# Лекция 12. Потоки в Java

Процесс — это совокупность кода и данных, разделяющих общее виртуальное адресное пространство. Чаще всего одна программа состоит из одного процесса. Процессы изолированы друг от друга, поэтому прямой доступ к памяти чужого процесса невозможен (взаимодействие между процессами осуществляется с помощью специальных средств). Для каждого процесса ОС создает так называемое «виртуальное адресное пространство», к которому процесс имеет прямой доступ. Это пространство принадлежит процессу, содержит только его данные и находится в полном его распоряжении. Операционная система же отвечает за то, как виртуальное пространство процесса проецируется на физическую память.

Поток – это единица исполнения инструкции процесса. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса.

При работе в многопоточной среде для распределяемых ресурсов необходимо обеспечивать 2 свойства:

* **Atomicity**: чтение и запись полей всех типов происходит атомарно (как единое целое, в рамках одной инструкции или транзакции).
* **Visibility**: изменения, сделанные одним потоком видны другим потокам.

В Java реализацией потока является класс java.lang.Thread. Существует 2 способа создания отдельного потока:

* **Способ 1:**Создание потомка класса ***Thread*** и переопределение его метода **run() (см. пример 1)**.
* **Способ 2:** Создание объекта типа ***Thread*** с передачей ему в конструкторе объект класса, реализующего интерфейс ***Runnable***. Этот интерфейс содержит метод **run()**, который будет выполняться в новом потоке **(см. пример 2)**.

|  |
| --- |
| **public** **class** CustomThread **extends** Thread  {  String threadName;  **public** CustomThread()  {  threadName=getName();  }  @Override  **public** **synchronized** **void** start()  {  **super**.start();  System.***out***.println(threadName + ": CustomThread started");  }  @Override  **public** **void** run()  {  System.***out***.println(threadName + ": CustomThread.run started");  System.***out***.println(threadName + " thread name=" + getName());  System.***out***.println(threadName + ": CustomThread.run finished");  }  } |
| **public** **class** Main  {  **public** **static** **void** main(String[] args)  {  *executeCustomThread*();  }  **public** **static** **void** executeCustomThread()  {  System.***out***.println(" ------- executeCustomThread started ------ ");  Thread simpleThread=**new** CustomThread();  simpleThread.start();  System.***out***.println(" ------- executeCustomThread finished ---- ");  }  } |
| Пример 1. Наследование класса Thread для создания собственного потока |

|  |
| --- |
| **public** **class** CustomRunnable **implements** Runnable  {  **private** Thread currentThread;  **private** String currentThreadName;  **public** **void** run()  {  System.***out***.println(" + CustomRunnable.run()");  currentThread=Thread.*currentThread*();  currentThreadName=currentThread.getName();  System.***out***.println(currentThreadName + " currentThread: " + currentThread);  System.***out***.println(currentThreadName + " currentThreadName: " + currentThreadName);  System.***out***.println(currentThreadName + " currentThread: " + currentThread);  System.***out***.println(" - CustomRunnable.run()");  }  } |
| **public** **class** Main  {  **public** **static** **void** main(String[] args)  {  *executeCustomRunnable*();  }  **public** **static** **void** executeCustomRunnable()  {  System.***out***.println(*currentName* + " - executeCustomRunnable started - ");  Thread customThread=**new** Thread(**new** CustomRunnable());  customThread.start();  System.***out***.println(*currentName* + " -executeCustomRunnable finished ---");  }  } |
| Пример 2. Имплементация интерфейса Runnable для создания собственного потока |

Каждый процесс имеет хотя бы один выполняющийся поток. Тот поток, с которого начинается выполнение программы, называется главным (main). В языке Java, после создания процесса, выполнение главного потока начинается с метода main(). Затем, по мере необходимости, в заданных программистом местах, и при выполнении заданных им же условий, запускаются другие, побочные потоки.

# Жизненный цикл потока

1. ***Создание потока.*** Создание объекта типа java.lang.Thread:   
   Thread thread = new Thread().
2. ***Запуск потока.*** Вызов на объекте типа Thread метода start(): thread.start()
3. ***Выполнение потока.*** Исполнение потока начинается при входе в метод run(), т.е: start() вызывает метод run() потока и выставляет флаг alive=true.
4. ***Завершение потока.*** «Штатное» завершение метода run() или проброс исключения
5. Невозможность запуска потока снова после его завершения: вызов метода start() на объекте потока после предшествующего отработанного раза приводит к возникновению исключения.

Каждый поток в своем жизненном цикле проходит следующие состояния:

* ***New***: поток был создан, но не был запущен (метод Thread.start() не вызван).
* ***Runnable***: поток был запущен, но еще не получил процессорного времени, либо он находится в порядке очереди на исполнение, или же отдал свое процессорное время другому, и ожидает следуюшей порции.
* ***Running***: поток в текуший момент выполняеться на процессоре.
* ***Blocked***: состояние, в которое поток перешел в момент получения доступа к обьекту (войти в synchronized-блок/метод), который занят другим потоком. Поток продолжит выполнение, как только обьект будет освобожден занимавшим его потоком.
* ***Waiting*:** на потоке был вызван метод wait/join, иначе говоря, поток в состоянии ожидания. При переводе потока в состояние ожидания с помощью wait() поток не может сам выйти из него. В случае с join(), поток выйдет из этого состояния по окончании выполнения другого потока.
* ***Timed*\_*waiting*:** поток в состоянии ожидания (wait(timeout)/join(timeout), sleep()), но, присутствует таймаут, который может раньше вывести поток из этого состояния, если ничего не произойдет, т.е. никто не вызовет метод notify(), либо поток которого ожидают не закончит свое выполнение, либо не будет вызван interrupt().
* ***Terminated*:** поток закончил свое выполнение.

