top
КОМПЬЮТЕРНАЯ АКАДЕМИЯ

C++

</>

C++

ПРОГРАММИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ТЕХНОЛОГИИ

JAVA

Урок № 4

Многопоточность, сетевое взаимодействие

Содержание

Введение в многопоточность

Этот урок мы с вами начнем с рассмотрения тонкостей перевода. Так случилось, что в переводе на русский язык двух англоязычных терминов *thread* и *stream* используется одно и то же слово — «поток». Потоки *stream* — это потоки ввода-вывода, о которых мы говорили в предыдущем уроке. Потоки *thread* не имеют к потокам ввода-вывода никакого отношения, просто они обозначаются в русском языке таким же термином. Иногда такие потоки называют еще потоками выполнения. Так что же такое поток *thread*? Давайте договоримся, что в дальнейшем в этом уроке под термином «поток» будем понимать именно поток *thread*.

Такой поток представляет собой исполняемую сущность приложения. Сейчас объясню подробнее. Каждое приложение можно рассматривать как набор инструкций или команд, которые выполняются, когда приложение активируется. В любом компьютере только процессор умеет выполнять инструкции, из которых состоят приложения. Поэтому для того, чтобы приложение выполнялось, его инструкции должны каким-то образом доставляться процессору. Вот поток и занимается тем, что доставляет процессору код своего приложения для выполнения. Когда любое приложение активируется, операционная система создает для него поток, и этот поток берет с собой часть инструкций своего приложения и становится в очередь к процессору, чтобы эти инструкции выполнить. Вы, конечно же, понимаете, что эти инструкции в потоке

уже находятся в бинарном скомпилированном виде, а не в текстовом, как их написал программист.

Процессор в каждый момент времени может работать только с одним потоком. Вы должны знать, что Windows — это однозадачная операционная система. Однако за счет того, что процессор работает с каждым потоком на протяжении очень короткого промежутка времени, а затем переходит к работе со следующим потоком, у нас создается впечатление, что одновременно выполняется много программ. Как правило, поток, проработав с процессором один сеанс, не успевает выполнить весь необходимый код, поэтому он снова становится в очередь к процессору для новой встречи. Так происходит до тех пор, пока поток не выполнит все, что надо. Очередь потоков к процессору — кольцевая с приоритетами.

Если, например, на вашем компьютере запущено четыре разных приложения, то в очереди к процессору находятся четыре потока. Для простоты примера полагаем, что никакие другие системные приложения не работают, и других потоков в очереди к процессору нет. Оказывается, что можно написать такое приложение, от имени которого с процессором будет работать не один поток, а больше. К чему это приведет? Это приведет к тому, что потоков в очереди к процессору станет больше, и несколько из этих потоков будут принадлежать одному приложению. Такому приложению процессор будет уделять больше времени, так как от его имени с процессором работает несколько потоков. А само приложение, у которого более одного потока, называется многопоточным. Ваша задача заключается в том, чтобы научиться правильно создавать многопоточные приложения. Создавать новые потоки

технически не сложно. Гораздо сложнее научится создавать дополнительные потоки так, чтобы они не ухудшали работу приложения. Бытует мнение, что чем больше потоков в приложении вы создадите, тем быстрее приложение будет выполняться. Это категорически неверное утверждение. Неправильно созданные дополнительные потоки замедлят выполнение приложения или вообще приведут к аварийному завершению. Давайте научимся работать с дополнительными потоками.

В Java существует два способа создавать в приложении дополнительные потоки: с помощью класса Thread и с помощью интерфейса Runnable. Рассмотрим оба эти способа.

На приведенном ниже рисунке схематично показано поведение потоков, ожидающих доступа к процессору. Когда для текущего активного потока завершается время его работы с процессором, его дальнейшее поведение зависит от того, успел он выполнить всю свою работу, или нет. В первом случае поток завершается, во втором — снова становится в очередь к процессору. Место в очереди, которое поток при этом займет, определяется целым рядом факторов, о чем мы поговорим далее.

