Лекция 11. Многопоточное программирование.

Оглавление

До сих пор во всех рассматриваемых примерах подразумевалось, что в один момент времени исполняется лишь одно *выражение* или действие. Однако начиная с самых первых версий, виртуальные машины Java поддерживают многопоточность, т.е. поддержку нескольких потоков исполнения (***threads***) одновременно.

Многопоточная архитектура

Реализацию многопоточной архитектуры проще всего представить себе для системы, в которой есть несколько центральных вычислительных процессоров. В этом случае для каждого из них можно выделить задачу, которую он будет выполнять. В результате несколько задач будут обслуживаться одновременно.

Однако возникает вопрос – каким же тогда образом обеспечивается многопоточность в системах с одним центральным процессором, который, в принципе, выполняет лишь одно *вычисление* в один момент времени? В таких системах применяется процедура квантования времени (***time-slicing***). Время разделяется на небольшие интервалы. Перед началом каждого интервала принимается решение, какой именно *поток* выполнения будет отрабатываться на протяжении этого кванта времени. За счет частого переключения между задачами эмулируется многопоточная *архитектура*.

Процедура квантования времени поддерживает приоритеты (*priority*) задач. В *Java* приоритет представляется целым числом. Чем больше число, тем

выше приоритет. Строгих правил работы с приоритетами нет, каждая реализация может вести себя по-разному на разных платформах. Однако есть *общее правило – поток* с более высоким приоритетом будет получать большее количество квантов времени на *исполнение* и таким образом сможет быстрее выполнять свои действия и реагировать на поступающие данные.

Рассмотрим здесь же еще одно свойство потоков. Раньше, когда рассматривались однопоточные приложения, завершение вычислений однозначно приводило к завершению выполнения программы. Теперь же *приложение* должно работать до тех пор, пока есть хоть один действующий *поток* исполнения. В то же время часто бывают нужны обслуживающие потоки, которые не имеют никакого смысла, если они остаются в системе одни. Например, автоматический сборщик мусора в *Java* запускается в виде фонового (низкоприоритетного) процесса. Его задача – отслеживать объекты, которые уже не используются другими потоками, и затем уничтожать их, освобождая оперативную *память*. Понятно, что работа одного потока *garbage collector* 'а не имеет никакого смысла.

Такие обслуживающие потоки называют ***демонами*** (***daemon***), это свойство можно установить любому потоку. В итоге *приложение* выполняется до тех пор, пока есть хотя бы один *поток* не- *демон*.

Рассмотрим, как потоки реализованы в *Java*.

Базовые классы для работы с потоками

Класс Thread

Поток выполнения в Java представляется экземпляром класса *Thread*. Для того, чтобы написать свой поток исполнения, необходимо наследоваться от этого класса и переопределить метод `run()`. Например,

```
public class MyThread extends Thread {
   public void run() {
      // некоторое долгое действие, вычисление
      long sum=0;
      for (int i=0; i<1000; i++) {
         sum+=i;
      }
      System.out.println(sum);
   }
}
```

Метод `run()` содержит действия, которые должны выполняться в новом потоке исполнения. Чтобы запустить его, необходимо создать экземпляр класса-наследника и вызвать унаследованный метод `start()`, который сообщает виртуальной машине, что требуется запустить новый поток исполнения и начать выполнять в нем метод `run()`.

```
MyThread t = new MyThread();
t.start();
```

В результате чего на консоли появится результат: `499500`

Когда метод `run()` завершен (в частности, встретилось выражение `return`), поток выполнения останавливается. Однако ничто не препятствует записи бесконечного цикла в этом методе. В результате поток не прервет своего исполнения и будет остановлен только при завершении работы всего приложения.

