Национальный Открытый Университет "ИНТУИТ": www.intuit.ru
Вадим Монахов
Лекция 9. Дополнительные элементы объектного программирования на языке Java

9.1. Потоки выполнения (threads) и синхронизация

В многозадачных операционных системах (MS *Windows*, Linux и др.) программу, выполняющуюся под управлением операционной системы (ОС), принято называть приложением операционной системы (application), либо, что то же, процессом (process). Обычно в ОС паралельно (или псевдопаралельно, в режиме разделения процессорного времени) выполняется большое число процессов. Для выполнения процесса на аппаратном уровне поддерживается независимое от других процессов *виртуальное адресное пространство*. Попытки процесса выйти за пределы адресов этого пространства отслеживаются аппаратно.

Такая модель удобна для разграничения независимых программ. Однако во многих случаях она не подходит, и приходится использовать *подпроцессы* (subprocesses), или, более употребительное название, threads . Дословный перевод слова threads - "нити". Иногда их называют легковесными процессами (lightweight processes), так как при прочих равных условиях они потребляют гораздо меньше ресурсов, чем процессы. Мы будем употреблять термин "потоки выполнения", поскольку термин *multithreading* - работу в условиях существования нескольких потоков,- на русский язык гораздо лучше переводится как *многопоточность*. Следует соблюдать аккуратность, чтобы не путать threads с *потоками ввода-вывода* (streams).

Потоки выполнения отличаются от процессов тем, что находятся в адресном пространстве своего *родительского процесса*. Они выполняются параллельно (псевдопараллельно), но, в отличие от процессов, легко могут обмениваться данными в пределах общего виртуального адресного пространства. То есть у них могут иметься общие переменные, в том числе - массивы и объекты.

В приложении всегда имеется главный (основной) *поток* выполнения. Если он закрывается - закрываются все остальные пользовательские потоки приложения. Кроме них возможно создание потоков-демонов (*daemons*), которые могут продолжать работу и после окончания работы главного потока выполнения.

Любая *программа Java* неявно использует потоки выполнения. В главном потоке виртуальная *Java*-машина (*JVM*) запускает метод main приложения, а также все методы, вызываемые из него. Главному потоку автоматически дается имя "main" . Кроме главного потока в фоновом режиме (с малым приоритетом) запускается дочерний *поток*, занимающийся *сборкой мусора*. Виртуальная *Java*-машина автоматически стартует при запуске на компьютере хотя бы одного приложения *Java*, и завершает работу в случае, когда у нее на выполнении остаются только потоки-демоны.

В *Java* каждый *поток* выполнения рассматривается как *объект*. Но интересно то, что в *Java* каждый *объект*, даже не имеющий никакого отношения к классу Thread , может работать в условиях многопоточности, поскольку в классе Object определены методы объектов, предназначенные для взаимодействия объектов в таких условиях. Это notify() , notifyAll() , wait() , wait(*timeout*) - "оповестить", "оповестить всех","ждать", "ждать до истечения таймаута". Об этих методах будет рассказано далее.

Преимущества и проблемы при работе с потоками выполнения

Почему бывают нужны потоки выполнения? Представьте себе программу управления спектрометром, в которой одно табло должно показывать время, прошедшее с начала измерений, второе - число импульсов со счетчика установки, третье - длину волны, для которой в данный момент идут измерения. Кроме того, в фоновом режиме должен отрисовываться получающийся после обработки данных спектр. Возможны две идеологии работы программы - последовательная и параллельная. При последовательном подходе во время выполнения алгоритмов, связанных с показом информации на экране, следует время от времени проверять, не пришли ли новые импульсы со счетчиков, и не надо ли установить спектрометр на очередную длину волны. Кроме того, во время обработки данных и отрисовки спектра следует через определенные промежутки обновлять табло времени и счетчиков.

В результате код программы перестает быть структурным. Нарушается принцип инкапсуляции - "независимые вещи должны быть независимы". Независимые по логике решаемой проблемы алгоритмы оказываются перемешаны друг с другом. Такой код оказывается ненадежным и плохо *модифицируемым*. Однако он относительно легко отлаживается, поскольку последовательность выполнения операторов программы однозначно определена.

