Лекция 5

Шаблоны в Java

В Java можно использовать шаблоны классов.

Рассмотрим пример класса - шаблона:

```
public class Pair<T, S> {
    public Pair(T f, S s) { first = f; second = s; }
    public T getFirst() { return first; }
    public S getSecond() { return second; }
    public String toString() {
      return "(" + first.toString() + ", " +
      second.toString() + ")"; }
   private T first;
   private S second; }
```

Использование данного класса имеет вид:

```
……….
Pair<String, String> grade440 = new Pair <String,
                                          String>("mike", "A");
Pair<String, Integer> marks440 = new Pair <String,
                                          Integer>("mike", 100);
System.out.println("grade:" + grade440.toString());
System.out.println("marks:" + marks440.toString());
```

……..

Однако стоит заметить, что конкретизация шаблона примитивным типом в Java невозможна. Т.е.

```
Pair<int, int> grade440 = new Pair <int, int>(5, 10); //error
```

Знак вопроса

В определении шаблона или при генерации можно использовать знак вопроса.

- **?** - указывает на любой класс, сама специфика класса не важна;
- **? extends T** - определяет множество классов потомков от T;
- **? super T** - определяет множество родительских классов класса T.

Например, если нужен метод вывода списка фигур, потомков абстрактного класса Shape, то определение метода будет выглядеть следующим образом.

```
public void draw(List<? extends Shape> shape) { ... }
```

Перечисления

Перечисления являются видом классов в Java, позволяющих задать набор значений, которые могут принимать объекты этих классов.

Значения задаются идентификаторами, каждому из которых сопоставляется целое число и строка, с именем идентификатора.

Для каждого перечисления определено два статических метода:

- **values()** - возвращает массив всех значений перечисления;
- **valueOf(String name)** - возвращает константу перечисления соответствующую указанной строке.

Пример:

```
public enum EnumTest {
                    RED, GREEN, BLUE, WHITE, BLACK, GRAY, YELLOW;
    boolean isMonochorome() {
                     return this==BLACK || this==WHITE ||   this==GRAY;
    }
}
public class Main {
   public static void main(String[] args) {
     try{
       for (EnumTest k: EnumTest.values())
        System.out.println(k);
        System.out.println(EnumTest.valueOf("RED")); //На зкране  RED
        System.out.println(EnumTest.valueOf(“wererw"));
                                                          //IllegalArgumentException
      }
      catch(IllegalArgumentException e){…….}
    }
}
```

Класс Enum

Внутренне все перечисления наследуются от класса **Enum**, для которого определены следующие методы:

- **compareTo(E o)** - метод сравнения с объектом (интерфейс Comparable). Возвращает отрицательное, ноль или положительное целое значение, если объект меньше, равен или больше указанного объекта;
- **int ordinal()** - возвращает целочисленную константу, связанную с этим объектом;
- **toString()** - возвращает имя строковой константы, связанной с этим объектом.

Перечисления встраиваются в оператор выбора switch.
Пример:

```
public class Test {
        public static void main(String[] args) {
          EnumTest e1 = EnumTest.BLACK;
          System.out.println(e1);
          System.out.println(e1.isMonochorome());
          System.out.println("==============");
          EnumTest e = EnumTest.GREEN;
          switch (e) {
            case RED: // хотя можно и EnumTest.RED
                  System.out.println("1 - " + e); break;
            case GREEN:
                   System.out.println("2 - " + e); break;
            case BLUE:
                  System.out.println("3 - " + e); break;
            default:
                  System.out.println("non rgb color");
          }
          System.out.println(e.compareTo(e1)); //На экране -3
          System.out.println(e.ordinal()); //На экране 1
        }
}
```

Назначение поведения.

Каждому значению перечисления можно сопоставить свое поведение.