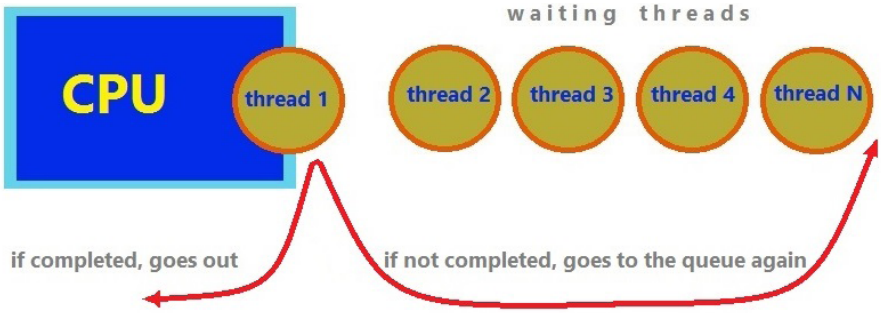

Рис 1. Потоки в череди к процессору

Создание и запуск потоков

Класс Thread

Чтобы создать в приложении новый поток выполнения, можно создать объект класса Thread. В дальнейшем под термином «поток» будем понимать именно объект этого класса.

Полное описание этого класса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html.

Еще раз хочу напомнить, что при запуске любого приложения операционная система создает для этого приложения поток выполнения. Иногда его называют первичным или основным. Этот поток активен, пока активно запущенное приложение (пока выполняется метод main()). Когда метод main() завершает работу (в нормальном режиме или в аварийном), основной поток прекращает существование. В однопоточном приложении весь код приложения выполняется в основном потоке, но если вы создадите объект класса Thread, то получите возможность выполнять определенный код своего приложения в дополнительном потоке одновременно с основным потоком.

Каждый поток имеет несколько важных характеристик.

Прежде всего, поток характеризуется приоритетом. Для обозначения приоритета потока в Java используется набор констант: Thread.MIN_PRIORITY, Thread.NORM_PRIORITY и Thread.MAX_PRIORITY.

Вы должны понимать, что значения этих констант не представляют собой точные значения приоритетов в какой-либо операционной системе, поскольку потоки Java выполняются на совершенно разных платформах. Приоритет влияет на то, как долго поток будет ожидать в очереди доступа к процессору. Приоритет потока можно изменять, об этом поговорим позже. По умолчанию, вновь созданный поток имеет такой же приоритет, как и создавший его поток.

У потока есть строковое имя, которое бывает полезным для ссылки на него. Имена потокам присваиваются системой по умолчанию, но при желании можно присваивать им собственные имена.

Кроме этого, поток может быть помечен как «демон». Особенность потоков демонов состоит в том, что они не препятствуют завершению работы JVM. Обычно виртуальная машина Java не прекратит работу до тех пор, пока активен хоть один не демон поток. Она подождет завершения его работы. Если же активен демон поток, JVM прекратит работу, принудительно завершив такой поток.

Бытует ошибочное мнение, что потоки–демоны имеют меньший приоритет, чем потоки не демоны. Это не так. «Демоничность» потока никак не влияет на его приоритет. Потоки демоны являются хорошим средством для выполнения фоновых задач.

Перейдем к рассмотрению примеров. Создайте новый проект и добавьте в его состав такой класс:

```
public class MyThread1 extends Thread{
    public void run(){
```

```
        while(true){
            System.out.print("A");
        }
    }
}
```

Если вам надо создать в приложении дополнительный поток, вы можете поступить таким образом. Создайте в приложении класс с произвольным именем и наследуйте его от класса Thread. В созданном классе вам надо переопределить метод с именем run(). Этот метод называется потоковым методом. Потоковый метод содержит в себе тот код, который будет выполняться в дополнительном потоке одновременно с другими потоками вашего приложения. Каждый поток обязательно должен иметь потоковый метод. Поток активен и выполняется, пока не завершен его потоковый метод. Когда потоковый метод завершается, связанный с ним поток удаляется. В нашем примере в потоковом методе будет выполняться бесконечный цикл, выводящий в консольное окно символ «A».

Посмотрим, как этот поток можно запустить. Добавьте в метод main() такие строки:

```
MyThread1 t1=new MyThread1();
t1.start();
while(true){
    System.out.print(".");
}
```

Сначала мы создаем объект нашего класса MyThread1, а затем от имени этого объекта вызываем метод start().