В параллельном варианте для каждого из независимых алгоритмов запускается свой поток выполнения, и ему задается необходимый приоритет. В нашем случае один (или более) поток занимается измерениями. Второй - показывает время, прошедшее с начала измерений, третий - число импульсов со счетчика, четвертый - длину волны. Пятый поток рисует спектр. Каждый из них занят своим делом и не вмешивается в другой. Связь между потоками идет только через данные, которые первый поток поставляет остальным.

Несмотря на изящество параллельной модели, у нее имеются очень неприятные недостатки.

Во-первых, негарантированное время отклика. Например, при использовании потоков для анимации графических объектов может наблюдаться сбой равномерности движения объекта.

Во-вторых, отладка программ, работающих с использованием параллелизма, с помощью традиционных средств практически невозможна. Если при каком-то сочетании условий по времени происходит ошибка, воспроизвести ее обычно оказывается невозможно: воссоздать те же временные интервалы крайне сложно. Поэтому требуется применять специальные технологии с регистрацией всех потоков данных в файлы с отладочной информацией. Для этих целей, а также для нахождения участков кода, вызывающих наибольшую трату времени и ресурсов во время выполнения приложения, используются специальные программы – *профилировщики* (profilers).

Синхронизация по ресурсам и событиям

Использование потоков выполнения порождает целый ряд проблем не только в плане отладки программ, но и при организации взаимодействия между разными потоками.

Синхронизация по ресурсам

Если разные потоки получают доступ к одним и тем же данным, причем один из них или они оба меняют эти данные, для них требуется обеспечить\установить *разграничение доступа*. Пока один поток меняет данные, второй не должен иметь права их читать или менять. Он должен дожидаться окончания доступа к данным первого потока. Говорят, что осуществляется *синхронизация* потоков. В Java для этих целей служит оператор `synchronize` ("синхронизировать"). Такой тип синхронизации называется *синхронизацией* по ресурсам и обеспечивает блокировку данных на то время, которое необходимо потоку для выполнения тех или иных действий. В Java такого рода блокировка осуществляется на основе концепции *мониторов* и в применении к Java заключается в следующем. Под монитором понимается некая *управляющая конструкция*, обеспечивающая монопольный доступ к объекту. Если во время выполнения синхронизованного метода объекта другой поток попробует обратиться к методам или данным этого объекта, он будет блокирован до тех пор, пока не закончится выполнение синхронизованного метода. При запуске синхронизованного метода говорят, что объект *входит в монитор*, при завершении – что объект *выходит из монитора*. При этом поток, внутри которого вызван синхронизованный метод, считается владельцем данного монитора.

Имеется два способа синхронизации по ресурсам: синхронизация объекта и синхронизация метода.

Синхронизация объекта `obj1` (его иногда называют *объектом действия*) осуществляется следующим образом:

```
synchronized(obj1) оператор;
```

Например:

```
synchronized(obj1){
   ...
   m1(obj1);
   ...
   obj1.m2();
   ...
}
```

В данном случае в качестве синхронизованного оператора выступает участок кода в фигурных скобках. Во время выполнения этого участка доступ к объекту `obj1` блокируется для всех других потоков. Это означает, что пока будет выполняться вызов оператора, выполнение вызова любого синхронизованного метода или синхронизованного оператора для этого объекта будет приостановлено до окончания работы оператора.

Данный способ синхронизации обычно используется для экземпляров классов, разработанных без расчета на работу в режиме многопоточности.

Второй способ синхронизации по ресурсам используется при разработке класса, рассчитанного на взаимодействия в многопоточной среде. При этом методы, критичные к *атомарности операций* с данными (обычно – требующие согласованное изменение нескольких полей данных), объявляются как синхронизованные с помощью модификатора `synchronized` :

```
public class ИмяКласса{
   ...
   public synchronized тип метод(...){
```

```
            ...
        }
     }
```

Вызов данного метода из объекта приведет к вхождению данного объекта в монитор.

Пример:

```
public class C1{
   public synchronized void m1(){
   }
}
C1 obj1=new C1();
obj1.m1();
```

Пока будет выполняться вызов `obj1.m1()`, доступ из других потоков к объекту `obj1` будет блокирован – выполнение вызова любого синхронизованного метода или синхронизованного оператора для этого объекта будет приостановлено до окончания работы метода `m1()`.