Интерфейс Runnable

Описанный подход имеет один недостаток. Поскольку в Java множественное наследование отсутствует, требование наследоваться от *Thread* может привести к конфликту. Если еще раз посмотреть на приведенный выше пример, станет понятно, что наследование производилось только с целью переопределения метода `run()`. Поэтому предлагается более простой способ создать свой поток исполнения. Достаточно реализовать интерфейс `Runnable`, в котором объявлен только один метод – уже знакомый `void run()`. Запишем пример, приведенный выше, с помощью этого интерфейса:

```
public class MyRunnable implements Runnable {
   public void run() {
      // некоторое долгое действие, вычисление
      long sum=0;
      for (int i=0; i<1000; i++) {
         sum+=i;
      }
      System.out.println(sum);
   }
}
```

Также незначительно меняется процедура запуска потока:

```
Runnable r = new MyRunnable();
Thread t = new Thread(r);
t.start();
```

Если раньше объект, представляющий сам поток выполнения, и объект с методом `run()`, реализующим необходимую функциональность, были объединены в одном экземпляре класса `MyThread`, то теперь они разделены. Какой из двух подходов удобней, решается в каждом конкретном случае.

Подчеркнем, что `Runnable` не является полной заменой классу `Thread`, поскольку создание и запуск самого потока исполнения возможно только через метод `Thread.start()`.

Работа с приоритетами

Рассмотрим, как в Java можно назначать *потокам приоритеты*. Для этого в классе `Thread` существуют методы `getPriority()` и `setPriority()`, а также объявлены три константы:

`MIN_PRIORITY`
`MAX_PRIORITY`
`NORM_PRIORITY`

Из названия понятно, что их значения описывают минимальное, максимальное и нормальное (по умолчанию) значения приоритета.

Рассмотрим следующий пример:

```
public class ThreadTest implements Runnable {
  public void run() {
    double calc;
    for (int i=0; i<50000; i++) {
      calc=Math.sin(i*i);
      if (i%10000==0) {
        System.out.println(getName()+ " counts " + i/10000);
      }
    }
  }

  public String getName() {
     return Thread.currentThread().getName();
  }

  public static void main(String s[]) {
    // Подготовка потоков
    Thread t[] = new Thread[3];
    for (int i=0; i<t.length; i++) {
      t[i]=new Thread(new ThreadTest(), "Thread "+i);
    }
    // Запуск потоков
    for (int i=0; i<t.length; i++) {
      t[i].start();
      System.out.println(t[i].getName()+" started");
    }
  }
}
```

В примере используется несколько новых методов класса `Thread`:

`getName()`

Обратите внимание, что конструктору класса `Thread` передается два параметра. К реализации `Runnable` добавляется строка. Это имя потока, которое используется только для упрощения его идентификации. Имена нескольких потоков могут совпадать. Если его не задать, то Java генерирует простую строку вида`"Thread-"` и номер потока (вычисляется простым счетчиком). Именно это имя возвращается методом `getName()`. Его можно сменить с помощью метода `setName()`.

`currentThread()`

Этот статический метод позволяет в любом месте кода получить ссылку на объект класса `Thread`, представляющий текущий поток исполнения.

Результат работы такой программы будет иметь следующий вид:

```
Thread 0 started
Thread 1 started
Thread 2 started
Thread 0 counts 0
Thread 1 counts 0
Thread 2 counts 0
Thread 0 counts 1
```

```
Thread 1 counts 1
Thread 2 counts 1
Thread 0 counts 2
Thread 2 counts 2
Thread 1 counts 2
Thread 2 counts 3
Thread 0 counts 3
Thread 1 counts 3
Thread 2 counts 4
Thread 0 counts 4
Thread 1 counts 4
```

Мы видим, что все три потока были запущены один за другим и начали проводить вычисления. Видно также, что потоки исполняются без определенного порядка, случайным образом. Тем не менее, в среднем они движутся с одной скоростью, никто не отстает и не догоняет.