Пример:

```
import java.util.*;
public enum Operation {
  PLUS {
        double eval(double x, double y) { return x + y; }
   },
  MINUS {
        double eval(double x, double y) { return x - y; }
   },
  TIMES {
        double eval(double x, double y) { return x * y; }
  },
  DIVIDED_BY {
       double eval(double x, double y) { return x / y; }
  };
  abstract double eval(double x, double y);
  public static void main(String args[ ]) {
       double x = 2.3;
       double y = 4.3;
       for (Operation op : Operation.values())
            System.out.println(x + " " + op + " " + y + " = " + op.eval(x, y));
   }
}
```

Потоки в Java

Потоки в Java позволяют распараллелить выполнение программы. Поток может работать независимо от других потоков.

Создание потоков

Потоки представлены классом в стандартных библиотеках Java. Чтобы создать новый поток выполнения, необходимо создать объект Thread:

```
Thread worker = new Thread();
```

После того как объект-поток будет создан, можное задать его конфигурацию и запустить.

В понятие конфигурации потока входит указание исходного приоритета, имени и т.д.

Когда поток готов к работе, следует вызвать его метод **start**.

Метод **start** порождает новый выполняемый поток на основе данных объекта класса Thread, после чего завершается.

Метод **start** также вызывает метод **run** нового потока, что приводит к активизации последнего.

Выход из метода **run** означает прекращение работы потока.

Поток можно завершить и явно, посредством вызова **stop**; его выполнение может быть приостановлено методом **suspend.**

Стандартная реализация Thread.run не делает ничего.

Поэтому необходимо либо расширить класс Thread, чтобы включить в него новый метод run, либо создать объект Runnable и передать его конструктору потока.

Прежде всего рассмотрим расширение класса Thread.

Рассмотрим приложение, создающее два потока, которые выводят слова “ping” и “PONG” с различной частотой.

```
class PingPong extends Thread {
    String word;
    int delay; // длительность паузы
 PingPong(String whatToSay, int delayTime) {
   word = whatToSay;
   delay = delayTime; }
 public void run() {
    try {
      for (;;) {
          System.out.print(word + " ");
           sleep(delay);
       }
    }
    catch (InterruptedException e) { return; }
  }
 public static void main(String[] args) {
   new PingPong("ping", 33).start(); // 1/30 секунды
   new PingPong("PONG", 100).start(); // 1/10 секунды
  }
}
```

В данном приложении метод run работает в бесконечном цикле, выводя содержимое поля word и делая паузу на delay микросекунд.

Метод PingPong.run не может возбуждать исключений, поскольку этого не делает переопределяемый им метод Thread.run.

Соответственно, необходимо перехватить исключение InterruptedException, которое может возбуждаться методом sleep.

Рассмотрим другой метод создания потока – создание объекта Runnable и передачи его конструктору потока.

Использование объектов Runnable

Поток служит абстракцией понятия исполнителя — субъекта, способного к выполнению каких-либо полезных действий.

План работы, подлежащей выполнению, описывается посредством инструкций метода run.

Чтобы некая цель была достигнута, необходимы исполнитель и план работы: интерфейс Runnable абстрагирует понятие работы и позволяет назначить ее исполнителю — объекту потока.

В составе интерфейса Runnable объявлен единственный метод:

```
 interface Runnable{
  public void  run();
}
```

Класс Thread реализует интерфейс Runnable, поскольку поток сам по себе способен определять план работы, подлежащей выполнению.

Реализация интерфейса Runnable во многих случаях представляется более простым решением, чем расширение класса Thread.

Рассмотрим тот же самый пример, но здесь напишем реализацию интерфейса Runnable:

```
public class RunPingPong implements Runnable{
    private String word;
    private int delay;
    RunPingPong(string whatToSay, int delayTime){
       word = whatToSay;
       delay = delayTime;
    }
    public void  run() {
     try {
      for  (;;){
       System.out.print(word + " ");
       Thread.sleep(delay);
      }
     }
     catch (interruptedException e) { return;  }
    }
   public static void main(String[] args){
      Runnable ping = new RunPingPong(“ping”, 33);
      Runnable pong = new RunPingPong(“PONG”, 100);
      new Thread(ping).start();
      new Thread(pong).start();
  }
}
```

Существует четыре перегруженные версии конструктора класса Thread, позволяющие передать в качестве параметра объект Runnable:

1. **public Thread(Runnable target);**

создает новый объект Thread, использующий метод run объекта target

2. **public Thread(Runnable target, String name);**

создает новый объект Thread с заданным именем name, использующий метод run объекта target.