Если синхронизованный метод является не методом объекта, а методом класса, при вызове метода в монитор входит класс, и приостановка до окончания работы метода будет относиться ко всем вызовам синхронизованных методов данного класса.

Итак, возможна синхронизация как целого метода (но только при задании его реализации в классе), так и отдельных операторов. Иногда синхронизованную область кода (метод или оператор) называют *критической секцией* кода.

Синхронизация по событиям

Кроме синхронизации по данным имеется *синхронизация по событиям*, когда параллельно выполняющиеся потоки приостанавливаются вплоть до наступления некоторого события, о котором им сигнализирует другой поток. Основными операциями при таком типе синхронизации являются `wait` ("ждать") и `notify` ("оповестить").

В Java синхронизацию по событиям обеспечивают следующие методы, заданные в классе `Object` и наследуемые всеми остальными классами:

- `void wait()` – поток, внутри которого какой-либо объект вызвал данный метод (владелец монитора), переводится в состояние ожидания. Поток приостанавливает работу своего метода `run()` вплоть до поступления объекту, вызвавшему приостановку ("засыпание") потока уведомления `notify()` или `notifyAll()`. При неправильной попытке "разбудить" поток соответствующий код компилируется, но при запуске вызывает появление исключения `llegalMonitorStateException`.
- `void wait(long millis)` – то же, но ожидание длится не более `millis` миллисекунд.
- `void wait(long millis, int nanos)` – то же, но ожидание длится не более `millis` миллисекунд и `nanos` наносекунд.
- `void notify()` – оповещение, приводящее к возобновлению работы потока, ожидающего выхода данного объекта из монитора. Если таких потоков несколько, выбирается один из них. Какой – зависит от реализации системы.
- `void notifyAll()` – оповещение, приводящее к возобновлению работы всех потоков, ожидающих выхода данного объекта из монитора.

Метод `wait` для любого объекта `obj` следует использовать следующим образом – необходимо организовать цикл `while`, в котором следует выполнять оператор `wait`:

```
synchronized(obj){
   while(not условие)
     obj.wait();
   …//выполнение операторов после того, как условие стало true
}
```

При этом не следует беспокоиться, что цикл `while` постоянно крутится и занимает много ресурсов процессора. Этого не происходит: после вызова `obj.wait()` поток, в котором находится указанный код, "засыпает" и перестает занимать ресурсы процессора. При этом метод `wait` на время "сна" потока снимает блокировку с объекта `obj`, задаваемую оператором `synchronized(obj)`. Что позволяет другим потокам обращаться к объекту с вызовом `obj.notify()` или `obj.notifyAll()`.

Класс Thread и интерфейс Runnable. Создание и запуск потока выполнения

Имеется два способа создать класс, экземплярами которого будут потоки выполнения: унаследовать класс от `java.lang.Thread` либо реализовать интерфейс `java.lang.Runnable`. Этот интерфейс имеет декларацию единственного метода `public void run()`, который обеспечивает последовательность действий при работе потока. При этом класс `Thread` уже реализует интерфейс `Runnable`, но с пустой реализацией метода `run()`.Так что при создании экземпляра `Thread` создается поток, который ничего не делает. Поэтому в потомке надо переопределить метод `run()`. В нем следует написать реализацию алгоритмов, которые должны выполняться в данном потоке. Отметим, что после выполнения метода `run()` поток прекращает существование - "умирает".

Рассмотрим первый вариант, когда мы наследуем класс от класса `Thread`, переопределив метод `run()`.

Объект-поток создается с помощью конструктора. Имеется несколько перегруженных вариантов конструкторов, самый простой из них - с пустым списком параметров. Например, в классе `Thread` их заголовки выглядят так:

- `public Thread()` - конструктор по умолчанию. *Подпроцесс* получает имя `"system"`.
- `public Thread(String name)` - поток получает имя, содержащееся в строке `name`.

Также имеется возможность создавать потоки в группах. Но в связи с тем, что данная технология устарела и не нашла широкого распространения, о группах потоков выполнения в данном учебном пособии рассказываться не будет.

В классе-потомке можно вызывать конструктор по умолчанию (без параметров), либо задать свои конструкторы, используя вызовы прародительских с помощью вызова `super ( список параметров )`. Из-за отсутствия наследования конструкторов в Java приходится в наследнике заново задавать конструкторы с той же сигнатурой, что и в классе `Thread`. Это является простой, но утомительной работой. Именно поэтому обычно предпочитают способ задания класса с *реализацией интерфейса* `Runnable`, о чем будет рассказано несколькими строками позже.

