Обобщения

Обобщения - это параметризованные типы.

Параметризованные типы позволяют объявлять классы, интерфейсы и методы, где тип данных, которыми они оперируют, указан в виде параметра. Используя обобщения, можно создать единственный класс, например, который будет автоматически работать с разными типами данных. Классы, интерфейсы или методы, имеющие дело с параметризованными типами, называются обобщениями, обобщенными классами или обобщенными методами.

```
Gen<T> {                                  // T – параметр типа
        T ob;
        Gen(T o) { ob = o; }
        T getob () { return ob; }
        void showType( ) {          // строковое представление имени класса
          System.out.println("Типом T является" + ob.getClass().getName()); }
}

class GenDemo {
  public static void main(String args[]) {
       Gen<Integer>  iOb;
       iOb = new Gen<Integer>(88);        // нельзя   new Gen<Double>(88.0);
       iOb.showType();                    // Типом T является java.lang.Integer
       int v = iOb.getob();
       System.out.println("значение:"  + v);          // Значение: 88
       System.out.println();
      Gen<String>   strOb = new Gen<String> ("Обобщенный тест");
      strOb.showType();                    // Типом т является java.lang.String
      String str = strOb.getob();          // Значение: Обобщенный тест
      System.out.println(" Значение:"+ str);   }  }
```

Обобщения работают только с объектами.
Когда объявляется экземпляр обобщенного типа, аргумент, переданный в качестве параметра типа, должен быть типом класса. Нельзя использовать примитивные типы (int,char, …).

Нельзя! Gen<int> strOb = new Gen<int>(53) ;

Обобщенные типы отличаются в зависимости от типов-аргументов. Ссылка на одну специфическую версию обобщенного типа не совместима с другой версией того же обобщенного типа.

Нельзя! iOb = strOb;

Несмотря на то, что iOb и strOb имеют тип Gen<T>, они являются ссылками на разные типы, потому что типы их параметров отличаются.

Благодаря обобщениям, то, что было ошибками времени выполнения, становится ошибками времени компиляции.

```
class NonGen {
   Object ob;
   NonGen(Object o) { ob = o; }
   Object   getob () { return ob;  }
   void showType() {
        System.out.println("Типом ob является" + ob.getClass().getName()); }
  }
class NonGenDemo {
public static void main(String args[]) {
     NonGen iOb;
     iOb = new NonGen(88);
     iOb.showType();
     int v= (Integer) iOb.getob();
     System.out.println("значение:"+ v);
     System.out.println();
     NonGen strOb = new NonGen ("Тест без обобщений") ;
     strOb.showType();
     String str = (String) strOb.getob();
     System.out.println("Значение: "+ str);
     iOb = strOb;
     v = (Integer) iOb.getob();    // ошибка времени выполнения!
} }
```

Для обобщенного типа можно объявлять более одного типа параметра.

```
class TwoGen<T, V> {
      T ob1;
      V ob2;
  TwoGen(T o1, V o2) {
      ob1 = o1;
      ob2 = o2;  }
  void showTypes() {
    System.out.println("Тип T: " + ob1.getClass().getName() );
    System.out.println ("Тип V: " + ob2.getClass().getName() );  }
  T getob1 () { return ob1; }
  V getob2 () { return ob2; }  }

class SimpGen {
public static void main(String args[]) {
TwoGen<Integer, String>  tgObj = new TwoGen<Integer, String> (88,"Обобщения");
       tgObj.showTypes();
       int v = tgObj.getobl();
       System.out.println("Значение: "+ v);
       String str = tgObj.getob2();
       System.out.println("Значение: " + str);    }   }
```

Ограниченные типы

В предыдущих примерах параметры типов могли быть заменены любыми типами классов. Иногда удобно ограничить перечень типов, передаваемых в параметрах.

Если требуется создать обобщенный класс, который содержит метод, возвращающий среднее значение массива чисел. Предполагается использовать этот класс для получения среднего значения чисел, включая целые и числа с плавающей точкой .

```
class Stats<T> {
  T[]  nums;
  Stats(T[]  o){
        nums = o;  }
  double average() {
    double sum  = 0.0;
    for(int i=0; i < nums.length; i++)
        sum += nums[i].doubleValue();
            return sum/nums.length;  }
        }
```

Метод average() в Stats пытается получить double-версию каждого числа в массиве nums, вызывая doubleValue().