Введем в программу работу с приоритетами, расставим разные значения для разных потоков и посмотрим, как это скажется на выполнении. Изменяется только метод `main()`.

```
public static void main(String s[]) {
// Подготовка потоков
   Thread t[] = new Thread[3];
   for (int i=0; i<t.length; i++) {
      t[i]=new Thread(new ThreadTest(), "Thread "+i);
      t[i].setPriority(Thread.MIN_PRIORITY +
      (Thread.MAX_PRIORITY - Thread.MIN_PRIORITY)/t.length*i);
   }

// Запуск потоков
   for (int i=0; i<t.length; i++) {
      t[i].start();
      System.out.println(t[i].getName()+" started");
   }
}
```

Формула вычисления приоритетов позволяет равномерно распределить все допустимые значения для всех запускаемых потоков. На самом деле, константа минимального приоритета имеет значение `1`, максимального `10`, нормального `5`. Так что в простых программах можно явно пользоваться этими величинами и указывать в качестве, например, пониженного приоритета значение `3`.

Результатом работы будет:

```
Thread 0 started
Thread 1 started
Thread 2 started
Thread 2 counts 0
Thread 2 counts 1
Thread 2 counts 2
Thread 2 counts 3
Thread 2 counts 4
Thread 0 counts 0
Thread 1 counts 0
Thread 1 counts 1
Thread 1 counts 2
Thread 1 counts 3
Thread 1 counts 4
```

```
Thread 0 counts 1
Thread 0 counts 2
Thread 0 counts 3
Thread 0 counts 4
```

Потоки, как и раньше, стартуют последовательно. Но затем мы видим, что чем выше приоритет, тем быстрее отрабатывает поток. Тем не менее, весьма показательно, что поток с минимальным приоритетом (`Thread 0`) все же получил возможность выполнить одно действие раньше, чем отработал поток с более высоким приоритетом (`Thread 1`). Это говорит о том, что приоритеты не делают систему однопоточной, выполняющей единовременно лишь один поток с наивысшим приоритетом. Напротив, приоритеты позволяют одновременно работать над несколькими задачами с учетом их важности.

Если увеличить параметры метода (выполнять 500000 вычислений, а не 50000, и выводить сообщение каждое 1000-е вычисление, а не 10000-е), то можно будет наглядно увидеть, что все три потока имеют возможность выполнять свои действия одновременно, просто более высокий приоритет позволяет выполнять их чаще.

Демон-потоки

Демон -потоки позволяют описывать фоновые процессы, которые нужны только для обслуживания основных потоков выполнения и не могут существовать без них. Для работы с этим свойством существуют методы `setDaemon()` и `isDaemon()`.

Несмотря на то, что *демон* -поток никогда не выходит из метода `run()`, виртуальная машина прекращает работу, как только все не- *демон* -потоки завершаются.

Приостановление выполнения потока.

Статический метод `sleep()`класса `Thread` приостанавливает выполнение текущего потока на указанное количество миллисекунд. Обратите внимание, что метод требует обработки исключения `InterruptedException`. Он связан с возможностью активизировать метод, который приостановил свою работу. Например, если поток занят выполнением метода `sleep()`, то есть бездействует на протяжении указанного периода времени, его можно вывести из этого состояния, вызвав метод `interrupt()` из другого потока выполнения. В результате метод `sleep()` прервется исключением `InterruptedException`.

Кроме метода `sleep()`, существует еще один статический метод `yield()` без параметров. Когда поток вызывает его, он временно приостанавливает свою работу и позволяет отработать другим потокам. Один из методов обязательно должен применяться внутри бесконечных циклов ожидания, иначе есть риск, что такой ничего не делающий поток затормозит работу остальных потоков.

Синхронизация

При многопоточной архитектуре приложения возможны ситуации, когда несколько потоков будут одновременно работать с одними и теми же данными, используя их значения и присваивая новые. В таком случае результат работы программы становится невозможно предугадать, глядя только на исходный код. Финальные значения переменных будут зависеть от случайных факторов, исходя из того, какой *поток* какое действие успел сделать первым или последним.