Создание и запуск потока осуществляется следующим образом:

```
public class T1 extends Thread{
   public void run(){
     ...
   }
   ...
}

Thread thread1= new T1();
thread1.start();
```

Второй вариант - использование класса, в котором реализован интерфейс `java.lang.Runnable`. Этот интерфейс, как уже говорилось, имеет единственный метод `public void run()`. Реализовав его в классе, можно создать поток с помощью перегруженного варианта конструктора `Thread`:

```
public class R1 implements Runnable{
   public void run(){
     ...
   }
   ...
}

Thread thread1= Thread( new R1() );
thread1.start();
```

Обычно таким способом пользуются гораздо чаще, так как в разрабатываемом классе не приходится заниматься дублированием конструкторов класса `Thread`. Кроме того, этот способ можно применять в случае, когда уже имеется класс, принадлежащий иерархии, в которой базовым классом не является `Thread` или его наследник, и мы хотим использовать этот класс для работы внутри потока. В результате от этого класса мы получаем метод `run()`, в котором реализован нужный алгоритм, и этот метод работает внутри потока типа `Thread`, обеспечивающего необходимое поведение в многопоточной среде. Однако в данном случае затрудняется доступ к методам из класса `Thread` - требуется приведение типа.

Например, чтобы вывести информацию о *приоритете потока*, в первом способе создания потока в методе `run()` надо написать оператор

```
System.out.println("Приоритет потока="+this.getPriority());
```

А во втором способе приходится это делать в несколько этапов. Во-первых, при задании класса нам следует добавить в объекты типа `R1` поле `thread`:

```
public class R1 implements Runnable{
    public Thread thread;

    public void run() {
        System.out.println("Приоритет потока="+thread.getPriority());
    }
}
```

С помощью этого поля мы будем добираться до объекта-потока. Но теперь после создания потока необходимо не забыть установить для этого поля ссылку на созданный объект-поток. Так что создание и запуск потока будет выглядеть так:

```
R1 r1=new R1();
Thread thread1=new Thread(r1, "thread1");
r1.thread=thread1;
thread1.start();//либо, что то же, r1.thread.start()
```

Через поле `thread` мы можем получать доступ к потоку и всем его полям и методам в алгоритме, написанном в методе `run()`. Указанные выше дополнительные действия – это всего три лишних строчки программы (первая – `R1 r1=new R1();` вторая – `r1.thread=thread1;` третья – объявление в классе `R1` – `public Thread thread;`).

Как уже говорилось ранее, напрямую давать доступ к полю данных – дурной тон программирования. Исправить этот недостаток нашей программы просто: в дереве элементов программы окна **Projects** в разделе **Fields** ("поля") щелкнем правой кнопкой мыши по имени **thread** и выберем в появившемся всплывающем меню **Refactor/Encapsulate Fields…** ("Провести рефакторинг"/ "Инкапсулировать поля…"). В появившемся диалоге нажмем на кнопку **"Next>"** и проведем рефакторинг, подтвердив выбор в нижнем окне.

В классе `Thread` имеется несколько перегруженных вариантов конструктора с параметром типа `Runnable`:

- `public Thread(Runnable target)` – с именем `"system"` по умолчанию.
- `public Thread(Runnable target, String name)` – с заданием имени.

Также имеются варианты с заданием группы потоков.

Поля и методы, заданные в классе Thread

В классе `Thread` имеется ряд полей данных и методов, про которые надо знать для работы с потоками.

Важнейшие константы и методы класса Thread:

- `MIN_PRIORITY` – минимально возможный приоритет потоков. Зависит от операционной системы и версии *JVM*. На компьютере автора оказался равен 1.
- `NORM_PRIORITY` – нормальный приоритет потоков. Главный поток создается с нормальным приоритетом, а затем приоритет может быть изменен. На компьютере автора оказался равен 5.
- `MAX_PRIORITY` – максимально возможный приоритет потоков. На компьютере автора оказался равен 10.
- `static int activeCount()` – возвращает число активных потоков приложения.
- `static Thread currentThread()` – возвращает ссылку на текущий поток.
- `boolean holdsLock(Object obj)` – возвращает true в случае, когда какой-либо поток (то есть текущий поток) блокирует объект `obj`.
- `static boolean interrupted()` – возвращает состояние статуса прерывания текущего потока, после чего устанавливает его в значение `false`.