Поскольку все числовые классы, такие как Integer и Double, являются подклассами Number, и в классе Number определяет метод doubleValue(), этот метод доступен всем числовым классам-оболочкам.

Проблема в том, что компилятор не знает, что будут создаваться объекты Stats только с числовыми типами. При компиляции Stats, выдается сообщение об ошибке, говорящее о том, что метод doubleValue() не известен.

Чтобы решить эту проблему, нужно как-то сообщить компилятору, что в параметре T будут передаваться только числовые типы.

Можно создать ограничение сверху, которое объявляет суперкласс, от которого все типы-аргументы должны быть унаследованы. Это достигается применением слова extends при указании типа параметра:

<T extends Superclass>

Это означает, что T может быть заменено только классом Superclass, либо его подклассами.

Можно использовать ограничение сверху, чтобы исправить класс Stats, задав верхнюю границу используемого параметра типа Number:

```
class Stats<T extends Number> {
  T[ ] nums;
  Stats (T[ ] o) { nums = o;  }
 double average() {
  double sum = 0.0;
        for(int i=0; i < nums.length; i++)
                sum += nums[i].doubleValue();
                        return sum / nums.length;  } }

class BoundsDemo {
       public static void main(String args[ ]) {
          Integer inums[] = { 1, 2, 3, 4, 5 };
          Stats<Integer> iob = new Stats<Integer>(inums);
         double v = iob.average();
         System.out.println("Среднее значение iob равно" + v);
         String strs[] = { "1", "2", "3", "4", "5" };
         Stats<String> strob  = new Stats<String>(strs);
         double x = strob.average();
         System.out.println("среднее значение strob равно" + v); } }
```

Поскольку тип T теперь ограничен классом Number, компилятор java знает, что все объекты типа T могут вызывать doubleValue(), так как это метод класса Number. Поскольку String не является подклассом Number, программу невозможно будет скомпилировать.

Как ограничения можно также применять тип интерфейса. Фактически можно специфицировать в качестве ограничений множество интерфейсов. Такое ограничение может включать как тип класса, так и один или более интерфейсов. В этом случае тип класса должен быть задан первым.

Когда ограничение включает тип интерфейса, допустимы только типы-аргументы, реализующие этот интерфейс. Указывая ограничение, имеющее класс и интерфейс либо множество интерфейсов, нужно применять операцию **&** для их объединения:

class Gen<T extends MyClass & MyInterface> { ... }

Тип T ограничен классом MyClass и интерфейсом MyInteface. То есть любой тип, переданный в T, должен быть подклассом MyClass и иметь реализацию MyInteface.

Аргументы-шаблоны

Пусть в класс Stats требуется добавить метод по имени sameAvg (), который определяет, содержат ли два объекта Stats массивы, порождающие одинаковое среднее значение, независимо от того, какого типа числовые значения в них содержатся. Например, если один объект содержит double-значения 1.0, 2.0 и 3.0, а другой объект - целочисленные значения 2, 1 и 3, то среднее значение будет одинаково.

```
Integer inums[] = { 1, 2, 3, 4, 5 };
Double dnums[] = { 1.0, 2.0, 3.0, 4.0, 5.0 };

Stats<Integer>  iob = new Stats<Integer>(inums);
Stats<Double>  dob = new Stats<Double>(dnums);

if(iob.sameAvg(dob))
        System.out.println(“Средние значения одинаковы“ );
 else
       System.out.println(“Средние значения отличаются”);
```

```
boolean sameAvg(Stats<T>  ob) {
        if (average ()  ==  ob.average())
                        return true;
        return false;  }
```

Такой код будет работать только с другим объектом Stats, чей тип совпадает с вызывающим объектом. Эту функцию нельзя использовать для сравнения среднего значения Stats<Double> со средним значением Stats<Short>.

Для создания обобщенной версии метода sameAvg(), нужно использовать аргументы-шаблоны.

Аргумент-шаблон специфицируется символом **?** и представляет собой неизвестный тип.

```
boolean sameAvg(Stats<?>  ob) {
        if (average ()  ==  ob.average())
                        return true;
        return false;  }
```

Stats<?> ob соответствует любому объекту Stats.