3. **public Thread(ThreadGroup group, Runnable target);**

создает новый объект Thread в указанном объекте ThreadGroup, использующий метод run объекта target.

4. **public Thread(ThreadGroup group,**
Runnable target, String name);

создает новый объект Thread с заданным именем name в указанном объекте ThreadGroup, использующий метод run объекта target.

Синхронизация

Когда два потока должны воспользоваться одним и тем же объектом, возникает опасность, что наложение операций приведет к разрушению данных.

Поэтому потоки должны синхронизировать свой доступ к критическим секциям.

Для этого в условиях многопоточности используется блокировка объекта.

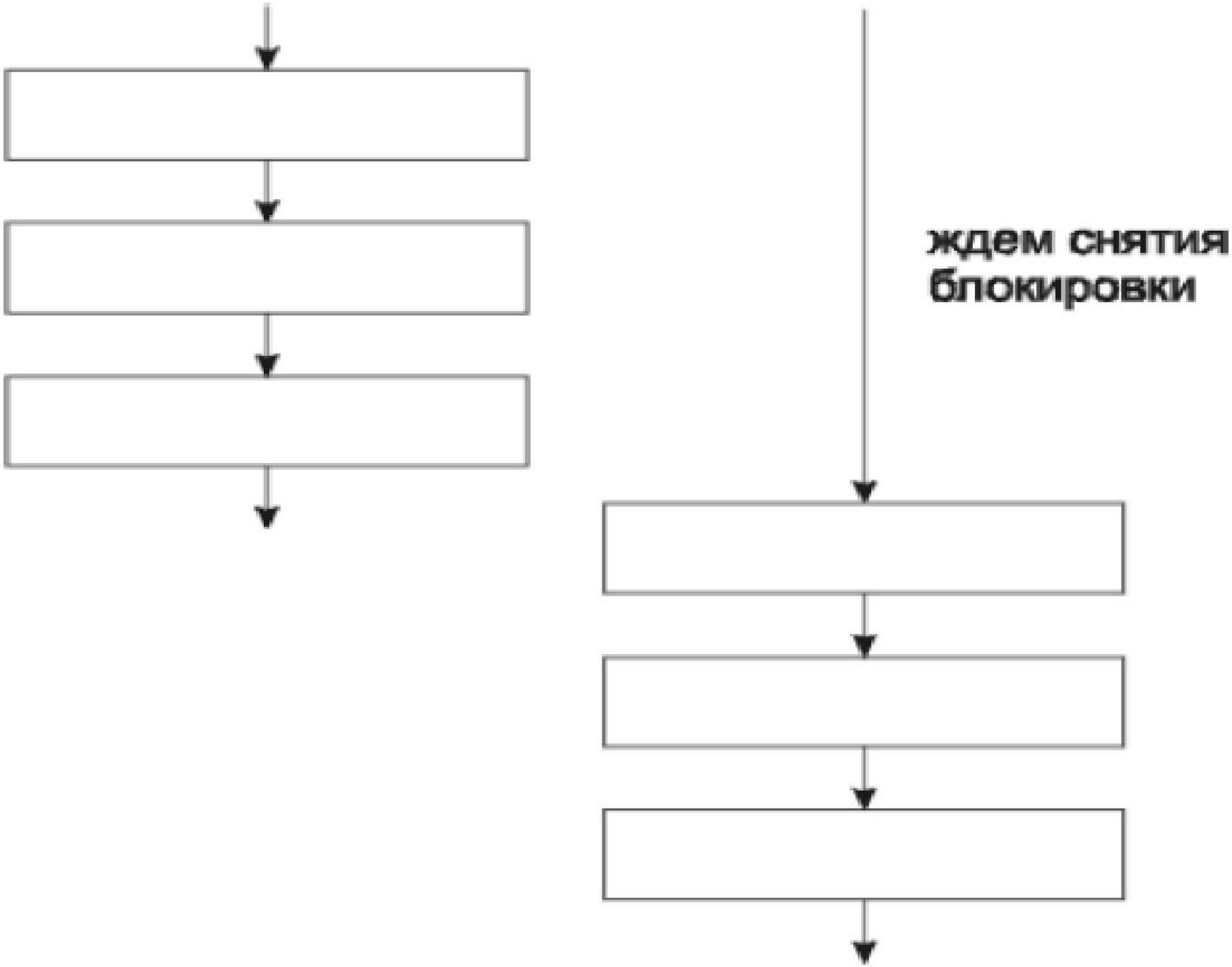
Когда объект заблокирован некоторым потоком, только этот поток может работать с ним.