# Методы класса Thread

В Java процесс завершается тогда, когда завершается последний его поток.

*(!) Даже если метод* ***main()*** *уже завершился, но еще выполняются порожденные им потоки, система будет ждать их завершения.*

Однако это правило не относится к особому виду потоков – «daemon»-потокам. Если завершился последний обычный поток процесса, и остались только потоки-демоны, то они будут принудительно завершены и выполнение процесса закончится. Объявить поток daemon можно, если перед запуском потока вызвать его метод setDaemon(true).

На экземпляре типа Thread можно вызвать:

* ***sleep()*** усыпляет поток. Важно понимать, что поток при этом не освобождает занятые им ресурсы.
* ***join()*** приостанавливает выполнение текущего потока, пока не выполниться другой. Не отпускает ресурс.
* ***wait()*** усыпляет поток до наступления какого-либо события. При этом поток освобождает занятые им ресурсы. Может быть вызван только внутри synchronized блока.
* ***yield()*** заставляет процессор переключиться на обработку других потоков системы.
* ***notify()*** метод «пробуждения» потока. Если существует несколько потоков, находящихся в состоянии wait, проснётся только один, при этом, невозможно опеределить, какой именно будет «разбужен».
* ***notifyAll()*** метод «пробуждения» всех «спящих» потоков. Если существует несоклько потоков, находящихся в состоянии ожидания, то пробудятся все, но ресурс захватит только один из них, остальные же выполняться после освобождения ресурса.

Когда между потоками возникает конкуретный доступ на чтение или запись к некоторым общим ресурсам, возникает необходимость управления этим доступ и обеспечить его синхронизированность. Java поддерживает два подхода к синхронизации:

* взаимного исключения (mutual exclusion);
* кооперации (cooperation).

Метод взаимного исключения реализуется синхронизацией по объекту – работа с этоим объектом взаимно исключается для нескольких потоков, и только один из них может выполняться, пока другие ожидают освобождения объекта. Синхронизация по объекту достигается благодаря synchronized-секциям: ***synchronized(mutual object)***.

Кооперация (комбинированный метод синхронизации) достигается достигаеться использованием ***synchronized(mutual object) + wait()/notify()***. Коопеарция позволяет потокам сотрудничать при работе с общими ресурсами.

# Ключевое слово synchronized

***synchronized*** – ключевое слово, используемое для методов и блоков (см. ниже) для обеспечения синхронизированного доступа к выделенному блоку кода и обеспечивает невозможность выполения этого участка кода несколькими параллельными потоками.

1. Синхронизованный метод:

public **synchronized** void doSomething () { // ... }

1. Синхронизованный блок внутри метода:

public void doSomething () {

**synchronized** ( obj )

{ // ... }

}

Таким образом, сихронизация применима:

* К блокам кода при помощи «захвата» монитора указанного объекта.
* К методам по монитору текущего объекта (this).
* К статическим методам по монитору класса (ИмяКласса.class).

|  |
| --- |
| **public** **class** Data  {  **public** **int** value=**0**;  } |
| **public** **class** DataPrinterRunnable **implements** Runnable  {  **private** Object syncMonitor;  **private** Data dataToPrint;  **public** DataPrinterRunnable(Object sync, Data data)  {  syncMonitor=sync;  dataToPrint=data;  }  **public** **void** run()  {  System.***out***.println("DataPrinterRunnable.run: run started");    **synchronized** (syncMonitor)  {  // condition to force the thread be waiting  **if** (dataToPrint.value == **0**) {  **try**  {  System.***out***.println("DataPrinterRunnable.run: Waiting for the data to print...");  syncMonitor.wait(); // waiting for the data  System.***out***.println("DataPrinterRunnable.run: Data is coming... Running again!");  }  **catch** (InterruptedException ex)  {  System.***err***.println("DataPrinterRunnable.run: Interrupted: " + ex.getMessage());  }  }    // data has been gotten: it's time to print it!  System.***out***.println("DataPrinterRunnable.run: dataToPrint = " + dataToPrint.value);  }  } |
| **public** **class** SyncDemo  {  **public** **static** **void** main(String[] args)  {  // common resource between threads to be like a semaphore  Object syncMonitor=**new** Object();  // shared data between threads  Data data=**new** Data();  // thread executor  Thread threadExecutor=**new** Thread(**new** DataPrinterRunnable(syncMonitor, data));  threadExecutor.start();  **try**  {  System.***out***.println("SyncDemo.main: the main thread is going to sleep for 500 ms...");  Thread.*sleep*(**500**);  }  **catch** (InterruptedException ex)  {  System.***err***.println("SyncDemo.main: Interrupted: " + ex.getMessage());  }  // change the data.value in the main thread and notify the waiting thread to print it  **synchronized** (syncMonitor)  {  System.***out***.println("SyncDemo.main: set data.value to 1");  data.value=**1**;  System.***out***.println("SyncDemo.main: notifying the waiting thread to wake up...");  syncMonitor.notify();  System.***out***.println("SyncDemo.main: the waiting thread is woked up!");  }  }  } |
|  |
| Пример 3. Организация синхронизированного доступа к объекту, использую механизм синхронизации |
|  |