Аргументы-шаблоны могут быть ограничены почти таким же способом, как параметры типов. Ограниченный шаблон важен, когда создается обобщенный тип, оперирующий иерархией классов.

```
class TwoD {
    int x, y;
    TwoD(int a, int b) {  x = a;  y= b;  } }

 class ThreeD extends TwoD{
         int z;
         ThreeD(int a, int b, int c) {    super (a,b);   z = c;   } }

 class FourD extends ThreeD {
             int t;
            FourD(int a, int b, int c, int d) { super (a, b, c);   t  = d;  } }

class Coords<T extends TwoD> {
       T [ ]  coords;
        Coords(T[ ] o) { coords = o; } }
```

Обратите внимание, что Coords определяет тип параметра, ограниченный TwoD. Это значит, что любой массив, сохраненный в объекте Coords, будет содержать объект типа TwoD или любой из его подклассов.

Теперь предположим, что вы хотите написать метод, который отображает координаты x и y для каждого элемента в массиве coords объекта Coords. Поскольку все типы объектов Coords имеют, как минимум, пару координат (X и У), это легко сделать с помощью шаблона:

```
static void showXY(Coords<?>  c) {
   System.out.println(“ X У Coordinates:“ );
    for(int i=O; i < c.coords.length; i++)
        System.out.println(c.coords[i].x + “  “  + c.coords[i] .y);  }
```

Поскольку Coords - ограниченный обобщенный тип, который определяет TwoD как верхнюю границу, все объекты, которые можно использовать для создания объекта Coords, будут массивами типа TwoD или классов, наследуемых от TwoD. Таким образом, showXY () может отображать содержимое любого объекта Coords.

Если требуется метод, отображающий координаты X,У и Z объекта ThreeD или FourD? Не все объекты Coords будут иметь три координаты. Требуется написать метод, который будет отображать координаты X,У и Z для Coords<ThreeD> и Coords<FourD>, в то же время предотвращая использование этого метода с объектами Coords<TwoD>.

Ограниченные шаблонные аргументы

Ограниченный шаблон определяет верхнюю или нижнюю границу типа аргумента. Это позволяет ограничить типы объектов, которыми будет оперировать метод. Более часто употребляемый ограниченный шаблон , это ограничивающий сверху, который создается применением оператора extends, почти таким же способом, как это делается при описании ограниченного типа.

```
static void showXYZ(Coords<? extends ThreeD> c) {
  System.out.println("X У Z Coordinates:");
  for(int i=0; i < c.coords.length; i++)
    System.out.println(c.coords[i].x + " " + c.coords[i].y + "  " + c. coords[i].z); }
```

? должен соответствовать типу ThreeD или унаследованным от него. Из-за этого ограничения showXYZ() может быть вызван со ссылкой на объекты типа Coords<ThreeD> или Coords<FourD>, но не со ссылкой на Coords<TwoD>. Попытка вызвать showXYZ () со ссылкой на Coords<TwoD> вызывает ошибку при компиляции.

Можно объявить обобщенный метод, который использует один или более параметров типов и включен в необобщенный класс.

```
class GenMethDemo {
static <T, V extends T> boolean isln (T x, V[ ] y) {
     for(int i=0; i < y.length; i++) {
                if(x.equals(y[i]))    return true; }
     return false; }

public static void main(String args[ ]) {
  Integer nums[] = { 1, 2, 3, 4, 5 };
  if( isln(2, nums) )     System.out.println(“ 2 содержится в nums“);
  if(! isln(7, nums) )    System.out.println(“ 7 не содержится в nums“);

String strs[] = {“один“, “два“, “три“, “четыре“, “пять“};
 if(isln(“два“, strs))       System.out.println(“два содержится в strs“);
 if(!isln(“семь“, strs))   System.out.println(“семь содержится в strs“);

// Не скомпилируется!
// if(isln("два", nums))   System.out.println ("два содержится в strs");
```

1. Параметр типа объявлен перед типом возврата метода.
2. Тип V ограничен сверху типом T (V либо должен быть тем же типом, что и T, либо типом его подклассов). Это отношение указывает, что isln() может быть вызван только с аргументами, совместимыми между собой.
3. При вызове нет необходимости определять аргументы типа (типы T и V определяются соответственно в вызове if (isln (2, nums)), тип T будет Integer (благодаря автоупаковке).

Рассмотрим:

if(isln("два", nums)) System.out.println ("два содержится в strs");

Тип параметра V ограничен типом T . Это значит, что V должно иметь либо тип T, либо тип его подкласса. Первый аргумент имеет тип String, а второй - Integer, который не является подклассом String. Это вызовет ошибку несоответствия типов во время компиляции.