Методы synchronized

Чтобы класс мог использоваться в многопоточной среде, необходимо объявить соответствующие методы с атрибутом **synchronized**.

Если некоторый поток вызывает метод synchronized, то происходит блокировка объекта.

Вызов метода synchronized того же объекта другим потоком будет приостановлен до снятия блокировки.


ждем снятия
блокировки

Синхронизация приводит к тому, что выполнение двух потоков становится взаимно исключающим по времени.

Рассмотрим пример:

```
class Account {
  private double balance; //данное поле защищеное от любых
    несинхронных действий
  public Account(double initialDeposit) {
     balance = initialDeposit;
  }
   public synchronized double getBalance() {
     return balance;
   }
   public synchronized void deposit(double amount) {
      balance += amount;
    }
}
```

Конструктор не обязан быть synchronized, поскольку он выполняется только при создании объекта, а это может происходить только в одном потоке для каждого вновь создаваемого объекта.

Два потока не могут одновременно выполнять синхронизированные статические методы одного класса.

Блокировка статического метода на уровне класса не отражается на объектах последнего — можно вызвать синхронизированный метод для объекта, пока другой поток заблокировал весь класс в синхронизированном статическом методе.

В последнем случае блокируются только синхронизированные статические методы.

Если синхронизированный метод переопределяется в расширенном классе, то новый метод не обязан быть синхронизированным.

Метод суперкласса при этом остается синхронизированным, так что несинхронность метода в расширенном классе не отменяет его синхронизированного поведения в суперклассе.

Если в несинхронизированном методе используется конструкция super.method() для обращения к методу суперкласса, то объект блокируется на время вызова до выхода из метода суперкласса.

Операторы synchronized

Оператор synchronized позволяет выполнить синхронизированный фрагмент программы, который осуществляет блокировку объекта, не требуя от программиста вызова синхронизированного метода для данного объекта.

Оператор synchronized состоит из двух частей: указания блокируемого объекта и оператора, выполняемого после получения блокировки.

Общая форма оператора synchronized выглядит следующим образом:

```
synchronized (выражение){
    операторы
}
```

Рассмотрим пример:

```
public static void abs(int[] values) {
   synchronized (values) { //доступ к массиву values
   блокируется со стороны других потоков
     for (int i = 0; i < values.length; i++) {
        if (values[i] < 0) values[i] = - values[i];
     }
    }
}
```

Операторы synchronized обладают преимуществами перед synchronized методами:

1. они дают возможность определения синхронизированного участка кода, охватывающего только некоторый фрагмент тела метода.
2. synchronized- операторы позволяют синхронизировать объекты, отличные от this, и дают возможность создавать самые разнообразные схемы синхронизации.

Одна из достаточно часто встречающихся ситуаций связана с необходимостью обеспечения более высокого уровня интенсивности конкурентного доступа к коду класса за счет уменьшения размеров блокируемых областей кода.