Рассмотрим пример:

```
public class ThreadTest {
  private int a=1, b=2;
  public void one() {
    a=b;
  }
  public void two() {
    b=a;
  }

  public static void main(String s[]) {
    int a11=0, a22=0, a12=0;
    for (int i=0; i<1000; i++) {
      final ThreadTest o = new ThreadTest();

      // Запускаем первый поток, который вызывает один метод
      new Thread() {
          public void run() {
            o.one();
          }
      }.start();

      // Запускаем второй поток, который вызывает второй метод
      new Thread() {
         public void run() {
           o.two();
         }
      }.start();

      // даем потокам время отработать
      try {
         Thread.sleep(100);
      } catch (InterruptedException e) {}

      // анализируем финальные значения
      if (o.a==1 && o.b==1) a11++;
      if (o.a==2 && o.b==2) a22++;
      if (o.a!=o.b) a12++;
    }
    System.out.println(a11+" "+a22+" "+a12);
  }
}
```

В этом примере два потока исполнения одновременно обращаются к одному и тому же объекту, вызывая у него два разных метода, `one()` и

`two()`. Эти методы пытаются приравнять два *поля класса* `a` и `b` друг другу, но в разном порядке. Учитывая, что исходные значения полей равны `1` и `2`, соответственно, можно было ожидать, что после того, как потоки завершат свою работу, поля будут иметь одинаковое *значение*. Однако понять, какое из двух возможных значений они примут, уже невозможно. Посмотрим на результат программы:

```
135 864 1
```

Первое число показывает, сколько раз из тысячи обе переменные приняли *значение* `1`. Второе число соответствует значению `2`. Такое сильное преобладание одного из значений обусловлено последовательностью запусков потоков. Если ее изменить, то и количества случаев с `1` и `2` также меняются местами. Третье же число сообщает, что на тысячу случаев произошел один, когда поля вообще обменялись значениями!

При количестве итераций, равном 10000, были получены следующие данные, которые подтверждают сделанные выводы:

```
494 9498 8
```

А если убрать задержку перед анализом результатов, то получаемые данные радикально меняются:

```
0 3 997
```

Видимо, потоки просто не успевают отработать.

Итак, наглядно показано, сколь сильно и непредсказуемо может меняться результат работы одной и той же программы, применяющей многопоточную архитектуру. Необходимо учитывать, что в приведенном простом примере задержки создавались вручную методом `Thread.sleep()`. В реальных сложных системах задержки могут возникать в местах проведения сложных операций, их *длина* непредсказуема и оценить их последствия невозможно.

Для более глубокого понимания принципов многопоточной работы в *Java* рассмотрим организацию памяти в виртуальной машине для нескольких потоков.

Хранение переменных в памяти

Виртуальная машина поддерживает основное хранилище данных (main storage), в котором сохраняются значения всех переменных и которое используется всеми потоками. Под переменными здесь понимаются поля объектов и классов, а также элементы массивов. Что касается локальных переменных и параметров методов, то их значения не могут быть доступны другим потокам, поэтому они не представляют интереса.

Для каждого потока создается его собственная рабочая память (*working memory*), в которую перед использованием копируются значения всех переменных.

Рассмотрим основные операции, доступные для потоков при работе с памятью:

`use` – чтение значения переменной из рабочей памяти потока;

`assign` – запись значения переменной в рабочую память потока;

`read` – получение значения переменной из основного хранилища;

`load` – сохранение значения переменной, прочитанного из основного хранилища, в рабочей памяти;

`store` – передача значения переменной из рабочей памяти в основное хранилище для дальнейшего хранения;

`write` – сохраняет в основном хранилище значение переменной, переданной командой store.

Подчеркнем, что перечисленные команды не являются методами каких-либо классов, они недоступны программисту. Сама виртуальная машина использует их для обеспечения корректной работы потоков исполнения.

Поток, работая с переменной, регулярно применяет команды `use` и `assign` для использования ее текущего значения и присвоения нового. Кроме того, должны осуществляться действия по передаче значений в основное хранилище и из него. Они выполняются в два этапа. При получении данных сначала основное хранилище считывает значение командой `read`, а затем поток сохраняет результат в своей рабочей памяти командой `load`. Эта пара команд всегда выполняется вместе именно в таком порядке, т.е. нельзя выполнить одну, не выполнив другую. При отправлении данных сначала поток считывает значение из рабочей памяти командой `store`, а затем основное хранилище сохраняет его командой `write`. Эта пара команд также всегда выполняется вместе именно в таком порядке, т.е. нельзя выполнить одну, не выполнив другую.

