность, определенную в интерфейсе **Set**. Следует отметить, что этот класс не переопределяет функциональность, реализованную в классе **AbstractCollection**.

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

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

Конкретные классы коллекций

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

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

Пример 15.

Результатом будет:

```
First element
Second element
Third element

Firts element
Second element
Insertion
Third element
```

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

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

Пример 14.17.

Результатом будет:

```
Element3
Element2
Element1
Element4

Element3
Element2
Element4

Element3
Element2
```

Классы **LinkedList** и **ArrayList** имеют схожую функциональность. Однако с точки зрения производительности они отличаются. Так, в **ArrayList** заметно быстрей (примерно на порядок) осуществляются операции прохода по всему списку (итерации) и получения

данных. **LinkedList** почти на порядок быстрее выполняет операции удаления и добавления новых элементов.

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

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

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

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

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

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

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

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

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

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

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

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

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

```
      System.out.println("Iteration over keySet");
      String key = "";
      // Итератор для перебора всех ключей в Map
      it = hm.keySet().Iterator();
      while(it.hasNext()){
         key = (String)it.next();
         System.out.println( "For key = " + key + " value = " +
            ((TestObject)hm.get(key)).getText());
      }
   }
}
```

Пример 19.

Результатом будет:

```
Object value for Key1 = Value 1

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

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

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

```
public class Test {

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

Результатом будет:

```
String1
String2
```

Класс Collections

Класс **Collections** является классом-утилитой и содержит несколько вспомогательных методов для работы с классами, обеспечивающими различные интерфейсы коллекций. Например, для сортировки элементов списков, для поиска элементов в упорядоченных

коллекциях и т.д. Но, пожалуй, наиболее важным свойством этого класса является возможность получения синхронизированных вариантов классов-коллекций. Например, для получения синхронизированного варианта **Map** можно использовать следующий подход:

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

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

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

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

Пример 21.

Класс Properties

Класс **Properties** предназначен для хранения набора свойств (параметров). Методы

```
String getProperty(String key)
```

```
String getProperty(String key,
                   String defaultValue)
```

позволяют получить свойство из набора.

С помощью метода **setProperty**(**String** key, **String** value) это свойство можно установить.

Метод **load**(InputStream **inStream**) позволяет загрузить набор свойств из входного потока. Как правило, это текстовый файл, в котором хранятся параметры. Параметры – это строки, которые представляют собой пары ключ/значение. Предполагается, что по умолчанию используется кодировка **ISO** 8859-1. Каждая строка должна оканчиваться символами \r,\n или \r\n. Строки из файла будут считываться до тех пор, пока не будет достигнут его конец. Строки, состоящие из одних пробелов, или начинающиеся со знаков ! или #, игнорируются, т.е. их можно трактовать как комментарии. Если строка оканчивается символом /, то следующая строка считается ее продолжением. Первый символ с начала строки, отличный от пробела, считается началом ключа. Первый встретившийся пробел, двоеточие или знак равенства считается окончанием ключа. Все символы окончания ключа при необходимости могут быть включены в название ключа, но при этом перед ними должен стоять символ \. После того, как встретился символ окончания ключа, все аналогичные символы будут проигнорированы до начала значения. Оставшаяся часть строки интерпретируется как значение. В строке, состоящей только из символов \t, \n, \r, \\, \", \', \ и \uxxxx, они все распознаются и интерпретируются как одиночные символы. Если встретится сочетание \ и символа конца строки, то следующая строка будет считаться продолжением текущей, также будут проигнорированы все пробелы до начала строки-продолжения.

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

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

```
public class Test {
   public Test() {
   }
   public static void main(String[] args) {
      Test test = new Test();
      Properties props = new Properties();
      StringWriter sw = new StringWriter();
      sw.write("Key1 = Vlaue1 \n");
      sw.write(" Key2 : Vlaue2 \r\n");
      sw.write(" Key3 Vlaue3 \n ");
      InputStream is = new ByteArrayInputStream(sw.toString().getBytes());

      try {
         props.load(is);
      }
      catch (IOException ex) {
```

```
            ex.printStackTrace();
         }
         props.List(System.out);
         props.setProperty("Key1","Modified Value1");
         props.setProperty("Key4","Added Value4");
         props.List(System.out);
      }
}
```

Пример 22.