Рассмотрим пример:

```
class SeparateGroups {
   private double aval =0.0;
   private double bval = 1.1;
   protected Object lockA = new Object();
   protected Object lockB = new Object();

   public double getA() {
      synchronized (lockA) { return aval;}
   }
   public void setA(double val)  {
      synchronized  (lockA)  { aval  = val;}
   }
   public double getB(){
      synchronized  (lockB)  { return bval;}
   }
   public void setB(double val){
      synchronized  (lockB) {bval  = val; }
   }
```

```
public void reset(){
synchronized (lockA) {
   synchronized (lockB) {
          aval = bval =0.0;
    }
 }
}
}
```

Еще одна из часто возникающих ситуаций, в которых удобно использовать инструкции synchronized, связана с необходимостью синхронизации кода внешнего объекта при обращении к нему из внутреннего объекта:

```
public class Outer {
   private int data;
   private class Inner {
     void setOuterData(){
       synchronized (Outer.this){data = 12;}
      }
    }
 }
```

Для задания блокируемого объекта можно использовать литерал типа class для текущего класса.

Такой подход применим и в том случае, если надлежит предотвратить доступ к статическим данным со стороны нестатического кода.

Рассмотрим пример:

```
class Body {
   public long idNum;
   public String nameFor;
   public Body orbits;
   public static long nextID = 0;
   Body(){
    synchronize(Body.class){
       idNum=nextID++;
    }
  }
}
```

Методы wait, notifyAll и notify

Механизм блокировки решает проблему с возможным влиянием нескольких потоков, однако хотелось бы, чтобы потоки могли обмениваться информацией друг с другом.

С этой целью применяются метод ожидания wait(), позволяющий приостановить выполнение потока до того момента пока не будет выполнено определенное условие и методы оповещения notify и notifyall.

Существует стандартный образец кода, которому важно следовать при использовании методов ожидания и оповещения. Потоку, ожидающему выполнения некоторого условия, всегда надлежит выполнять действия, подобные таким:

```
synchronized void dowhenCondition(){
   while(!условие) wait();
 //  выполнить то, что необходимо, если условие равно true
}
```

При использовании методов ожидания и оповещения следует придерживаться следующим правилам.

1. **Все функции по обеспечению взаимодействия потоков должны выполняться в рамках синхронизированного кода.**

Если это требование не удовлетворяется, состояние объекта не может считаться стабильным.

Например, если метод не объявлен как synchronized, после выполнения блока while нельзя твердо гарантировать, что проверяемое условие остается равным true, поскольку другой поток может изменить ситуацию.

2. **Один из важных аспектов метода wait состоит в том, что при приостановке выполнения потока он атомарным образом освобождает блокировку объекта.**

3. **Условие ожидания должно всегда проверяться циклически. Не достаточно проверить его только один раз, — после того как условие удовлетворено, оно может измениться вновь.**

Другими словами, нельзя заменять while на if.

Методы оповещения, в свою очередь, вызываются синхронизированным кодом и изменяют одно или несколько условий, разрешения которых могут ожидать какие-либо другие потоки.

Код оповещения обычно выглядит следующим образом:

```
synchronized void changeCondition(){
 //  ...   изменить некоторое значение,
      используемое в выражении условия ожидания
   notifyAll(); // или  notify();
}
```

Метод notifyAll оповещает все ожидающие потоки, а notify выбирает для этого только один поток.

Потоки могут ожидать выполнения условий (возможно, различных), относящихся к одному и тому же объекту.

Если условия действительно различны, для оповещения всех ожидающих потоков следует всегда использовать метод notifyAll вместо notify.

Метод notify позволяет несколько повысить эффективность кода и может применяться только в тех случаях, когда:

- все потоки ожидают выполнения одного и того же условия;
- самое большее один поток приобретет преимущества ввиду выполнения условия;

Во всех остальных ситуациях надлежит использовать notifyAll.

Рассмотрим несколько примеров.