Набор этих правил составлялся с тем, чтобы операции с памятью были достаточно строги для точного анализа их результатов, а с другой стороны, правила должны оставлять достаточное пространство для различных технологий оптимизаций (регистры, очереди, кэш и т.д.).

Последовательность команд подчиняется следующим правилам:

- все действия, выполняемые одним потоком, строго упорядочены, т.е. выполняются одно за другим;
- все действия, выполняемые с одной переменной в основном хранилище памяти, строго упорядочены, т.е. следуют одно за другим.

За исключением некоторых дополнительных очевидных правил, больше никаких ограничений нет. Например, если поток изменил значение сначала одной, а затем другой переменной, то эти изменения могут быть переданы в основное хранилище в обратном порядке.

Поток создается с чистой рабочей памятью и должен перед использованием загрузить все необходимые переменные из основного хранилища. Любая переменная сначала создается в основном хранилище и лишь затем копируется в рабочую память потоков, которые будут ее применять.

Таким образом, потоки никогда не взаимодействуют друг с другом напрямую, только через главное хранилище.

Модификатор volatile

При объявлении полей объектов и классов может быть указан модификатор `volatile`. Он устанавливает более строгие правила работы со значениями переменных.

Если поток собирается выполнить команду `use` для `volatile` переменной, то требуется, чтобы предыдущим действием с этой переменной было обязательно `load`, и наоборот – операция `load` может выполняться только перед `use`. Таким образом, переменная и главное хранилище всегда имеют самое последнее значение этой переменной.

Аналогично, если поток собирается выполнить команду `store` для `volatile` переменной, то требуется, чтобы предыдущим действием над этой переменной было обязательно `assign`, и наоборот – операция `assign` может выполняться, только если следующей будет `store`. Таким образом, переменная и главное хранилище всегда имеют самое последнее значение этой переменной.

Наконец, если проводятся операции над несколькими `volatile` переменными, то передача соответствующих изменений в основное хранилище должна проводиться строго в том же порядке.

При работе с обычными переменными компилятор имеет больше пространства для маневра. Например, при благоприятных обстоятельствах может оказаться возможным предсказать значение переменной, заранее вычислить и сохранить его, а затем в нужный момент использовать уже готовым.

Следует обратить внимание на два 64-разрядных типа, `double` и `long`. Поскольку многие платформы поддерживают лишь 32-битную память, величины этих типов рассматриваются как две переменные и все описанные действия выполняются независимо для двух половинок таких значений. Конечно, если производитель виртуальной машины считает возможным, он может обеспечить атомарность операций и над этими типами. Для `volatile` переменных это является обязательным требованием.

Блокировки

В основном хранилище для каждого объекта поддерживается блокировка (`lock`), над которой можно произвести два действия – установить (`lock`) и снять (`unlock`). Только один поток в один момент времени может установить блокировку на некоторый объект. Если до того, как этот поток выполнит операцию `unlock`, другой поток попытается установить блокировку, его выполнение будет приостановлено до тех пор, пока первый поток не отпустит ее.

Операции `lock` и `unlock` накладывают жесткое ограничение на работу с переменными в рабочей памяти потока. После успешно выполненного `lock` рабочая память очищается и все переменные необходимо заново считывать из основного хранилища. Аналогично, перед операцией `unlock` необходимо все переменные сохранить в основном хранилище.

Важно подчеркнуть, что блокировка является чем-то вроде флага. Если блокировка на объект установлена, это не означает, что данным объектом нельзя пользоваться, что его поля и методы становятся недоступными, – это не так. Единственное действие, которое становится невозможным, – установка этой же блокировки другим потоком, до тех пор, пока первый поток не выполнит `unlock`.