Результатом будет:

```
-- Listing Properties --
Key3=Vlaue3
Key2=Vlaue2
Key1=Vlaue1

-- Listing Properties --
Key4=Added Value4
Key3=Vlaue3
Key2=Vlaue2
Key1=Modified Value1
```

Интерфейс Comparator

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

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

Класс Arrays

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

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

```
      System.out.println();
      for(int cnt=0;cnt < arr.length;cnt++) {
         System.out.println(arr[cnt]);
      }
   }
}
```

Класс StringTokenizer

Этот класс предназначен для разбора строки по лексемам (**tokens**). Строка, которую необходимо разобрать, передается в качестве параметра конструктору **StringTokenizer**(**String** str). Определено еще два перегруженных конструктора, которым дополнительно можно передать строку-разделитель лексем **StringTokenizer**(**String** str, **String** delim) и признак возврата разделителя лексем **StringTokenizer**(**String** str, **String** delim, **Boolean** returnDelims).

Разделителем лексем по умолчанию служит пробел.

```
public class Test {

   public Test() {
   }
   public static void main(String[] args) {
      Test test = new Test();
      String toParse =
         "word1;word2;word3;word4";
      StringTokenizer st =
         new StringTokenizer(toParse,";");
      while(st.hasMoreTokens()){
         System.out.println(st.nextToken());
      }
   }
}
```

Результатом будет:

```
word1
word2
word3
word4
```

Класс BitSet

Класс **BitSet** предназначен для работы с последовательностями битов. Каждый компонент этой коллекции может принимать булево значение, которое обозначает, установлен бит или нет. Содержимое **BitSet** может быть модифицировано содержимым другого **BitSet** с использованием операций **AND**, **OR** или **XOR** (исключающее или).

BitSet имеет текущий размер (количество установленных битов), может динамически изменяться. По умолчанию все биты в наборе устанавливаются в 0 (**false**). Установка и очистка битов в **BitSet** осуществляется методами **set**(**int** index) и clear(**int** index).

Метод **int length**() возвращает "логический" размер набора битов, **int size**() возвращает количество памяти, занимаемой битовой последовательностью **BitSet**.

```
public class Test {
```

```
   public Test() {
   }
   public static void main(String[] args) {
      Test test = new Test();
      BitSet bs1 = new BitSet();
      BitSet bs2 = new BitSet();
      bs1.set(0);
      bs1.set(2);
      bs1.set(4);
      System.out.println("Length = " +
         bs1.length()+" size = "+bs1.size());
      System.out.println(bs1);
      bs2.set(1);
      bs2.set(2);
      bs1.AND(bs2);
      System.out.println(bs1);
   }
}
```

Результатом будет:

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

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

Класс Random

Класс **Random** используется для получения последовательности псевдослучайных чисел. В качестве "зерна" применяется 48-битовое число. Если для инициализации **Random** задействовать одно и то же число, будет получена та же самая последовательность псевдослучайных чисел.

В классе **Random** определено также несколько методов, которые возвращают псевдослучайные величины для примитивных типов **Java**.

Дополнительно следует отметить наличие двух методов: **double nextGaussian**() – возвращает случайное число в диапазоне от 0.0 до 1.0 распределенное по нормальному закону, и **void nextBytes**(byte[] arr) – заполняет массив arr случайными величинами типа byte.

Пример использования **Random**:

```
public class Test {
   public Test() {
   }
   public static void main(String[] args) {
      Test test = new Test();
      Random r = new Random(100);
      // Generating the same sequence numbers
      for(int cnt=0;cnt<9;cnt++){
         System.out.print(r.nextInt() + " ");
      }
      System.out.println();
      r = new Random(100);
      for(int cnt=0;cnt<9;cnt++) {
```

```
            System.out.print(r.nextInt() + " ");
         }
         System.out.println();
         // Generating sequence of bytes
         byte[] randArray = new byte[8];
         r.nextBytes(randArray);
         test.dumpArray(randArray);
      }
      void dumpArray(byte[] arr) {
         for(int cnt=0;cnt< arr.length;cnt++) {
            System.out.print(arr[cnt]);
         }
         System.out.println();
      }
}
```

Пример 24.

