**чПараллельное программирование.**

**1)Основы:**

Если запустить эту программу на выполнение, мяч будет скакать почти идеально, но программа будет полностью занята управлением его движением. Если вам надоест наблюдать за скачущим мячом до того, как он выполнит 1000 движений, и вы щелкнете на кнопке Close (Закрыть), мяч все равно продолжит скакать. Вы не сможете взаимодействовать с программой до тех пор, пока мяч не перестанет скакать. Очевидно, что такое поведение данной программы никуда не годится. Оно совершенно не подходит для программ, рассчитанных на более или менее длительную работу. Очевидно, что было бы неплохо иметь возмож ность щелчком на кнопке Stop (Останов) или Back (Назад) прервать процесс загрузки. Чуть ниже мы покажем, как предоставить пользователю контроль над таким процессом, запуская критические разделы кода в отдельном потоке исполнения. Ниже приведена простая процедура для исполнения задачи в отдельном потоке :

* Введите код выполняемой задачи в тело метода run () из класса, реализующего интерфейс Runnable. Этот интерфейс очень прост и содержит следующий единственный метод:

public interface Runnable { void run(); }

* Интерфейс Runnable является функциональным, и поэтому его экземпляр можно создать с помощью лямбда-выражения следующим образом:

Runnable г = () -> { код задачи };

* Сконструируйте объект типа Threa d из объекта г типа Runnable, как показано ниже.

Thread t = new Thread(г);

* Запустите поток на исполнение следующим образом:

t .start ();

**2)Прерывание потока:**

Поток можно прервать принудительно только с помощью метода stop ( ) , не рекомендованного к применению. Но для запроса на прерывание потока исполнения может быть вызван метод interrupt (). Когда метод interrupt () вызывается для потока исполнения, устанавливается состояние прерывания данного потока. Это признак типа boolean, имеющийся у каждого потока исполнения. В каждом потоке исполнения следует периодически проверять значение данного признака, чтобы знать, когда поток должен быть прерван. Чтобы выяснить, было ли установлено состояние прерывания потока исполнения, вызывается статический метод Thread.currentThread(), получающий текущий поток исполнения и вызывающий далее метод isInterrupted (), как показано ниже.

* while ( !Thread.currentThread().islnterrupted() && дополнительные действия) {

выполнить дополнительные действия

}

**3)**

Но если поток исполнения заблокирован, то проверить состояние его прерывания нельзя. И здесь на помощь приходит исклю чение типа InterruptedException. Когда метод interrupt () вызывается для потока исполнения, который заблокирован, например, в результате вызова метода sleep () или wait ( ) , блокирующий вызов прерывается исключением типа InterruptedException.

* Runnable г = () -> {

while (!Thread.currentThread().islnterrupted() && дополнительные действия) {

выполнить дополнительные действия }

} catch ( InterruptedException е ) {

// поток прерван во время ожидания или приостановки

} finally {

выполнить очистку, если требуется

}

// выходом из метода run() завершается исполнение потока };

**4)** Можно найти массу опубликованного кода, где прерывание типа InterruptedException подавляется на низком уровне, He поступайте так! Если вы не можете придумать ничего полезного из того, что можно было бы сделать в блоке catch, вам остаются на выбор два обоснованных варианта.

• Сделать в блоке catch вызов Thread. currentThread () .interrupt (), чтобы установить состояние прерывания потока исполнения, как выделено ниже полужирным. И тогда это состояние может быть проверено в вызывающей части программы.

void mySubTaskO {

try {

sleep(delay);

} catch (InterruptedException e) {

Thread.currentThread() .interrupt() ;

} }

* А еще лучше указать выражение throws InterruptedException в сигнатуре метода, как выделено ниже, а также удалить блок try из его тела. И тогда это прерывание может быть перехвачено в вызывающей части программы, а в крайнем случае — в методе run ().

void mySubTask() throws InterruptedException {

sleep(delay);

}

**5) Потоки могут находиться в одном из шести состояний:**

• новый;

• исполняемый;

• блокированный;

• ожидающий;

• временно ожидающий;

• завершенный.