Пример 1.

```
class Cell   {      //  Элемент очереди
   Cell   next;
   Object item;
   Cell(Object item){this.item  = item;}
}
class Queue {
  private Cell head, tail; //элементы в "голове" и "хвосте" очереди
  public synchronized void add(Object o) {
    Cell p = new Cell(o);
    if (tail == null)
      head = p;
   else
      tail.next = p;
      p.next = null;
      tail = p;
      notifyAll(); //  в очередь добавлен элемент
}
```

```
public synchronized object take()
                                    throws interruptedException{
  while(head==null){
    wait(); // Ждать уведомления о добавлении элемента
   Cell p = head; // запомнить элемент, занимающий
                                        место в "голове" очереди
   head = head.next; // Удалить элемент из "головы"
                                                очереди
   if (head == null); // проверить, не пуста ли очередь
        tail  = null;
    return p.item;
 }
}
```

Пример 2.

```
class MyThread implements Runnable {
   private static long time1=0;
   private static Object obj=new Object();
   private int flag;
   public MyThread(int flag1){
      flag=flag1;
   }
   synchronized public static void setTime(){
     time1++;
     synchronized(obj){
        obj.notifyAll();
     }
  }
```

```
 synchronized public void write() throws InterruptedException{
  if (flag==0)  {
    System.out.println(time1);}
  if(flag==1){
     long time2=time1;
     while(time1-time2<15){
        synchronized(obj) { obj.wait(); }
     }
     System.out.println("Hello1");
  }
  if(flag==2){
    long time2=time1;
    while(time1-time2<7){
    synchronized(obj){
       obj.wait();
     }
  }
 System.out.println("Hello2");
 }
}
```

```
public void run(){
  try{
     for(;;){
     if (flag==0){ setTime();}
     write();
     Thread.sleep(1000);}
  }
  catch(InterruptedException e){
     System.out.println(“error");
     return;
  }
}
public static void main(String[] args){
    Runnable th1=new MyThread(0);
    Runnable th2=new MyThread(1);
    Runnable th3=new MyThread(2);
    new Thread(th1).start();
    new Thread(th2).start();
    new Thread(th3).start();
}
}
```

Подробности, касающиеся wait, notify и notifyAll

Рассмотрим подробнее методы wait, notify и notifyAll:

1. **public final void wait(long timeout) throws InterruptedException**

Выполнение текущего потока приостанавливается до получения извещения или до истечения заданного интервала времени timeout.

Значение timeout задается в миллисекундах.

Если оно равно нулю, то ожидание не прерывается по таймауту, а продолжается до получения извещения.

2. **public final void wait(long timeout,**
int nanos) throws InterruptedException

Аналог предыдущего метода с возможностью более точного контроля времени; интервал тайм-аута представляет собой сумму двух параметров: timeout (в миллисекундах) и nanos (в наносекундах, значение в диапазоне 0–999999).

3. **public final void wait()**
throws InterruptedException

Эквивалентно wait(0).

4. **public final void notify()**

Посылает извещение ровно одному потоку, ожидающему выполнения некоторого условия.

Потоки, которые возобновляются лишь после выполнения данного условия, могут вызвать одну из разновидностей wait.

При этом выбрать извещаемый поток невозможно, поэтому данная форма notify используется лишь в тех случаях, когда точно известно, какие потоки ожидают событий, какие это события и сколько длится ожидание.

5. **public final void notifyAll()**

Посылает извещения всем потокам, ожидающим выполнения некоторого условия.

Обычно потоки стоят, пока какой-то другой поток не изменит некоторое условие.

Используя этот метод, управляющий условием поток извещает все ожидающие потоки об изменении условия.

Потоки, которые возобновляются лишь после выполнения данного условия, могут вызывать одну из разновидностей wait.

Все эти методы реализованы в классе Object.

Тем не менее они могут вызываться только из синхронизированных фрагментов, с использованием блокировки объекта, в котором они применяются

Планирование потоков

В отношении потоков в Java даются лишь общие гарантии.

В качестве количественного показателя важности выполняемых функций потоку ставится в соответствие приоритет (priority), значение которого используется системой для определения того, какой из потоков должен выполняться в каждый момент времени.

В системе с N процессорами одновременно может выполняться N высокоприоритетных потоков.

Потокам, обладающим более низкими значениями приоритета, ресурсы процессоров обычно отдаются только в том случае, когда более важные потоки блокированы.