Результатом будет:

```
-1193959466 -1139614796 837415749 -1220615319 -1429538713 118249332 -
951589224 -1193959466
-1139614796 837415749 -1220615319 -1429538713 118249332 -951589224 81;-6;-
107;77;118;17;93;
-98;
```

Локализация

Класс Locale

Класс **Locale** предназначен для отображения определенного региона. Под регионом принято понимать не только географическое положение, но также языковую и культурную среду. Например, помимо того, что указывается страна Швейцария, можно указать также и язык – французский или немецкий.

Определено два варианта конструкторов в классе **Locale**:

```
Locale(String language, String country)
Locale(String language, String country,
       String variant)
```

Первые два параметра в обоих конструкторах определяют язык и страну, для которой определяется локаль, согласно кодировке **ISO**. Список поддерживаемых стран и языков можно получить и с помощью вызова статических методов **Locale**.**getISOLanguages**() **Locale.getISOCountries**(), соответственно. Во втором варианте конструктора указан также строковый параметр variant, в котором кодируется информация о платформе. Если здесь необходимо указать дополнительные параметры, то их требуется разделить символом подчеркивания, причем, более важный параметр должен следовать первым.

Пример использования:

```
Locale l = new Locale("ru","RU");
Locale l = new Locale("en","US","WINDOWS");
```

Статический метод **getDefault**() возвращает текущую локаль, сконструированную на основе настроек операционной системы, под управлением которой функционирует JVM.

Для наиболее часто использующихся локалей заданы константы. Например, **Locale**.US или **Locale**.GERMAN.

После того как экземпляр класса **Locale** создан, с помощью различных методов можно получить дополнительную информацию о локали.

```
public class Test {
   public Test() {
   }
   public static void main(String[] args) {
      Test test = new Test();
      Locale l = Locale.getDefault();
      System.out.println(l.getCountry() + " " +
         l.getDisplayCountry() + " " + l.getISO3Country());
      System.out.println(l.getLanguage() + " " +
         l.getDisplayLanguage() + " " + l.getISO3Language());
      System.out.println(l.getVariant() + " " + l.getDisplayVariant());
   l = new Locale("ru","RU","WINDOWS");
      System.out.println(l.getCountry() + " " +
         l.getDisplayCountry() + " " + l.getISO3Country());
      System.out.println(l.getLanguage() + " " +
         l.getDisplayLanguage() + " " + l.getISO3Language());
      System.out.println(l.getVariant() + " " + l.getDisplayVariant());
   }
}
```

Пример 26.

Результатом будет:

```
US United States USA
en English eng

RU Russia RUS
ru Russian rus
WINDOWS WINDOWS
```

Пример 27.

Класс ResourceBundle

Абстрактный класс **ResourceBundle** предназначен для хранения объектов, специфичных для локали. Например, когда необходимо получить набор строк, зависящих от локали, используют **ResourceBundle**.

Применение **ResourceBundle** настоятельно рекомендуется, если предполагается использовать программу в многоязыковой среде. С помощью этого класса легко манипулировать наборами ресурсов, зависящих от локалей, их можно менять, добавлять новые и т.д.

Набор ресурсов – это фактически набор классов, имеющих одно базовое имя. Далее наименование класса дополняется наименованием локали, с которой связывается этот класс. Например, если имя базового класса будет MyResources, то для английской локали имя класса будет MyResources_en, для русской – MyResources_ru. Помимо этого, может добавляться идентификатор языка, если для данного региона определено несколько языков. Например, MyResources_de_CH – так будет выглядеть швейцарский вариант немецкого языка. Кроме того, можно указать дополнительный признак variant (см. описание **Locale**). Так, описанный раннее пример для платформы UNIX будет выглядеть следующим образом: MyResources_de_CH_UNIX .