Каждое из этих состояний поясняется в последующих разделах. Чтобы определить текущее состояние потока исполнения, достаточно вызвать метод getState( ) .

**6) Новые потоки исполнения :**

Если поток исполнения создан в результате операции new, например new Thread (г) , то он еще не запущен на выполнение. Это означает, что он находится в новом состоянии и что программа еще не запустила на исполнение код в данном потоке.

**7) Исполняемые потоки:**

Как только вызывается метод start( ) , поток оказывается в исполняемом состоянии. Исполняемый поток может выполняться или не выполняться в данный момент, поскольку от операционной системы зависит, будет ли выделено потоку время на исполнение. Если поток запущен, он не обязательно продолжает исполняться. На самом деле даже желательно, чтобы исполняемый поток периодически приостанавливался, давая возможность выполниться другим потокам.

**8) Блокированные и ожидающие потоки исполнения:**

Когда поток исполнения находится в состоянии блокировки или ожидания, он временно не активен. Он не выполняет никакого кода и потребляет минимум ресурсов. На планировщике потоков лежит обязанность повторно активизировать его.Подробности зависят от того, как было достигнуто неактивное состояние.

**9) Завершенные потоки исполнения :**

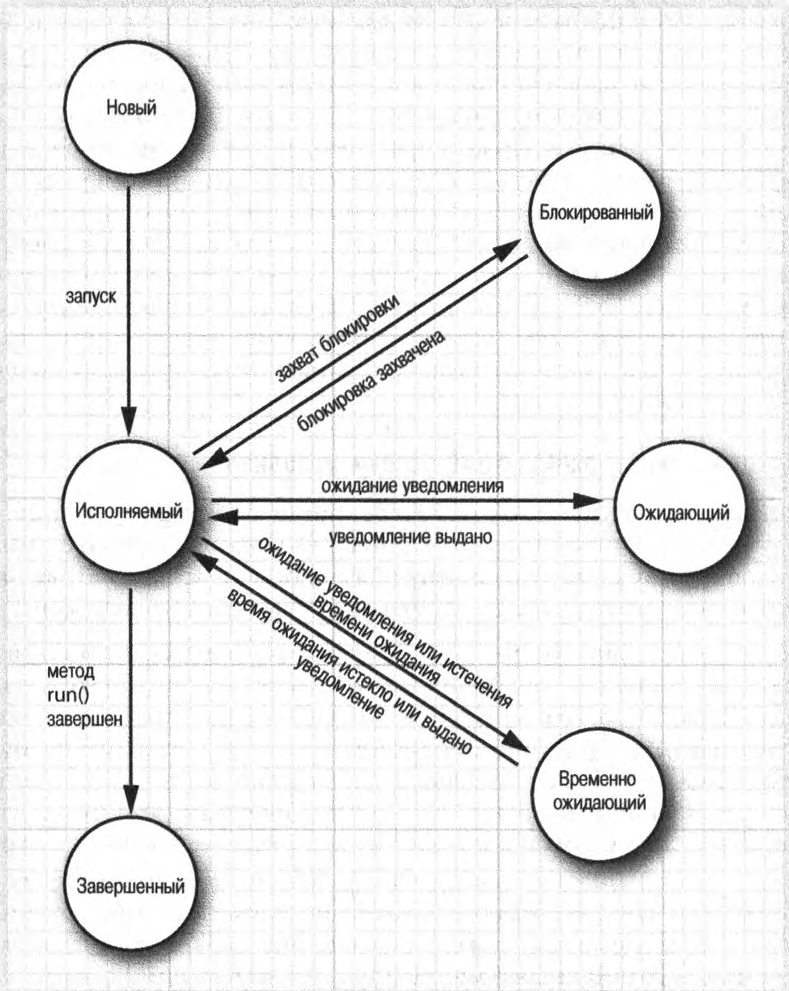
Поток исполнения завершается по одной из следующих причин.

• Прекращает свое существование естественным образом при нормальном завершении метода run ().

• Прекращает свое существование внезапно, поскольку неперехваченное исключение прерывает выполнение метода run ().