Конструкторы также могут быть обобщенными, даже если их классы таковыми не являются.

```
class GenCons {
  private double val;
<T extends Number> GenCons(T   arg) {
                    val = arg.doubleValue();   }
void showval() {
      System.out.println(“ val: “ + val);  }
}

class GenConsDemo {
      public static void main(String args[]) {
          GenCons test = new GenCons(100);
          GenCons test2 = new GenCons(123.5F);
          test.showval();
          test2.showval();    }    }
```

Поскольку GenCons () определяет параметр обобщенногоо типа, который может быть подклассом Number, GenCons () можно вызывать с любым числовым типом, включая Integer, Float или Double.

```
class Gen<T extends Number> {
        T ob;
        T vals[];
        Gen(T o, T[ ] nums) {
                        ob = o;
    //                  vals = new T[10];        // НЕЛЬЗЯ!
                        vals = nums;              // МОЖНО
                         }
                }
class GenArrays {
 public static void main(String args[]) {
    Integer  n[] = { 1, 2, 3, 4, 5 };
    Gen<Integer> iOb = new Gen<Integer> (50, n);
 //  Gen<Integer> gens[] = new Gen<Integer>[10]; // НЕЛЬЗЯ
    Gen<?> gens[] = new Gen<?>[10];  // МОЖНО!
  }
}
```

Ограничения на использование обобщений

1. Нельзя создавать экземпляр типа параметра (T не существует во время выполнения).
2. Нельзя объявлять статические члены класса типа параметра T, статический метод не может возвращать значение типа параметра T, статический метод не может иметь доступ к объекту типа параметра T (можно только объявлять обобщенные static методы, определяющие их собственные параметры типа T).
3. Нельзя создавать экземпляр массива, базовый тип которого - это параметр типа (объявлять можно T nums[], T vals[] нельзя! nums = new T[10]; но можно присвоить ссылку на существующий массив nums = vals).
4. Нельзя создать массив специфичных для типа обобщенных ссылок (Нельзя! Gen<Integer> gens[] = new Gen<Integer>[10]; такие массивы могут нарушить безопасность типов).
   Можно: Gen<?> gens[] = new Gen<?>[10];

java.util: каркас коллекций

Каркас коллекций - это сложная иерархия интерфейсов и классов, предоставляющих технологию управления группами объектов.

Каркас коллекций был разработан для достижения нескольких целей:

1. быть высокопроизводительным (реализация основных коллекций (динамических массивов, связных списков, деревьев и хеш-таблиц) отличается высокой эффективностью).
2. позволить разным типам коллекций работать в единой манере и с высокой степенью взаимодействия.
3. расширение и/или адаптация коллекций должна быть простой.
4. единый набор стандартных интерфейсов для построения всего каркаса коллекций.

Все коллекции теперь обобщенные, и многие из методов, оперирующих коллекциями, также принимают обобщенные параметры.

Все классы коллекций модифицированы таким образом, что реализуют интерфейс Iterable. Это означает, что можно пройти циклом по коллекции, используя стиль «for-each» цикла for. Раньше для прохода по коллекциям необходимо было использовать итератор Iterator, программно конструируя цикл. Хотя итераторы все еще применяются для некоторых целей, во многих случаях циклы на основе итераторов могут быть заменены циклами for.

Интерфейсы коллекций

Каркас коллекций определяет несколько интерфейсов.

Интерфейсы, реализуемые коллекциями:

1. Интерфейс **Collection** - позволяет работать с группами объектов. Это вершина иерархии коллекций, он должен быть реализован всеми классами коллекций.

Collection - обобщенный интерфейс: **interface Collection<E>,**
E - указывает тип объектов коллекции.

Collection расширяет интерфейс Iterable. Это означает, что по всем коллекциям можно проходить циклами вида «for-each».

Методы интерфейса Collection (имеют все коллекции):

1.boolean add(E obj) - добавляет obj к вызывающей коллекции. (true, если obj был добавлен к коллекции, false – иначе).

2. boolean addAll(Collection<? extends E> c) - добавляет все элементы коллекции **c** к вызывающей коллекции (true, если все элементы добавлены, иначе - false).

3. void clear () - удаляет все элементы вызывающей коллекции.

4. boolean contains(Object obj) - возвращает true, если obj является элементом вызывающей коллекции, иначе - false.

5. boolean containsAll(Collection<?> c) - возвращает true, если вызывающая коллекция содержит все элементы **c**, иначе - false.