Чтобы предотвратить вероятность зависания, система вправе предоставлять ресурсы низкоприоритетным потокам и в другие моменты времени — в связи с так называемым старением приоритетов (priority aging), — но прикладные программы не в состоянии серьезно использовать такую возможность.

Определение расписания приоритетного обслуживания потоков с прерываниями входит в компетенцию конкретной виртуальной машины Java.

Зачастую твердых гарантий поведения системы в отношении планирования заданий не существует — можно только ожидать, что предпочтение в том или ином случае будет отдано потоку, обладающему более высоким приоритетом.

Исходное значение приоритета потока соответствует приоритету потока-"родителя".

Величина приоритета может быть изменена посредством вызова метода setPriority с аргументом из интервала, который задается значениями именованных констант **MIN_PRIORITY** и **MAX_PRIORITY**, определенных в составе класса Thread.

Приоритету потока, предлагаемому по умолчанию, соответствует константа **NORM_PRIORITY**.

Приоритет работающего потока допускается изменять в любой момент.

Если приоритет потока понижается, система может передать вычислительные ресурсы другому потоку, поскольку исходный утратит "членство" в группе потоков с наивысшими приоритетами.

Чтобы получить текущее значение приоритета потока, следует воспользоваться методом **getPriority**.

Существуют методы в классе Thread управляющие планировкой потоков в системе.

1. **public static void sleep(long millis)**
   throws InterruptedException

Приостанавливает работу текущего потока как минимум на указанное число миллисекунд.

“Как минимум” означает, что не существует гарантий возобновления работы потока точно в указанное время.

2. **public static void sleep(long millis, int nanos)**
   throws InterruptedException

Приостанавливает работу текущего потока как минимум на указанное число миллисекунд и дополнительное число наносекунд.

Значение интервала в наносекундах лежит в диапазоне 0–999999.

3. **public static void yield()**

Текущий поток передает управление, чтобы дать возможность работать и другим исполняемым потокам.

Планировщик потоков выбирает новый поток среди исполняемых потоков в системе.

При этом может быть вызван поток, только что уступивший управление, если его приоритет окажется самым высоким.

Рассмотрим пример.

```
class Babble extends Thread {
  static boolean doYield; // передавать управление другим потокам?
  static int howOften; // количеств повторов при выводе
  String word;
 Babble(String whatToSay) { word = whatToSay; }
  public void run() {
    for (int i = 0; i < howOften; i++) {
      System.out.println(word);
      if (doYield) yield(); // передать управление другому потоку
    }
   }
 public static void main(String[] args) {
  howOften = Integer.parseInt(args[1]);
  doYield = new Boolean(args[0]).booleanValue();
  Thread cur = currentThread();
  cur.setPriority(Thread.MAX_PRIORITY);
  for (int i = 2; i < args.length; i++)
    new Babble(args[i]).start();
 }
}
```

Когда потоки работают, не передавая управления друг другу, им отводятся большие кванты времени — обычно этого бывает достаточно, чтобы закончить вывод в монопольном режиме.

Например, при запуске программы с присвоением doYield значения false:

Babble false 2 Did DidNot

результат будет выглядеть следующим образом:

Did

Did

DidNot

DidNot

Если же каждый поток передает управление после очередного println, то другие потоки также получат возможность работать.

Если присвоить doYield значение true:

Babble true 2 Did DidNot

то остальные потоки также смогут выполняться между очередными выводами и, в свою очередь, будут уступать управление, что приведет (как вариант) к следующему:

Did

DidNot

Did

DidNot

Метод getState()

Это нестатический метод. Возвращает Enum. Пример использования:

Runnable r=….;

Thread th=new Thread(r);

Thread.Enumj en=th.getState();

Возможные значения:

.new – поток не бы еще запущен.

.runnable – поток выполняется.

. blocked- поток блокирован

. waiting – поток ждет окончания выполнения действия другим потоком

. time_waiting - поток ждет окончания выполнения действия другим потоком до определенного времени

.terminated – поток завершен.