Важнейшие методы объектов типа Thread:

`void run()` – метод, который обеспечивает последовательность действий во время жизни потока. В классе `Thread` задана его пустая реализация, поэтому в классе потока он должен быть переопределен. После выполнения метода `run()` поток умирает.

`void start()` – вызывает выполнение текущего потока, в том числе запуск его метода `run()` в нужном контексте. Может быть вызван всего один раз.

`void setDaemon(boolean on)` – в случае `on==true` устанавливает потоку статус демона, иначе – статус пользовательского потока.

`boolean isDaemon()` – возвращает `true` в случае, когда текущий поток является демоном.

`void yield()` – "поступиться правами" – вызывает временную приостановку потока, с передачей права другим потокам выполнить необходимые им действия.

`long getId()` – возвращает уникальный идентификатор потока. Уникальность относится только ко времени жизни потока – после его завершения (смерти) данный идентификатор может быть присвоен другому создаваемому потоку.

`String getName()` – возвращает имя потока, которое ему было задано при создании или методом `setName`.

`void setName(String name)` – устанавливает новое имя потока.

`int getPriority()` – возвращает *приоритет потока*.

`void setPriority(int newPriority)` – устанавливает *приоритет потока*.

`void checkAccess()` – осуществление проверки из текущего потока на позволительность доступа к другому потоку. Если поток, из которого идет вызов, имеет право на доступ, метод не делает ничего. Иначе – возбуждает исключение `SecurityException`.

`String toString()` – возвращает строковое представление объекта потока, в том числе – его *имя*, *группу*, приоритет.

`void sleep(long millis)` – вызывает приостановку ("засыпание") потока на `millis` миллисекунд. При этом все блокировки (мониторы) потока сохраняются. Перегруженный вариант `sleep(long millis,int nanos)` – параметр *nanos* задает число наносекунд. Досрочное пробуждение осуществляется методом `interrupt()` – с возбуждением исключения `InterruptedException`.

`void interrupt()` – прерывает "сон" потока, вызванный вызовами `wait(…)` или `sleep(…)`, устанавливая ему статус прерванного (статус прерывания=true). При этом возбуждается проверяемая исключительная ситуация `InterruptedException`.

`boolean isInterrupted()` – возвращает текущее состояние статуса прерывания потока без изменения значения статуса.

`void join()` – "слияние". Переводит поток в режим умирания – ожидания завершения (смерти). Это ожидание – выполнение метода `join()` – может длиться сколь угодно долго, если соответствующий поток на момент вызова метода `join()` блокирован. То есть если в нем выполняется синхронизованный метод или он ожидает завершения синхронизованного метода. Перегруженный вариант `join(long millis)` – ожидать завершения потока в течение `millis` миллисекунд. Вызов `join(0)` эквивалентен вызову `join()`. Еще один перегруженный вариант `join(long millis,int nanos)` – параметр *nanos* задает число наносекунд. Ожидание смерти может быть прервано другим потоком с помощью метода `interrupt()` – с возбуждением исключения `InterruptedException`. Метод `join()` является аналогом функции `join` в UNIX. Обычно используется для завершения главным *потоком работы* всех дочерних пользовательских потоков ("слияния" их с главным потоком).

`boolean isAlive()` – возвращает `true` в случае, когда текущий поток жив (не умер). Отметим, что даже если поток завершился, от него остается объект-"призрак", отвечающий на запрос `isAlive()` значением `false` – то есть сообщающий, что объект умер.

Следует отметить, что все ведущие разработчики процессоров перешли на многоядерную технологию. При этом в одном корпусе процессора расположено несколько процессорных ядер, способных независимо выполнять вычисления, но они имеют доступ к одной и той же общей памяти. В связи с этим программирование в многопоточной среде признано наиболее перспективной моделью параллелизации программ и становится одним из важнейших направлений развития программных технологий. Модель многопоточности Java позволяет весьма элегантно реализовать преимущества многоядерных процессорных систем. Во многих случаях программы Java, написанные с использованием многопоточности, эффективно распараллеливаются автоматически на уровне виртуальной машины – без изменения не только исходного, но даже скомпилированного байт-кода приложений.