В Java-программе для того, чтобы воспользоваться механизмом блокировок, существует ключевое слово `synchronized`. Оно может быть применено в двух вариантах – для объявления `synchronized` -блока и как модификатор метода. В обоих случаях действие его примерно одинаковое.

`Synchronized` -блок записывается следующим образом:

```
synchronized (ref) {
   ...
}
```

Прежде, чем начать выполнять действия, описанные в этом блоке, поток обязан установить блокировку на объект, на который ссылается переменная `ref` (поэтому она не может быть `null`). Если другой поток уже установил блокировку на этот объект, то выполнение первого потока приостанавливается до тех пор, пока не удастся выполнить операцию `lock`.

После этого блок выполняется. При завершении исполнения (как успешном, так и в случае ошибок) производится операция `unlock`, чтобы освободить объект для других потоков.

Рассмотрим пример:

```
public class ThreadTest implements Runnable {
   private static ThreadTest shared = new ThreadTest();
   public void process() {
      for (int i=0; i<3; i++) {
         System.out.println(
                     Thread.currentThread().
            getName()+" "+i);
         Thread.yield();
      }
   }

   public void run() {
      shared.process();
   }
   public static void main(String s[]) {
      for (int i=0; i<3; i++) {
         new Thread(new ThreadTest(),

            "Thread-"+i).start();
      }
   }
}
```

В этом простом примере три потока вызывают метод у одного объекта, чтобы тот распечатал три значения. Результатом будет:

```
Thread-0 0
```

```
Thread-1 0
Thread-2 0
Thread-0 1
Thread-2 1
Thread-0 2
Thread-1 1
Thread-2 2
Thread-1 2
```

То есть все потоки одновременно работают с одним методом одного объекта. Заключим обращение к методу в `synchronized` -блок:

```
public void run() {
   synchronized (shared) {
      shared.process();
   }
}
```

Теперь результат будет строго упорядочен:

```
Thread-0 0
Thread-0 1
Thread-0 2
Thread-1 0
Thread-1 1
Thread-1 2
Thread-2 0
Thread-2 1
Thread-2 2
```

`Synchronized` -методы работают аналогичным образом. Прежде, чем начать выполнять их, поток пытается заблокировать объект, у которого вызывается метод. После выполнения блокировка снимается. В предыдущем примере аналогичной упорядоченности можно было добиться, если использовать не `synchronized` -блок, а объявить метод `process()` синхронизированным.

Также допустимы методы `static synchronized`. При их вызове блокировка устанавливается на объект класса `Class`, отвечающего за тип, у которого вызывается этот метод.

При работе с блокировками всегда надо помнить о возможности появления `deadlock` – взаимных блокировок, которые приводят к зависанию программы. Если один поток заблокировал один ресурс и пытается заблокировать второй, а другой поток заблокировал второй и пытается заблокировать первый, то такие потоки уже никогда не выйдут из состояния ожидания.

Рассмотрим простейший пример:

```
public class DeadlockDemo {

  // Два объекта-ресурса
  public final static Object one=new Object(), two=new Object();

  public static void main(String s[]) {

    // Создаем два потока, которые будут
    // конкурировать за доступ к объектам
       // one и two
```

```
    Thread t1 = new Thread() {
      public void run() {
        // Блокировка первого объекта
        synchronized(one) {
        Thread.yield();
          // Блокировка второго объекта
          synchronized (two) {
            System.out.println("Success!");
          }
        }
      }
    };
    Thread t2 = new Thread() {
      public void run() {
        // Блокировка второго объекта
        synchronized(two) {
          Thread.yield();
          // Блокировка первого объекта
          synchronized (one) {
            System.out.println("Success!");
          }
        }
      }
    };

    // Запускаем потоки
    t1.start();
    t2.start();
  }
}
```