6. boolean equals(Object obj) - true, если вызывающая коллекция и obj эквивалентны, иначе - false.

7. **Iterator<E> iterator()** - возвращает итератор для вызывающей коллекции.

8. boolean remove(Object obj) - удаляет один экземпляр obj из вызывающей коллекции (true, если элемент удален, иначе – false).

9. boolean removeAll(Collection<?> c) - удаляет все элементы **c** из вызывающей коллекции (true, если элементы удалены, иначе – false).

10. int size () - возвращает количество элементов, содержащихся в коллекции.

11. Object[ ] toArray() - возвращает массив, содержащий все элементы вызывающей коллекции. Элементы массива являются копиями элементов коллекции.

2. Интерфейс **List -** расширяет Collection и определяет поведение коллекций, которое сохраняет последовательность элементов. Элементы могут быть вставлены или извлечены по их позиции в списке, используя начинающийся с нуля индекс. Список может содержать повторяющиеся элементы.
List - обобщенный интерфейс: **interface List<E>,** E - тип объектов, которые должен содержать список.

В дополнение к методам, объявленным в Collection, List определяет некоторые методы:

1. void add(int index, E obj) - вставляет obj в вызывающий список в позицию index. Любые ранее вставленные элементы за указанной позицией вставки смещаются вверх.

2. boolean addAll(int index, Collection<? extends E> c) - вставляет все элементы **c** в вызывающий список, начиная с позиции, переданной в index. Все ранее существовавшие элементы за точкой вставки смещаются вверх (true, если вызывающий список изменяется, false –иначе).

3. E get (int index) - возвращает объект, сохраненный в указанной позиции вызывающего списка.

3. Интерфейс **Set** - определяет множество (набор). Определяет поведение коллекций, не допускающих дублирования элементов.

4. Интерфейс **SortedSet** - расширяет Set и объявляет поведение множеств, сортированных в порядке возрастания.

5. Интерфейс **Queue -** расширяет Collection и объявляет поведение очередей, которые представляют собой список с дисциплиной «первый вошел - первый вышел».

6. Интерфейс **Dequeue** - расширяет Queue и описывает поведение двунаправленной очереди. Двунаправленная очередь может функционировать как стандартная очередь "первый вошел - первый вышел", либо как стек "последний вошел первый вышел".

Классы коллекций

1. Класс **ArrayList** реализует интерфейс List.
ArrayList - обобщенный класс: **class ArrayList<E>,** E - тип сохраняемых объектов.

ArrayList поддерживает динамические массивы, которые могут расти по мере необходимости. По сути, ArrayList - это массив переменной длины, содержащий объектные ссылки. То есть ArrayList может динамически увеличиваться или уменьшаться в размере.

Списки-массивы создаются с некоторым начальным размером. Когда этот первоначальный размер становится недостаточным, коллекция автоматически увеличивается. Когда объекты удаляются, коллекция может уменьшаться.

Конструкторы:
ArrayList() - создает пустой массив-список.
ArrayList(Collection <? extends E> c) - строит массив-список, который инициализируется элементами коллекции с.
ArrayList(int capacity) - создает массив-список, который имеет начальный размер capacity. Размер растет автоматически по мере добавления элементов в массив□список.

```
import java.util.*;
class ArrayListDemo {
public static void main(String args[]) {
   ArrayList<String>  al = new ArrayList<String>( );
   System.out.println(“Начальный размер al:“ + al.size());
   al.add("C");
   al.add("A");
   al.add("E") ;
   al.add("B") ;
   al.add("D");
   al.add("F");
   al.add(1, "A2");      // добавить по индексу 1 A2
 System.out.println ("Размер al после вставки: " + al.size( ) ) ;
 System.out.println("Содержимое al: " + al);  //весь список
  al.remove("F") ;    //удалить  F
  al.remove(2);          // удалить по индексу 2
System.out.println("Размер al после удалений: "+ al.size());
System.out.println ("Содержимое al: " + al);
  }   }
```

Чтобы увеличить размер вручную, метод
void ensureCapacity(int cap), cap – новый размер коллекции.