В частности, поток исполнения можно уничтож ить, вызвав его метод stop ( ) . Этот метод генерирует объект ошибки типа ThreadDeath, который уничтожает поток исполнения. Но метод stop () не рекомендован к применению, поэтому следует избегать его применения в прикладном коде.

**10) Общая картина потоков:**



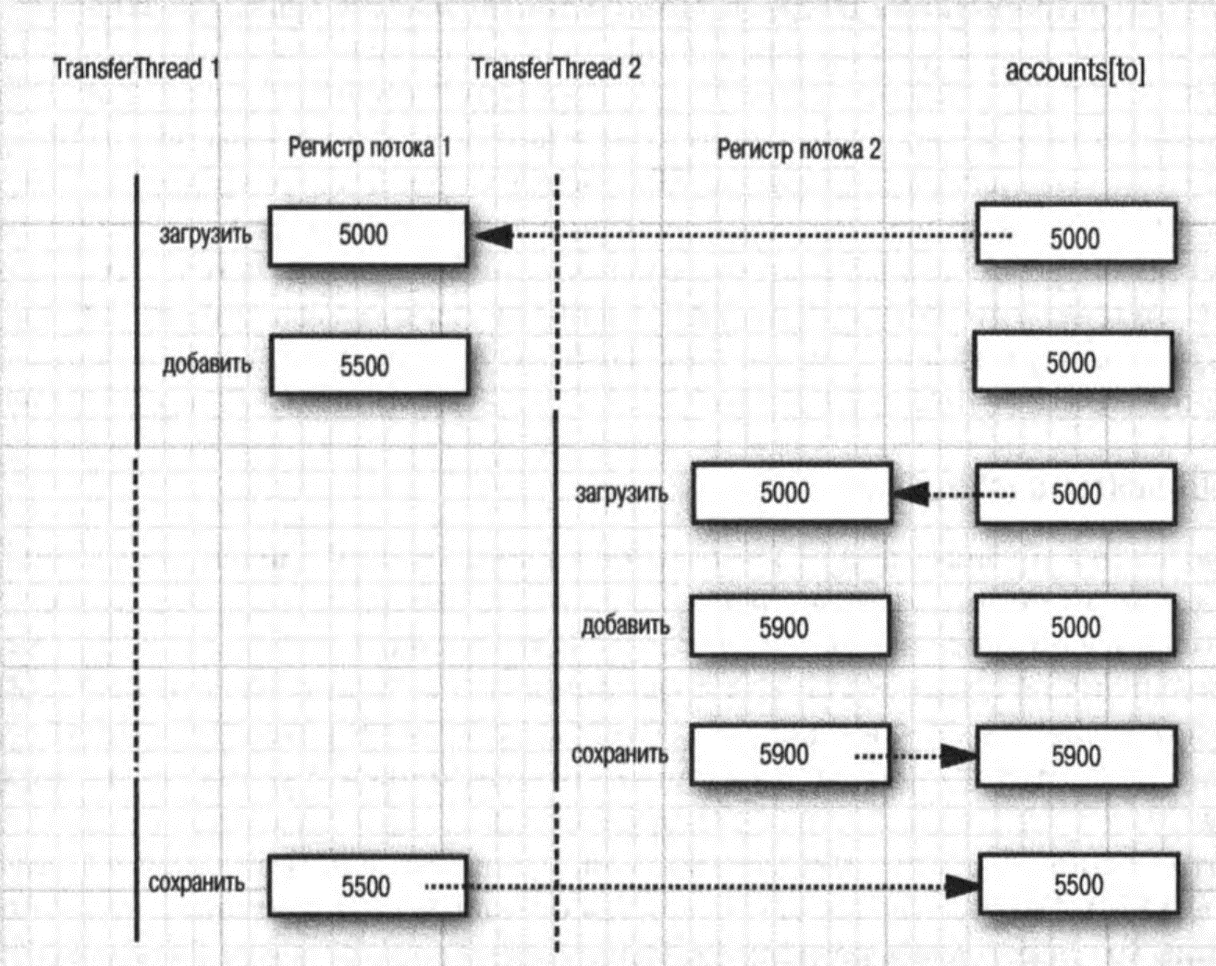
**11)** В языке программирования Java каждый поток исполнения имеет свой приоритет. По умолчанию поток исполнения наследует приоритет того потока, который его создал. Повысить или понизить приоритет любого потока исполнения можно, вызвав метод setPriority(). А установить приоритет потока исполнения можно, указав любое значение в пределах от MIN PRIORITY (определено в классе Thread равным 1) до MAX PRIORITY (равно 10). Обычному приоритету соответствует значение NORM\_PRIORITY, равное 5.