Загрузка объекта для нужной локали производится с помощью статического метода **getBundle**.:

```
ResourceBundle myResources =
   ResourceBundle.getBundle("MyResources",
                            someLocale);
```

На основе указанного базового имени (первый параметр), указанной локали (второй параметр) и локали по умолчанию (задается настройками ОС или JVM) генерируется список возможных имен ресурса. Причем, указанная локаль имеет более высокий приоритет, чем локаль по умолчанию. Если обозначить составляющие указанной локали (язык, страна, вариант) как 1, а локали по умолчанию – 2, то список примет следующий вид:

```
baseclass + "_" + language1 + "_" + country1 + "_" + variant1
baseclass + "_" + language1 + "_" + country1 + "_" + variant1 +
   ".Properties"
baseclass + "_" + language1 + "_" + country1
baseclass + "_" + language1 + "_" + country1 + ".Properties"
baseclass + "_" + language1
baseclass + "_" + language1 + ".Properties"
baseclass + "_" + language2 + "_" + country2 + "_" + variant2
baseclass + "_" + language2 + "_" + country2 + "_" + variant2 +
   ".Properties"
baseclass + "_" + language2 + "_" + country2
baseclass + "_" + language2 + "_" + country2 + ".Properties"
baseclass + "_" + language2
baseclass + "_" + language2 + ".Properties"
baseclass
baseclass + ".Properties"
```

Пример 28.

Например, если необходимо найти **ResourceBundle** для локали fr_CH (Швейцарский французский), а локаль по умолчанию en_US, при этом название базового класса **ResourceBundle** MyResources, то порядок поиска подходящего **ResourceBundle** будет таков.

```
MyResources_fr_CH
MyResources_fr
MyResources_en_US
MyResources_en
MyResources
```

Результатом работы **getBundle** будет загрузка необходимого класса ресурсов в память, однако данные этого класса могут быть сохранены на диске. Таким образом, если нужный класс не будет найден, то к требуемому имени класса будет добавлено расширение "**.Properties**" и будет предпринята попытка найти файл с данными на диске.

Следует помнить, что необходимо указывать полностью квалифицированное имя класса ресурсов, т.е. имя пакета, имя класса. Кроме того, класс ресурсов должен быть доступен в контексте его вызова (там, где вызывается getResourceBundle), то есть не быть private и т.д.

Всегда должен создаваться базовый класс без суффиксов, т.е. если вы создаете ресурсы с именем MyResource, должен быть в наличии класс MyResource.**class**.

ResourceBundle хранит объекты в виде пар ключ/значение. Как уже отмечалось ранее, класс **ResourceBundle** абстрактный, поэтому при его наследовании необходимо переопределить методы:

```
Enumeration getKeys()
protected Object handleGetObject(String key)
```

Первый метод должен возвращать список всех ключей, которые определены в **ResourceBundle**, второй должен возвращать объект, связанный с конкретным ключом.

Рассмотрим пример использования **ResourceBundle**:

```
public class MyResource extends ResourceBundle {

   private Hashtable res = null;
   public MyResource() {
      res = new Hashtable();
      res.put("TestKey","English Variant");
   }
   public Enumeration getKeys() {
      return res.keys();
   }
   protected Object handleGetObject(String key) throws
   Java.util.MissingResourceException {
      return res.get(key);
   }
}
public class MyResource_ru_RU extends ResourceBundle {
   private Hashtable res = null;
   public MyResource_ru_RU() {
      res = new Hashtable();
      res.put("TestKey","Русский варинат");
   }
   public Enumeration getKeys() {
      return res.keys();
   }
   protected Object handleGetObject(String key)
   throws Java.util.MissingResourceException {
      return res.get(key);
   }
}
public class Test {
   public Test() {
   }
   public static void main(String[] args) {
      Test test = new Test();
   ResourceBundle rb =
ResourceBundle.getBundle("experiment.MyResource",Locale.getDefault());
   System.out.println(rb.getString("TestKey"));
   rb = ResourceBundle.getBundle("experiment.MyResource", new
Locale("ru","RU"));
      System.out.println(rb.getString("TestKey"));
   }
}
```

Пример 29.