Вызов метода start() активирует дополнительный поток, это приводит к вызову потокового метода run(), и в консольное окно начинают выводиться символы «A». Дальше в методе main() запускается собственный бесконечный цикл, который выводит в консольное окно символ «.». Сейчас от имени нашего приложения с процессором работают два потока: основной, который выводит символ «.», и дополнительный, который выводит символ «A». Они по очереди будут получать доступ к процессору, поэтому в консольном окне вы увидите хаотичное чередование «.» и «A»:

```
......AAAAAAAAAA..........AAAAAAA.........AA......
AA..AA..AA..AA..AA..AA......AA..AAAAAA....
```

Теперь немного изменим код в методе main():

```
MyThread1 t1=new MyThread1();
t1.start();
int laps=10000;
while(laps>0){
    System.out.print(".");
    laps--;
}
```

Сделаем так, чтобы основной поток — метод main() — работал не бесконечно, а завершался, выполнив определенное число итераций. Если вы сейчас запустите приложение, то увидите, что оно продолжает работать «бесконечно», и при этом спустя какое-то время в консольное окно выводятся только символы «A». О чем это говорит? Это говорит о том, что метод main() завершил

работу, а приложение продолжает выполняться! Вы уже понимаете, почему это происходит. Потому что от нашего приложения активным остается дополнительный поток, ведь в нем мы оставили бесконечный цикл. И поскольку один поток от приложения активен, JVM не завершает приложение. Добавьте в main() еще одну строку:

```
MyThread1 t1=new MyThread1();
t1.setDaemon(true);
t1.start();
int laps=10000;
while(laps>0){
    System.out.print(".");
    laps--;
}
```

Мы сделали дополнительный поток демоном. Запустите приложение снова. Сейчас приложение завершает работу, как только выполняется метод main(). Да, казалось бы, дополнительный поток должен продолжать выполнять свой бесконечный цикл, но сейчас этот поток — демон. Поэтому, в отличие от предыдущего случая, JVM завершает приложение сразу же по завершении метода main(), прекращая работу потока демона. Вот в чем особенности потоков демонов.

Давайте изменим приоритет дополнительного потока на минимальный. Будет ли визуально заметно отличие в выводе в консольном окне? Добавьте еще одну строку:

```
MyThread1 t1=new MyThread1();
t1.setDaemon(true);
t1.setPriority(Thread.MIN_PRIORITY);
```

```
t1.start();
int laps=10000;
while(laps>0){
    System.out.print(".");
    laps--;
}
```

Запустите приложение. У меня символов «A» заметно поубавилось. Это произошло потому, что поток, выводящий их, получает теперь меньше доступа к процессору.

Вы должны помнить, что в ОС Windows два самых высоких уровня приоритетов обозначаются константами THREAD_PRIORITY_HIGHEST и THREAD_PRIORITY_TIME_CRITICAL. При этом последний уровень приоритета обычно присваивается жизненно важным для операционной системы процессам. До версии Java 5 включительно, значение приоритета потока Thread.MAX_PRIORITY при выполнении приложения в ОС Windows могло отображаться на приоритет THREAD_PRIORITY_TIME_CRITICAL. Однако, начиная с Java 6 максимальное значение Thread.MAX_PRIORITY может соответствовать только значению THREAD_PRIORITY_HIGHEST. Изменение уровня приоритета потока, вообще говоря, только опосредованно влияет на доступ потока к процессору. Например, если вы создадите поток с минимальным приоритетом, в сравнении с другими потоками, активными в одно время с ним, вы все равно не можете быть уверенны в том, что этот поток не получит доступ к процессору. Поток с минимальным приоритетом получит доступ к процессору, например, если все другие потоки с более высокими приоритетами перейдут в состояние ожидания.

Более того, в ОС Windows есть специальный механизм, который предоставляет потоку доступ к процессору, если поток долгое время не имел такого доступа из-за низкого приоритета. Поэтому запомните такое утверждение: значение приоритета потока не является единственным фактором, влияющим на доступ потока к процессору. В любой операционной системе управление потоками (*Thread Scheduling*) — это сложный процесс с большим количеством разных факторов.

Обсудим дополнительный поток, созданный в этом приложении. Он оформлен в виде отдельного класса, производного от класса Thread. Такой класс часто называют потоковым. В потоковом классе обязательно должен быть переопределен метод с именем run(), который, в свою очередь, называется потоковым методом. В потоковом методе располагается тот код, который будет выполняться в созданном дополнительном потоке. Потоковый метод не вызывается явно. Он вызывается системой после того, как вы вызовете метод start() от имени потокового класса. Когда потоковый метод выполнится, дополнительный поток будет уничтожен. Запомните: поток нельзя запустить повторно после его завершения. Т.е. в нашем случае вы не можете второй раз вызвать метод start() от объекта t1. Попытка запустить поток повторно приведет к исключению *IllegalThreadStateException*. Если вам надо выполнить потоковый метод повторно — создавайте новый объект потокового класса и вызывайте start() от него.