**12) Потоковые демоны:**  
Превратить поток исполнения в потоковый демон можно, сделав следующий вызов:

t .setDaemon(true);

Правда, в таком потоке исполнения нет ничего демонического. Демон — это лишь поток, у которого нет других целей, кроме служения другим. В качестве примера можно привести потоки исполнения таймера, посылающие регулярные "та к т ы " другим потокам, или же потоки исполнения, очищающие устаревшие записи в кеше. Когда остаются только потоковые демоны, виртуальная машина завершает работу.Начинающие программисты, которые не хотят думать о действиях, завершающих программу, иногда ошибочно используют потоковые демоны. Но такой подход не безопасен. Потоковый демон вообще не должен обращаться к такому постоянному ресурсу, как файл или база данных, поскольку он может быть прерван в любой момент, даже посередине операции.

**13)** В большинстве практических многопоточных приложений двум или более потокам исполнения приходится разделять общий доступ к одним и тем же данным. Что произойдет, если два потока исполнения имеют доступ к одному объекту и каждый из них вызывает метод, изменяющий состояние этого объекта? Нетрудно догадаться, что потоки исполнения станут наступать друг другу на пятки. В зависимости от порядка обращения к данным можно в конечном итоге получить поврежденный объект. Такая ситуация обычно называется состоянием гонок. Пример возможной ошибки:



**14) ReentrantLock:**

В языке Java для этой цели предусмотрено ключевое слово synchronized, а в версии Java SE 5.0 появился еще и класс ReentrantLock. Защита блока кода средствами класса ReentrantLock в общих чертах выглядит следующим образом:

myLock.lock(); // объект типа ReentrantLock

try {

критический раздел кода

} finally {

myLock.unlock(); // непременно снять блокировку, даже если генерируется исключение

}

Такая конструкция гарантирует, что только один поток исполнения в единицу времени сможет войти в критический раздел кода. Как только один поток исполнения заблокирует объект блокировки, никакой другой поток не сможет обойти вызов метода lock( ). И если другие потоки исполнения попытаются вызвать метод lock(),то они будут деактивизированы до тех пор, пока первый поток не снимет блокировку с объекта блокировки.

**15) Методы wait и notify:**

Иногда при взаимодействии потоков встает вопрос о извещении одних потоков о действиях других. Например, действия одного потока зависят от результата действий другого потока, и надо как-то известить один поток, что второй поток произвел некую работу. И для подобных ситуаций у класса Object определено ряд методов:

* **wait()**: освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()
* **notify()**: продолжает работу потока, у которого ранее был вызван метод wait()
* **notifyAll()**: возобновляет работу всех потоков, у которых ранее был вызван метод wait()

Все эти методы вызываются только из синхронизированного контекста - синхронизированного блока или метода.

**16) Условия в блоках:**

Применение объектов Condition во многом аналогично использованию методов wait/notify/notifyAll класса Object, которые были рассмотрены в одной из прошлых тем. В частности, мы можем использовать следующие методы интерфейса Condition:

* **await**: поток ожидает, пока не будет выполнено некоторое условие и пока другой поток не вызовет методы signal/signalAll. Во многом аналогичен методу wait класса Object
* **signal**: сигнализирует, что поток, у которого ранее был вызван метод await(), может продолжить работу. Применение аналогично использованию методу notify класса Object
* **signalAll:** сигнализирует всем потокам, у которых ранее был вызван метод await(), что они могут продолжить работу. Аналогичен методу notifyAll() класса Object

Эти методы вызываются из блока кода, который попадает под действие блокировки ReentrantLock. Сначала, используя эту блокировку, нам надо получить объект Condition:

ReentrantLock locker = new ReentrantLock();