Чтобы уменьшить размер массива объектов до текущего реального количества хранимых объектов, метод **void trimToSize().**

Чтобы получить обыкновенный массив, содержащий все элементы списка, методы **Object[ ] toArray()** или **<T>T[ ] toArray(T array[ ]).**

```
ArrayList<Integer> al = new ArrayList<Integer>();
 al.add(l);
 al.add(2);
 al.add(3);
 al.add(4);
 Integer ia[ ] = new Integer[ al.size() ];
 ia = al.toArray( ia );
```

2. Класс **LinkedList** - реализует интерфейсы List, Dequeue и Queue. Он представляет структуру данных связного списка.
LinkedList - обобщенный класс: **class LinkedList<E>,** E - тип сохраняемых в списке объектов.
Конструкторы:
LinkedList() - создает пустой связный список.
LinkedList(Collection<? extends E> c) - строит связный список и инициирует его содержимым коллекции **c**.

```
LinkedList<String> LL = new LinkedList<String>( ) ;
  LL.add("F");
  LL.add("B.") ;
  LL.add("D") ;
  LL.add(1,“A");            // добавить на позицию 1
  LL.remove ("F") ;
  LL.removeFirst();         // удалить первый
  LL.removeLast();          //  удалить последний
String val = LL.get(2);     // получить значение
  LL.set(2, val + "Изменен");   // установить значение
```

3. Класс TreeSet - создает коллекцию, которая для хранения элементов применяет дерево. Объекты сохраняются в отсортированном порядке по возрастанию. **class TreeSet<E>,** E - тип объектов.
Конструкторы:
TreeSet () - пустой набор-дерево, который будет сортировать элементы в естественном порядке возрастания.
TreeSet(Collection<? extends E> c) - строит набор-дерево, содержащий элементы коллекции с.
TreeSet(SortedSet<E> ss) - строит набор-дерево, содержащий элементы из **ss**.

```
TreeSet<String> ts = new TreeSet<String>( );
  ts.add("C");
  ts.add("A") ;
  ts.add("B") ;
  ts.add("E") ;
  ts.add("F") ;
  ts.add("D") ;
  System.out.println( ts );
```

Применение цикла «for-each» для прохода по элементам коллекции

```
import java.util.*;
class ForEachDemo {
public static void main(String args[ ]) {
ArrayList<Integer>  vals = new ArrayList<Integer>( );
vals.add(1);
vals.add(2);
vals.add(3);
vals.add(4);
vals.add(5);
System.out.print("Исходное содержимое vals: ");
for(int v : vals)
   System.out.print(v + “  “ );    // System.out.print(vals);
System.out.println(); } }
```

Класс Date - инкапсулирует текущую дату и время.
Конструкторы:
Date() - инициализирует объект текущей датой и временем
Date(long millisec) – имеет аргумент, который представляет собой количество миллисекунд, прошедших с полуночи 1 января 1970.

```
import java.util.Date;
class DateDemo {
public static void main(String args[]){
    Date date = new Date();
    System.out.println(date);
    long msec = date.getTime();
    System.out.println ("Миллисекунд, прошедших с 1 января 1970 г. по
                                                    GMT = " + msec) ;
 }}
```

Класс Calendar (java.util)

Абстрактный класс Calendar представляет набор методов, позволяющих работать с датой и временем (год, месяц, день, часы, минуты и секунды). Calendar не имеет общедоступных конструкторов.

Методы:

boolean after(Object calendarObj) - true, если вызывающий объект Calendar содержит более позднюю дату, чем calendarObj, иначе - false.

boolean before(Object calendarObj) - true, если вызывающий объект Calendar содержит более раннюю дату, чем calendarObj, иначе - false.

final void clear() - обнуляет все компоненты времени в вызывающем объекте.
final void clear (int which) - обнуляет компонент времени вызывающего объекта, указанный в which.

boolean equals(Object calendarObj) - true, если вызывающий объект Calendar содержит дату, эквивалентную calendarObj, иначе - false.

int get(int calendarField) - возвращает значение одного компонента вызывающего объекта. Компонент указывается в calendarField (Calendar.YEAR, Calendar.MONTH, Calendar.MINUTE и т.д.)

Calendar getInstance() - возвращает объект Calendar для локали и часового пояса по умолчанию.

static Calendar getInstance(TimeZone tz) - возвращает объект Calendar для локали по умолчанию и часовоrо пояса tz.

final Date getTime() - возвращает объект Date, содержащий время, эквивалентное вызывающему объекту.

void set (int which, int val) - устанавливает компонент даты или времени вызывающего объекта, указанный в which, равным значению val. which должен иметь значение одного из полей Calendar, например, Calendar.HOUR.

final void set (int year, int month, int dayOfMonth) - устанавливает в вызывающем объекте различные компоненты даты и времени.

final void setTime (Date d) - устанавливает в вызывающем объекте различные компоненты даты и времени из объекта d.