Но программирование в многопоточной среде является сложным и ответственным занятием, требующим очень высокой квалификации. Многие алгоритмы, кажущиеся простыми, естественными и надежными, в многопоточной среде оказываются неработоспособными. Из-за чего способы решения даже самых простых задач становятся необычными и запутанными, не говоря уж о проблемах, возникающих при решении сложных задач. В связи с этим автор рекомендует на начальном этапе не увлекаться многопоточностью, а только ознакомиться с данной технологией.

Тем, кто все же хочет заняться таким программированием, рекомендуется сначала прочитать главу 9 в книге Джошуа Блоха [8].

9.2. Подключение внешних библиотек DLL."Родные" (native) методы*

*- *данный параграф приводится в ознакомительных целях*

Для *прикладного программирования* средств *Java* в подавляющем большинстве случаев хватает. Однако иногда возникает необходимость подключить к программе ряд системных вызовов. Либо обеспечить *доступ* к библиотекам, написанным на других языках программирования. Для таких целей в *Java* используются методы, объявленные с модификатором *native* - "родной". Это *слово* означает, что при выполнении метода производится вызов "родного" для конкретной платформы двоичного кода, а не платформо-независимого байт-кода как во всех других случаях. Заголовок "родного" метода описывается в классе *Java*, а его реализация осуществляется на каком-либо из языков программирования, позволяющих создавать динамически подключаемые библиотеки (*DLL - Dynamic Link Library* под *Windows*, *Shared Objects* под *UNIX*-образными операционными системами).

Правило для объявления и реализации таких методов носит название *JNI - Java Native Interface*.

Объявление "родного" метода в *Java* имеет вид

Модификаторы native ВозвращаемыйТип имяМетода(список параметров);

Тело "родного" метода не задается - оно является внешним и загружается в *память компьютера* с помощью загрузки той библиотеки, из которой этот метод должен вызываться:

System.loadLibrary("ИмяБиблиотеки");

При этом имя библиотеки задается без пути и без расширения. Например, если под *Windows* библиотека имеет имя `myLib.dll`, или под *UNIX* или Linux имеет имя `myLib.so`, надо указывать `System.loadLibrary("myLib")`;

В случае, если файла не найдено, возбуждается непроверяемая *исключительная ситуация* `UnsatisfiedLinkError`.

Если требуется указать имя библиотеки с путем, применяется вызов

System.load ("ИмяБиблиотекиСПутем");

Который во всем остальном абсолютно аналогичен вызову `loadLibrary`.

После того, как библиотека загружена, с точки зрения использования в программе вызов "родного" метода ничем не отличается от вызова любого другого метода.

Для создания библиотеки с методами, предназначенными для работы в качестве "родных", обычно используется язык C++. В *JDK* существует *утилита* javah.exe, предназначенная для создания заголовков C++ из скомпилированных классов *Java*. Покажем, как ей пользоваться, на примере класса `ClassWithNativeMethod`. Зададим его в пакете нашего приложения:

```
package java_example_pkg;
public class ClassWithNativeMethod {

    /** Creates a new instance of ClassWithNativeMethod */
    public ClassWithNativeMethod() {
    }

  public native void myNativeMethod();
}
```

Для того, чтобы воспользоваться утилитой javah, скомпилируем проект и перейдем в папку **build\classes**. В ней будут располагаться *папка* с пакетом нашего приложения **java_example_pkg** и *папка* META-INF. В режиме командной строки выполним команду

```
javah.exe java_example_pkg.ClassWithNativeMethod
```

- задавать *имя класса* необходимо с полной квалификацией, то есть с указанием перед ним имени пакета. В результате в папке появится *файл* **java_example_pkg_ClassWithNativeMethod.h** со следующим содержимым:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class java_example_pkg_ClassWithNativeMethod */
```