Как быть в том случае, если в дополнительный поток надо передать какие-либо данные? Эта задача решается таким образом. Вы можете в потоковом классе создать

необходимые поля и инициализировать их при создании объекта потокового класса. Значения таких полей будут доступны в потоковом методе run().

Добавим в наш проект новый класс:

```
public class MyThread2 extends Thread{
    private int value;

    public MyThread2(int v)
    {
        this.value=v;
    }

    public void run(){
        while(this.value >=0){
            System.out.println("*** The thread "
                +Thread.currentThread().getName()+
                " started with " +this.value);
        while(this.value >=0){
                System.out.println("From "
                    +Thread.currentThread().getName()+
                    " value =" +this.value);
                this.value-=1;
                try {
                     Thread.sleep(300);
                } catch (InterruptedException ex) {
            }
        }
        System.out.println("*** The thread "
            +Thread.currentThread().getName()+
            " has finished");
        }
    }
}
```

В этом потоковом классе мы объявили поле value, которое инициализируется в конструкторе при создании

объекта класса. В самом начале потокового метода run() мы выводим имя потока Thread.currentThread().getName() и начальное значение поля value. Затем запускаем цикл, в котором на каждой итерации уменьшаем значение value на 1. Чтобы притормозить эти действия, мы приостанавливаем выполнение потока на каждой итерации на 300 миллисекунд вызовом метода Thread.sleep(). Вызов этого метода должен выполняться в try-catch блоке. Этот поток завершит работу, когда значение value достигнет 0. После цикла поток выведет сообщение об окончании работы и будет завершен.

Создадим в методе main() два объекта этого потокового класса с разными случайными значениями value и посмотрим, как будет выполняться наше приложение.

```
int v = (new Random()).nextInt(10);
MyThread2 t2=new MyThread2(v);
t2.setDaemon(true);
v = (new Random()).nextInt(10);
MyThread2 t3=new MyThread2(v);
t3.setDaemon(true);
t2.start();
t3.start();
```

Запустите приложение и объясните то, что вы увидели. Сформулируем вопрос точнее: объясните, почему вы ничего не увидели? Почему приложение молча завершило работу и ничего не вывело в консольное окно? Вы уже должны понимать, почему это произошло. Посмотрите внимательно на код в методе main(). Создаются два потока демона, по очереди для каждого из них вызывается метод start(). А что происходит дальше? А дальше метод

main() просто завершает работу, а, следовательно, завершается работа всего приложения. Поскольку потоки t2 и t3 демоны, они уничтожаются при завершении работы приложения, даже не успев начать выполняться.

Закомментируйте строки t2.setDaemon(true) и t3.setDaemon(true) и запустите приложение снова. Теперь картина другая. Например, у меня получился такой вывод:

```
*** The thread Thread-0 started with 4
*** The thread Thread-1 started with 0
From Thread-1 value =0
From Thread-0 value =4
From Thread-0 value =3
*** The thread Thread-1 has finished
From Thread-0 value =2
From Thread-0 value =1
From Thread-0 value =0
*** The thread Thread-0 has finished
```

Рассмотрим, что здесь происходит. Сразу бросаются в глаза имена потоков: Thread-0 и Thread-1. Эти имена присваиваются потокам системой, но при необходимости вы можете переименовать их. Обратите внимание, у меня получилась интересная ситуация, когда одному из потоков, а именно, потоку Thread-1, было присвоено случайное значение 0, а второму потоку — значение 4. Поток Thread-0 вывел сообщение о начале работы, но не успел выполнить ни одной итерации своего цикла и был оттеснен от процессора потоком Thread-1, который сообщил о начале работы со значением 0 и успел выполнить одну итерацию, но затем сам был оттеснен от процессора потоком Thread-0. Дальше все понятно. Почему мы сейчас

увидели результаты работы наших двух потоков? Потому, что сейчас эти потоки — не демоны. И приложение не завершается, пока каждый из них не выполнит свой потоковый метод.