В Calendar определены следующие целые константы, которые используются, когда вы получаете или устанавливаете компоненты календаря: ALL STYLES APRIL AUGUST и т.п.
MONTH FRIDAY HOUR MILLISECOND MINUTE

GregorianCalendar - конкретная реализация Calendar, которая представляет обычный Григорианский календарь. Метод getInstance() класса Calendar обычно возвращает GregorianCalendar, инициализированный текущей датой и временем в локали и часовом поясе по умолчанию.
Конструкторы:

GregorianCalendar(int year, int month, int dayOfMonth)

GregorianCalendar(int year, int month, int dayOfMonth, int hours, int minutes)

GregorianCalendar(int year, int month, int dayOfMonth, int hours, int minutes, int seconds)

Класс **TimeZone** позволяет работать с часовыми поясами, смещенными относительно Гринвича (GMT).

Класс **Locale** предназначен для создания объектов, каждый из которых описывает географический или культурный регион. Класс **Locale** определяет следующие константы: CANADA FRENCH CHINA CHINESE ENGLISH FRANCE GERMAN GERMANY и т.п.

```
import java.util.*;
class GregorianCalendarDemo {
  public static void main(String args[]) {

    String months[] = { “Янв “, “ Фев “, “ Мар “, “ Апр “, “ Май “, “ Июн “,
                        “ Июл “ , “ ABr “, “ Сен “, “ Окт “, “ Ноя “, “ Дек “};
     int year;

     GregorianCalendar gcalendar = new GregorianCalendar();

     System.out.print("дата: ");

     System.out.print( months[ gcalendar.get(Calendar.MONTH) ] );

     System.out.print(“ “ + gcalendar.get(Calendar.DATE) + “ “);

     System.out.println(year = gcalendar.get(Calendar.YEAR));

     System.out.print(“Время: “);
     System.out.print (gcalendar.get (Calendar.HOUR) + “ : “);
     System.out.print (gcalendar.get (Calendar.MINUTE) + “ : “);
     System.out.println(gcalendar.get(Calendar.SECOND));

 if(gcalendar.isLeapYear(year)) {
          System.out.println (“ Текущий год високосный “) ; }
 else {  System.out.println (“ Текущий год не високосный “) ;  }
} }
```

Timer и TimerTask

Полезным средством, предоставляемым пакетом java.util является возможность планировать запуск задания на определенное время в будущем. Классы, которые поддерживают эту возможность - **Timer** и **TimerTask**.

Используя эти классы, можно создать поток, выполняющийся в фоновом режиме и ожидающий заданное время. Когда время истечет, задача, связанная с этим потоком, будет запущена. Различные опции позволяют запланировать задачу на повторяющийся запуск либо на запуск по определенной дате. Timer и TimerTask работают вместе.

Timer - это класс, используемый для планирования выполнения задачи. Запланированная к выполнению задача должна быть экземпляром TimerTask. Чтобы запланировать задачу, нужно сначала создать объект TimerTask, а затем запланировать его запуск с помощью экземпляра Timer.
Конструкторы класса Timer:
Timer () - создает объект Timer и затем запускает его как обычный поток.
Timer(boolean DThread) – поток-демон, если параметр Dthread равен true. Поток-демон будет выполняться только до тех пор, пока выполняется остальная часть программы.
Timer(String tNate) и **Timer(String tNate, boolean DThread) –** потоки с именем.

TimerTask реализует интерфейс Runnable, поэтому он может быть использован для создания потока выполнения. Конструктор : TimerTask(). Метод run() должен быть переопределен.

Простейший способ создать задачу для таймера - это расширить TimerTask и переопределить run(). Как только задача создана, она планируется для выполнения объектом типа Timer.

```
import java.util.*;
class MyTimerTask extends TimerTask {
    public void run() {
        System.out.println ("Задание таймера выполняется") ; }  }

class Ttest {
  public static void main(String args[]) {
    MyTimerTask myTask = new MyTimerTask();
    Timer  myTimer = new Timer();
    myTimer.schedule(myTask, 1000, 500);
       try{
          Thread.sleep(5000);   }
      catch (InterruptedException exc) {   }
    myTimer.cancel();  } }
```