Результатом будет:

```
English Variant
Русский Вариант
```

Кроме того, следует обратить внимание, что **ResourceBundle** может хранить не только строковые значения. В нем можно хранить также двоичные данные, или просто методы, реализующие нужную функциональность, в зависимости от локали.

```
public interface Behavior {
   public String getBehavior();
   public String getCapital();
   }
public class EnglishBehavior implements Behavior{
   public EnglishBehavior() {
   }
   public String getBehavior(){
      return "English behavior";
   }
   public String getCapital(){
      return "London";
   }
}
public class RussianBehavior implements Behavior {
   public RussianBehavior() {
   }
   public String getBehavior(){
      return "Русский вариант поведения";
   }
   public String getCapital(){
      return "Москва";
   }
}
public class MyResourceBundle_ru_RU extends ResourceBundle {
   Hashtable bundle = null;
   public MyResourceBundle_ru_RU() {
      bundle = new Hashtable();
      bundle.put("Bundle description","Набор ресурсов для русской локали");
      bundle.put("Behavior",new RussianBehavior());
   }
   public Enumeration getKeys() {
      return bundle.keys();
   }
   protected Object handleGetObject(String key) throws
         Java.util.MissingResourceException {
      return bundle.get("key");
   }
}

public class MyResourceBundle_en_EN {
   Hashtable bundle = null;
   public MyResourceBundle_en_EN() {
      bundle = new Hashtable();
      bundle.put("Bundle description","English resource set");
      bundle.put("Behavior",new EnglishBehavior());
   }
   public Enumeration getKeys() {
      return bundle.keys();
   }
   protected Object handleGetObject(String key) throws
         Java.util.MissingResourceException {
      return bundle.get("key");
   }
}
public class MyResourceBundle extends ResourceBundle {
   Hashtable bundle = null;
   public MyResourceBundle() {
      bundle = new Hashtable();
```

```
      bundle.put("Bundle description","Default resource bundle");
      bundle.put("Behavior",new EnglishBehavior());
   }
   public Enumeration getKeys() {
      return bundle.keys();
   }
   protected Object handleGetObject(String key) throws
         Java.util.MissingResourceException {
      return bundle.get(key);
   }
}
public class Using {
   public Using() {
   }
   public static void main(String[] args) {
      Using u = new Using();
      ResourceBundle rb =
      ResourceBundle.getBundle("lecture.MyResourceBundle",
Locale.getDefault());
      System.out.println((String)rb.getObject("Bundle description"));
      Behavior be = (Behavior)rb.getObject("Behavior");
      System.out.println(be.getBehavior());
      System.out.println(be.getCapital());
      rb = ResourceBundle.getBundle("lecture.MyResourceBundle", new
Locale("en","EN"));
      System.out.println((String)rb.getObject("Bundle description"));
      Behavior be = (Behavior)rb.getObject("Behavior");
      System.out.println(be.getBehavior());
      System.out.println(be.getCapital());
   }
```

Пример 30.

Результатом будет:

```
Русский набор ресурсов
Русский вариант поведения
Москва
English resource bundle
English behavior
London
```

Классы ListResourceBundle и PropertiesResourceBundle

У класса **ResourceBundle** определено два прямых потомка **ListResourceBundle** и **PropertiesResourceBundle**. **PropertiesResourceBundle** хранит набор ресурсов в файле, который представляет собой набор строк.