Предположим, мы хотим, чтобы оба этих потока были демонами. Есть ли какая-нибудь возможность, чтобы главный поток, метод main(), подождал завершения этих потоков и только потом завершил приложение? Да, есть несколько таких возможностей, но не всеми надо пользоваться в реальных проектах. Рассмотрим сначала приемы, которыми пользоваться не следует.

Первый способ ожидания завершения потока, которым пользоваться *не следует*.

В метод main() сразу после запуска потоков t2 и t3 можно вставить такой цикл:

```
while(t2.isAlive() || t3.isAlive())
{
}
```

Булев метод isAlive() возвращает true, если поток еще активен, и false — если поток уже завершился. Этот пустой цикл в главном потоке будет выполняться пока хоть из дополнительных потоков будет активен, и таким образом, метод main() будет активен тоже и не завершит дополнительные потоки преждевременно. Будет это работать? Да, будет. Почему же не надо использовать этот способ ожидания завершения потоков? Потому что главный поток для выполнения этого пустого цикла будет тратить ресурсы процессора, отнимая их у других потоков, выполняющих полезную работу.

Второй способ ожидания завершения потока, которым пользоваться не следует.

В метод main() сразу после запуска потоков t2 и t3 можно вставить такой код:

```
try {
    Thread.sleep(5000);
} catch (InterruptedException ex) {
}
```

Мы приостанавливаем главный поток на время, необходимое для того, чтобы дополнительные потоки завершили работу. Потом главный поток активируется и завершает приложение уже после того, как оба потока завершены. Этот способ также будет работать, к тому же, в этом случае главный поток не занимает ресурсы процессора. Почему же этот способ плох? Потому что вы не можете точно определить, сколько времени надо ожидать в спящем состоянии. И такой способ не дает вам гарантии того, что ваши потоки обязательно завершат свою работу к тому моменту, когда проснется главный поток.

Перейдем к рассмотрению правильного способа ожидания завершения работы потока.

В метод main() сразу после запуска потоков t2 и t3 можно вставить такой код:

```
try {
    t2.join();
    t3.join();
} catch (InterruptedException ex) {
}
```

Что делает метод join()? Он приостанавливает выполнение того потока, в котором он вызван, до тех пор, пока не завершится работа потока, от имени которого вызван join(). В нашем случае t2.join() вызывается в главном потоке, в методе main(), поэтому главный поток будет приостановлен. Возобновится выполнение главного потока тогда, когда завершится поток t2. Аналогично с потоком t3. А что произойдет в случае, если дополнительный поток, t2 или t3, зависнет или попадет в бесконечный цикл? Тогда и главный поток будет ожидать «вечно». Если вас не устраивает такая ситуация, вы можете вызывать метод join(), указав ему в качестве параметра таймаут в миллисекундах. При этом, если дополнительный поток не завершит работу до истечения таймаута, главный поток разблокируется и продолжит свое выполнение. Например, в таком случае:

```
try {
    t2.join(10000);
    t3.join(10000);
} catch (InterruptedException ex) {
}
```

главный поток продолжит выполнение через 10 секунд независимо от того, завершатся к этому моменту t2 и t3 или нет.

Запомните еще один момент. С потоками никогда нельзя быть уверенным в том, какой из них запустится или завершится первым. В нашем случае мы сначала запускаем поток t2, а затем — t3, но если вы выполните приложение несколько раз, вы поймаете тот момент, когда

t3 запустится раньше. Дело в том, что на работу потоков влияет очень много разных факторов, делающих их поведение непредсказуемым.

А что делать, когда ваш класс уже является производным от какого-то другого класса, а вы хотите сделать его потоковым? Ведь для того, чтобы сделать класс потоковым, его надо наследовать от Thread, а в Java множественное наследование запрещено. В этом случае вы можете сделать свой класс потоковым с помощью интерфейса Runnable.

Интерфейс Runnable

Вы можете сделать класс потоковым, наследовав его от интерфейса Runnable. При этом вам надо будет переопределить в своем классе метод run(), который, конечно же, является потоковым методом. Полное описание интерфейса Runnable можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html.