```
#ifndef _Included_java_example_pkg_ClassWithNativeMethod
#define _Included_java_example_pkg_ClassWithNativeMethod
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class: java_example_pkg_ClassWithNativeMethod
 * Method: myNativeMethod
 * Signature: ()V
 */
JNIEXPORT void JNICALL
Java_java_1example_1pkg_ClassWithNativeMethod_myNativeMethod
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Функция `Java_java_1example_1pkg_ClassWithNativeMethod_myNativeMethod(JNIEnv *, jobject)`, написанная на C++, должна обеспечивать реализацию метода `myNativeMethod()` в классе *Java*. *Имя функции* C++ состоит из: префикса *Java*, разделителя "_", модифицированного имени пакета (знаки подчеркивания "_" заменяются на "_1"), разделителя "_", имени класса, разделителя "_", имени "родного" метода. Первый *параметр* `JNIEnv *` в функции C++ обеспечивает *доступ* "родного" кода к параметрам и объектам, передающимся из функции C++ в *Java*. В частности, для доступа к стеку. Второй *параметр*, `jobject`, – *ссылка* на экземпляр класса, в котором задан "родной" метод, для методов объекта, и `jclass` – *ссылка* на сам *класс* – для методов класса. В языке C++ нет ссылок, но в *Java* все переменные *объектного типа* являются ссылками. Соответственно, второй *параметр* отождествляется с этой переменной.

Если в "родном" методе имеются параметры, то *список* параметров функции C++ расширяется. Например, если мы зададим метод

```
public native int myNativeMethod(int i);
```

то *список* параметров функции C++ станет

```
(JNIEnv *, jobject, jint)
```

А тип функции станет `jint` вместо `void`.

Таблица 9.1. Соответствие типов Java и C++

Тип Java	Тип *JNI* (C++)	Характеристика типа *JNI*
boolean	jboolean	1 байт, беззнаковый
byte	jbyte	1 байт
char	jchar	2 байта, беззнаковый
short	jshort	2 байта
int	jint	4 байта
long	jlong	8 байт
float	jfloat	4 байта
double	jdouble	8 байт
void	void	–
Object	jobject	Базовый для остальных классов

Class	jclass	Ссылка на класс Java
String	jstring	Строки Java
массив	jarray	Базовый для классов массивов
Object[]	jobjectArray	Массив объектов
boolean[]	jbooleanArray	Массив булевских значений
byte[]	jbyteArray	Массив байт (знаковых значений длиной в байт)
char[]	jcharArray	Массив кодов символов
short[]	jshortArray	Массив коротких целых
int[]	jintArray	Массив целых
long[]	jlongArray	Массив длинных целых
float[]	jfloatArray	Массив значений float
double[]	jdoubleArray	Массив значений double
Throwable	jthrowable	Обработчик исключительных ситуаций

В реализации метода требуется объявить переменные. Например, если мы будем вычислять квадрат переданного в метод значения и возвращать в качестве результата *значение* параметра, возведенное в квадрат (пример чисто учебный), код реализации функции на C++ будет выглядеть так:

```
#include "java_example_pkg_ClassWithNativeMethod.h"
JNIEXPORT jint JNICALL
Java_java_1example_1pkg_ClassWithNativeMethod_myNativeMethod
  (JNIEnv *env, jobject obj, jint i ){
    return i*i
};
```

Отметим, что при работе со строками и массивами для получения и *передачи параметров* требуется использовать переменную `env`. Например, получение длины целого массива, переданного в переменную `jintArray intArr`, будет выглядеть так:

```
jsize length=(*env)->GetArrayLength(env, intArr);
```

Выделение памяти под переданный *массив*:

```
jint *intArrRef=(*env)->GetIntArrayElements(env, intArr,0);
```

Далее с массивом `intArr` можно работать как с обычным массивом C++. Высвобождение памяти из-под массива:

```
(*env)->ReleaseIntArrayElements(env, intArr, intArrRef ,0);
```

Имеются аналогичные функции для доступа к элементам массивов всех *примитивных типов*: `GetBooleanArrayElements, GetByteArrayElements,…, GetDoubleArrayElements`. Эти функции копируют содержимое массивов *Java* в новую область памяти, с которой и идет работа в C++. Для *массивов объектов* имеется не только *функция* `GetObjectArrayElement`, но и `SetObjectArrayElement` – для получения и изменения отдельных элементов таких массивов.