Алгоритм конструирования объекта, содержащего набор ресурсов, был описан в предыдущем параграфе. Во всех случаях, когда в качестве последнего элемента используется .**Properties**, например, baseclass + "_" + language1 + "_" + country1 + ".**Properties**", речь идет о создании **ResourceBundle** из файла с наименованием baseclass + "_" + language1 + "_" + country1 и расширением **Properties**. Обычно класс **ResourceBundle** помещают в пакет resources, а файл свойств – в каталог resources. Тогда для того, чтобы инстанциировать нужный класс, необходимо указать полный путь к этому классу (файлу):

```
getBundle("resources.MyResource",
          Locale.getDefault());
```

ListResourceBundle хранит набор ресурсов в виде коллекции и является абстрактным классом. Классы, которые наследуют **ListResourceBundle**, должны обеспечить:

- переопределение метода **Object**[][] **getContents**(), который возвращает массив ресурсов;
- собственно двумерный массив, содержащий ресурсы.

Рассмотрим пример:

```
public class MyResource extends ListResourceBundle {
   Vector v = new Vector();
   Object[][] resources = {
   {"StringKey","String"},
   {"DoubleKey",new Double(0.0)},
   {"VectorKey",v},
   };
   public MyResource() {
      super();
      v.add("Element 1");
      v.add("Element 2");
      v.add("Element 3");
   }
   protected Object[][] getContents() {
      return resources;
   }
}
public class Test {
   public Test() {
   }
   public static void main(String[] args) {
      Test test = new Test();
      ResourceBundle rb =
ResourceBundle.getBundle("experiment.MyResource",Locale.getDefault());
      Vector v = (Vector)rb.getObject("VectorKey");
      Iterator it = v.Iterator();
      while(it.hasNext()) {
         System.out.println(it.next());
      }
   }
}
```

Пример 32.

Результатом будет:

```
Element 1
Element 2
Element 3
```

Создание ресурсов для локалей, отличных от локали по умолчанию, осуществляется так же, как было показано для **ResourceBundle**.

Заключение

Выше были рассмотрены вспомогательные классы пакета **Java.util.** Как можно было заметить, они относятся к самым разным задачам, а потому редкая программа обходится без использования хотя бы одного класса этого пакета.

Напомним кратко все основные классы и их особенности:

- Для работы с датой и временем должны использоваться классы **Date**, **Calendar**. Класс **Calendar** абстрактный, существует конкретная реализация этого класса **GregorianCalendar**.
- Интерфейс **Observer** и класс **Observable** реализуют парадигму MVC и предназначены для уведомления одного объекта об изменении состояния другого.
- Коллекции (**Collections**) не накладывают ограничений на порядок следования и дублирование элементов.
- Списки (**List**) поддерживают порядок элементов (управляются либо самими данными, либо внешними алгоритмами).
- Наборы (**Set**) не допускают дублированных элементов.
- Карты (**Maps**) используют уникальные ключи для поиска содержимого.
- Применение массивов делает добавление, удаление и увеличение количества элементов затруднительным.
- Использование связанных списков (**LinkedList**) обеспечивает хорошую производительность при вставке, удалении элементов, но снижает скорость индексированного доступа к ним.
- Использование деревьев (**Tree**) облегчает вставку, удаление и увеличение размера хранилища, снижает скорость индексированного доступа, но увеличивает скорость поиска.
- Применение хэширования облегчает вставку, удаление и увеличение размера хранилища, снижает скорость индексированного доступа, но увеличивает скорость поиска. Однако хэширование требует наличия уникальных ключей для запоминания элементов данных.
- Класс **Properties** удобен для хранения наборов параметров в виде пар ключ/значение. Параметры могут сохраняться в потоки (файлы) и загружаться из них.
- Реализация классом интерфейса **Comparator** позволяет сравнивать экземпляры класса друг с другом и, соответственно, сортировать их, например, в коллекциях.
- Arrays является классом-утилитой и обеспечивает набор методов, реализующих различные приемы работы с массивами. Не имеет конструктора.
- **StringTokenizer** – вспомогательный класс, предназначенный для разбора строк на лексемы.
- При необходимости работать с сущностями, представленными в виде битовых последовательностей, удобно использовать класс **BitSet**.
- Манипулировать ресурсами, которые различаются в зависимости от локализации, удобно с помощью классов **ResourceBundle**, **ListResourceBundle**, **PropertiesResourceBundle**. Собственно локаль задается с помощью класса **Locale**.