Condition condition = locker.newCondition();

Как правило, сначала проверяется условие доступа. Если соблюдается условие, то поток ожидает, пока условие не изменится:

while (условие) {

condition.await();

}

После выполнения всех действий другим потокам подается сигнал об изменении условия:

condition.signalAll();

**17) synchronized**

Если метод объявлен с ключевым словом synchronized, то блокировка объекта защищает весь этот метод. Следовательно, поток исполнения должен захватить встроенную блокировку объекта, чтобы вызвать такой метод. Иными словами, следующий фрагмент кода:

public synchronized void method() {

тело метода

}

равнозначен приведенному ниже фрагменту кода:

public void method() {

this.intrinsicLock.lock();

try {

тело метода

} finally {

this.intrinsicLock.unlock();

} } }

**18) Какой же блокиратор использовать?**

Что же лучше использовать в прикладном коде: объекты типа Lock и Condition или синхронизированные методы? Ниже приведены некоторые рекомендации, которые дают ответ на этот вопрос.

* Лучше не пользоваться ни объектами типа Lock/Condition, ни ключевым словом synchronized. Зачастую вместо этого можно выбрать подходящий механизм из пакета java.util.concurrent, который организует блокировку автоматически. Так, в разделе 14.6 далее в этой главе будет показано, как пользоваться блокирующими очередями для синхронизации потоков, выполняющих общую задачу.
* Если ключевое слово synchronized подходит в конкретной ситуации, непременно воспользуйтесь им. В этом случае вам придется написать меньше кода, а следовательно, допустить меньше ошибок. В листинге 14.8 приведен пример очередного варианта программы, имитирующей банк и реализованной на основе синхронизированных методов.
* Пользуйтесь объектами типа Lock/Condition, если действительно нуждаетесь в дополнительных возможностях подобных конструкций.

**19) Мониторы:**

В терминологии Java монитор обладает следующими свойствами:

* Монитор — это класс, имеющий только закрытые поля.
* У каждого объекта такого класса имеется связанная с ним блокировка.
* Все методы блокируются этой блокировкой. Иными словами, если клиент вызывает метод obj .method( ) , блокировка объекта obj автоматически захватывается в начале этого метода и снимается по его завершении. А поскольку все поля класса монитора закрытые, то такой подход гарантирует, что к ним нельзя будет обратиться ни в одном из потоков исполнения до тех пор, пока ими манипулирует какой-то другой поток.
* У блокировки может быть любое количество связанных с ней условий.

Создатели Java вольно адаптировали принцип монитора. Каждый объект в Java обладает встроенной блокировкой и встроенным условием. Если метод объявлен с ключевым словом synchronized, он действует как метод монитора. А переменная условия доступна через вызовы методов wait() ,notifyAll() ,notify() .

Но объекты в Java отличаются от мониторов в следующих трех важных отношениях, нарушающих безопасность потоков исполнения.

* Поля не обязательно должны быть закрытыми (private).
* Методы не обязаны быть синхронизированными (synchronized).
* Встроенная блокировка доступна клиентам.

Это — явное пренебрежение требованиями безопасности, изложенными Пером Бринчем Хансеном.

**20)** **Volatile:**

Ключевое слово volatile обозначает неблокирующий механизм синхронизированного доступа к полю экземпляра. Если поле объявляется как volatile, то компилятор и виртуальная машина принимают во внимание тот факт, что поле может быть параллельно обновлено в другом потоке исполнения.

**21) Stop:**  
Обратимся сначала к методу stop (). Он прекращает выполнение любых незавершенных методов, включая и run() . Когда поток исполнения останавливается, данный метод немедленно снимает блокировки со всех объектов, которые он блокировал. Это может привести к тому, что объекты останутся в несогласованном состоянии. Д оп устим, метод TransferThread() остановлен посредине перевода денежных средств с одного счета на другой: после снятия денежных средств с одного счета, но перед их переносом на другой счет. В этом случае объект имитируемого банка оказывается поврежденным. А поскольку блокировка снята, этот поврежденный объект будет доступен и другим потокам исполнения, которые не были остановлены.

**22) Блокирующие очереди: ?????**