Строка *Java* jstring s преобразуется в *массив символов* C++ так:

```
const char *sRef=(*env)->GetStringUTFChars(env,s,0);
```

Ее *длина* находится как `int s_len=strlen(sRef)` ;

Высвобождается из памяти как

```
(*env)->ReleaseStringUTFChars(env,s,sRef);
```

Краткие итоги

Программу, выполняющуюся под управлением операционной системы, называют *процессом* (`process`), или, что то же, приложением. У каждого процесса свое адресное пространство. *Потоки выполнения* (`threads`) отличаются от процессов тем, что выполняются в адресном пространстве своего *родительского процесса*. Потоки выполняются параллельно (псевдопараллельно), но, в отличие от процессов, легко могут обмениваться данными в пределах общего виртуального адресного пространства. То есть у них могут иметься общие переменные, в том числе – массивы и объекты.

В приложении всегда имеется главный (основной) поток. Если он закрывается – закрываются все остальные пользовательские потоки приложения. Кроме них возможно создание потоков-демонов, которые могут продолжать работу и после окончания работы главного потока.

Любая программа Java неявно использует потоки. В главном потоке виртуальная Java-машина (*JVM*) запускает метод main приложения, а также все методы, вызываемые из него. Главному потоку автоматически дается имя `"main"`.

Если разные потоки получают доступ к одним и тем же данным, причем один из них или они оба меняют эти данные, для них требуется обеспечить установить *разграничение доступа*. Пока один поток меняет данные, второй не должен иметь права их читать или менять. Он должен дожидаться окончания доступа к данным первого потока. Говорят, что осуществляется *синхронизация* потоков. В Java для этих целей служит оператор `synchronize` ("синхронизировать"). Иногда синхронизованную область кода (метод или оператор) называют *критической секцией* кода.

При запуске синхронизованного метода говорят, что объект *входит в монитор*, при завершении – что объект *выходит из монитора*. При этом поток, внутри которого вызван синхронизованный метод, считается владельцем данного монитора.

Имеется два способа синхронизации по ресурсам: синхронизация объекта и синхронизация метода.

Синхронизация объекта `obj1` при вызове несинхронизованного метода:

```
synchronized(obj1) оператор;
```

Синхронизация метода с помощью модификатора `synchronized` при задании класса:

```
public synchronized тип метод(...){...}
```

Кроме синхронизации по данным имеется *синхронизация по событиям*, когда параллельно выполняющиеся потоки приостанавливаются вплоть до наступления некоторого события, о котором им сигнализирует другой поток. Основными операциями при таком типе синхронизации являются `wait` ("ждать") и `notify` ("оповестить").

Имеется два способа создать класс, экземплярами которого будут потоки: унаследовать класс от `java.lang.Thread` либо реализовать интерфейс `java.lang.Runnable`.

Интерфейс `java.lang.Runnable` имеет декларацию единственного метода `public void run()`, который обеспечивает последовательность действий при работе потока. Класс `Thread` уже реализует интерфейс `Runnable`, но с пустой реализацией метода `run()`. Поэтому в потомке `Thread` надо переопределить метод `run()`.

При работе с большим количеством потоков требуется их объединение в группы. Такая возможность инкапсулируется классом `TreadGroup` ("Группа потоков").

Для получения доступа к библиотекам, написанным на других языках программирования, в Java используются методы, объявленные с модификатором *native* – "родной". При выполнении такого метода производится вызов "родного" для конкретной платформы двоичного кода, а не платформо-независимого байт-кода как во всех других случаях. Заголовок "родного" метода описывается в классе Java, а его реализация осуществляется на каком-либо из языков программирования, позволяющих создавать динамически подключаемые библиотеки.

Задания

Написать приложение, в котором используются потоки. Использовать задание класса потока как наследника `Thread`. Класс потока должен обеспечивать в методе `run` построчный несинхронизированный вывод в консольное окно чисел от 1 до 100 порциями по 10 чисел в строке, разделенных пробелами, причем перед каждой такой порцией должна стоять надпись "Thread 1:" для первого потока, "Thread 2:" для второго, и т.д. Для вывода строки задать в классе метод `print10`. В приложении по нажатии на первую кнопку должны создаваться два или более потоков, а при нажатии на вторую они должны стартовать.

Усовершенствовать приложение, обеспечив синхронизацию за счет объявления вызова `print10` в методе `run` синхронизированным.

Создать копию класса потока, отличающуюся от первоначальной тем, что выводятся числа от 101 до 200, класс задан как реализующий интерфейс `Runnable`, а метод `print10` задан как синхронизированный. Добавить в приложение создание и старт потоков – экземпляров данного класса.