Если запустить такую программу, то она никогда не закончит свою работу. Обратите внимание на вызовы метода `yield()` в каждом потоке. Они гарантируют, что когда один поток выполнил первую блокировку и переходит к следующей, второй поток находится в таком же состоянии. Очевидно, что в результате оба потока "замрут", не смогут продолжить свое выполнение. Первый поток будет ждать освобождения второго объекта, и наоборот. Именно такая ситуация называется "мертвой блокировкой", или `deadlock`. Если один из потоков успел бы заблокировать оба объекта, то программа успешно бы выполнилась до конца. Однако многопоточная архитектура не дает никаких гарантий, как именно потоки будут выполняться друг относительно друга. Задержки (которые в примере моделируются вызовами `yield()`) могут возникать из логики программы (необходимость произвести вычисления), действий пользователя (не сразу нажал кнопку "ОК"), занятости ОС (из-за нехватки физической оперативной памяти пришлось воспользоваться виртуальной), значений *приоритетов потоков* и так далее.

В Java нет никаких средств распознавания или предотвращения ситуаций `deadlock`. Также нет способа перед вызовом синхронизированного метода узнать, заблокирован ли уже объект другим потоком. Программист сам должен строить работу программы таким образом, чтобы неразрешимые

блокировки не возникали. Например, в рассмотренном примере достаточно было организовать блокировки объектов в одном порядке (всегда сначала первый, затем второй) – и программа всегда выполнялась бы успешно.

Опасность возникновения взаимных блокировок заставляет с особенным вниманием относиться к работе с потоками. Например, важно помнить, что если у объекта потока был вызван метод `sleep(..)`, то такой поток будет бездействовать определенное время, но при этом все заблокированные им объекты будут оставаться недоступными для блокировок со стороны других потоков, а это потенциальный `deadlock`. Такие ситуации крайне сложно выявить путем тестирования и отладки, поэтому вопросам синхронизации надо уделять много времени на этапе проектирования.

Методы wait(), notify(), notifyAll() класса Object

Наконец, перейдем к рассмотрению трех методов класса `Object`, завершая описание механизмов поддержки многопоточности в *Java*.

Каждый *объект* в *Java* имеет не только блокировку для `synchronized` блоков и методов, но и так называемый `wait-set`, набор потоков исполнения. Любой *поток* может вызвать метод `wait()` любого объекта и таким образом попасть в его `wait-set`. При этом выполнение такого потока приостанавливается до тех пор, пока другой *поток* не вызовет у этого же объекта метод `notifyAll()`, который пробуждает все потоки из `wait-set`. Метод `notify()` пробуждает один случайно выбранный *поток* из данного набора.

Однако применение этих методов связано с одним важным ограничением. Любой из них может быть вызван потоком у объекта только после установления блокировки на этот *объект*. То есть либо внутри `synchronized` -блока с ссылкой на этот *объект* в качестве аргумента, либо обращения к методам должны быть в синхронизированных методах класса самого объекта. Рассмотрим пример:

```
public class WaitThread implements Runnable {
      private Object shared;

      public WaitThread(Object o) {
         shared=o;
}

public void run() {
   synchronized (shared) {
      try {
         shared.wait();
      } catch (InterruptedException e) {}
      System.out.println("after wait");
   }
}
```

```
public static void main(String s[]) {
   Object o = new Object();
   WaitThread w = new WaitThread(o);
   new Thread(w).start();
   try {
      Thread.sleep(100);
   } catch (InterruptedException e) {}
   System.out.println("before notify");
   synchronized (o) {
      o.notifyAll();
   }
   }
}
```

Результатом программы будет:

```
before notify
after wait
```

Обратите внимание, что метод `wait()`, как и `sleep()`, требует обработки `InterruptedException`, то есть его выполнение также можно прервать методом `interrupt()`.

В заключение рассмотрим более сложный пример для трех потоков:

```
public class ThreadTest implements Runnable {
      final static private Object shared=new Object();
      private int type;
      public ThreadTest(int i) {
         type=i;
      }

public void run() {
   if (type==1 || type==2) {
      synchronized (shared) {
         try {
            shared.wait();
         } catch (InterruptedException e) {}
         System.out.println("Thread "+type+" after wait()");
   }
} else {
   synchronized (shared) {
      shared.notifyAll();
      System.out.println("Thread "+type+" after notifyAll()");
      }
   }
}

public static void main(String s[]) {
   ThreadTest w1 = new ThreadTest(1);
   new Thread(w1).start();
   try {
      Thread.sleep(100);
   } catch (InterruptedException e) {}
   ThreadTest w2 = new ThreadTest(2);
   new Thread(w2).start();
```

```
    try {
       Thread.sleep(100);
    } catch (InterruptedException e) {}
    ThreadTest w3 = new ThreadTest(3);
    new Thread(w3).start();
    }
}
```

Результатом работы программы будет:

```
Thread 3 after notifyAll()
Thread 1 after wait()
Thread 2 after wait()
```

Рассмотрим, что произошло. Во-первых, был запущен *поток* `1`, который тут же вызвал метод `wait()` и приостановил свое выполнение. Затем то же самое произошло с потоком `2`. Далее начинает выполняться *поток* `3`.

Сразу обращает на себя внимание следующий факт. Еще *поток* `1` вошел в `synchronized` -блок, а стало быть, установил блокировку на *объект* `shared`. Но, судя по результатам, это не помешало и потоку `2` зайти в `synchronized` -блок, а затем и потоку `3`. Причем, для последнего это просто необходимо, иначе как можно "разбудить" потоки `1` и `2`?

Можно сделать *вывод*, что потоки, прежде чем приостановить выполнение после вызова метода `wait()`, отпускают все занятые блокировки. Итак, вызывается метод `notifyAll()`. Как уже было сказано, все потоки из `wait-set` возобновляют свою работу. Однако чтобы корректно продолжить *исполнение*, необходимо вернуть блокировку на *объект*, ведь следующая *команда* также находится внутри `synchronized` -блока!

Получается, что даже после вызова `notifyAll()` все потоки не могут сразу возобновить работу. Лишь один из них сможет вернуть себе блокировку и продолжить работу. Когда он покинет свой `synchronized` -блок и отпустит *объект*, второй *поток* возобновит свою работу, и так далее. Если по какой-то причине *объект* так и не будет освобожден, *поток* так никогда и не выйдет из метода `wait()`, даже если будет вызван метод `notifyAll()`. В рассмотренном примере потоки один за другим смогли возобновить свою работу.

Кроме того, определен метод `wait()` с параметром, который задает период тайм-аута, по истечении которого *поток* сам попытается возобновить свою работу. Но начать ему придется все равно с повторного получения блокировки.

Заключение

В этой лекции были рассмотрены принципы построения многопоточного приложения. В начале разбирались достоинства и недостатки такой архитектуры – как правило ОС не выделяет отдельный *процессор* под каждый процесс, а значит применяется процедура *time* slicing. Было выделено три признака, указывающие на целесообразность запуска нескольких потоков в рамках программы.

Основу работы с потоками в *Java* составляют *интерфейс* `Runnable` и *класс* `Thread`. С их помощью можно запускать и останавливать потоки, менять их свойства, среди которых основные: приоритет и свойство *daemon*. Главная проблема, возникающая в таких программах - одновременный *доступ* нескольких потоков к одним и тем же данным, в первую *очередь* -– к полям объектов. Для понимания, как в *Java* решается эта задача, был сделан краткий обзор по организации памяти в *JVM*, работы с переменными и блокировками. Блокировки, несмотря на название, сами по себе не ограничивают *доступ* к переменной. Программист использует их через *ключевое слово* `synchronized`, которое может быть указано в сигнатуре метода или в начале блока. В результате выполнение не будет продолжено, пока *блокировка* не освободится.

Новый механизм порождает новую проблему - взаимные блокировки (*deadlock*), к которой программист всегда должен быть готов, тем более, что *Java* не имеет встроенных средств для определения такой ситуации. В лекции разбирался пример, как организовать работу программы без "зависания" ожидающих потоков.

В завершение рассматривались специализированные методы базового класса `Object`, которые также позволяют управлять последовательностью работы потоков.