Повторим предыдущий пример, но теперь дополнительные потоки создадим с использованием интерфейса Runnable. Добавьте в наше приложение еще один класс:

```
public class MyThread3 implements Runnable {
    private int value;
    public MyThread3(int v)
    {
        this.value=v;
    }

    public void run(){
        System.out.println("*** The thread "+
            Thread.currentThread().getName()+
```

```
        " started with " +this.value);
        while(this.value >=0){
            System.out.println("From "+
                Thread.currentThread().getName()+
                " value =" +this.value);
            this.value-=1;
            try {
                Thread.sleep(300);
            } catch (InterruptedException ex) {
            }
        }
        System.out.println("*** The thread "+
            Thread.currentThread().getName()+
            " has finished");
    }
}
```

Приведенный класс является точной копией нашего класса MyThread2, отличаясь только базовым типом. Добавим в main() код, использующий этот класс, и разберем, как создаются дополнительные потоки при использовании интерфейса Runnable:

```
int v = (new Random()).nextInt(10);
MyThread3 r4=new MyThread3(v);
Thread t4 = new Thread(r4);
t4.setDaemon(true);
v = (new Random()).nextInt(10);
MyThread3 r5=new MyThread3(v);
Thread t5 = new Thread(r5);
t5.setDaemon(true);
t4.start();
t5.start();
try {
    t4.join();
```

```
    t5.join();
    } catch (InterruptedException ex) {
}
```

Для создания такого потока сначала надо создать объект класса, производного от Runnable, а затем — объект класса Thread, передав в конструктор объект наследника Runnable. У меня выполнение этого приложения сгенерировало такой вывод в консольное окно:

```
*** The thread Thread-1 started with 2
From Thread-1 value =2
*** The thread Thread-0 started with 0
From Thread-0 value =0
From Thread-1 value =1
*** The thread Thread-0 has finished
From Thread-1 value =0
*** The thread Thread-1 has finished
```

Кстати, обратите внимание, что второй поток запустился раньше первого.

У вас может возникнуть вопрос: какой способ создания потоков является предпочтительным — использование Thread или Runnable? Хотя с функциональной точки зрения оба способа являются одинаковыми, большинство пользователей склоняется к тому, что использование Runnable является более удобным. Для этого есть несколько причин. Во–первых, вашему потоковому классу может требоваться наследование от другого класса, и наследовать его еще и от Thread возможности у вас не будет. Во–вторых, оформление потокового кода в виде Runnable используется многими другими классами, на-

пример ThreadPoolExecutor, что будет рассмотрено позже. В–третьих, с семантической точки зрения, при extends мы переопределяем нужные нам методы, а при implements мы их реализуем. При создании потока речь идет именно о реализации метода run().

Использование Runnable оставляет больше свободы выбора при использовании вашего кода. Например, если вы передумали выполнять свой потоковый метод, оформленный в Runnable, в дополнительном потоке, вы можете вызвать его непосредственно от объекта Runnable - t.run(), и тогда этот метод выполнится в основном потоке, без создания дополнительного потока.

Многие утверждают, что эти отличия несущественны и принципиальной разницы между Thread и Runnable нет. Возможно и так. Каждый судит на основании собственного опыта, поэтому я желаю вам побыстрее набраться опыта работы с потоками и самостоятельно решить для себя, когда и каким способом создавать потоки.

Управление потоками

Мы научились создавать и запускать в приложении дополнительные потоки. Теперь давайте рассмотрим, как мы можем управлять их выполнением. Можно ли остановить поток принудительно? Можно ли приостановить поток, а затем возобновить его выполнение? В ранних версиях Java были созданы методы, которые предназначались для выполнения таких действий: stop(), suspend(), resume(). Однако с тех пор отношение к тому, что делают эти методы, сильно изменилось. Эти методы уже давно объявлены deprecated, и пользоваться ими нельзя. На сегодняшний день в Java нет средств принудительного завершения или остановки потока. Почему это произошло?

Рассмотрим мысленный эксперимент. Дополнительный поток открыл соединение с базой данных и начал выполнять транзакцию, а его принудительно завершают. Кто завершит транзакцию? Дополнительный поток прочитал данные из сети, но не успел с ними ничего сделать, так как его принудительно завершили. Сохраняться ли эти данные при следующем запуске этого или другого потока? Таких примеров можно придумать много. Дело в том, что принудительная остановка потока или его приостановка с последующим возобновлением очень часто приводит к потере данных. Поэтому в Java предлагается другой способ выполнения подобных действий с потоками. Он заключается в том, что при необходимости остановить поток вы просто сообщаете потоку об этом, а решение о том, прекращать работу или нет, будет принимать сам

поток. Рассмотрим, как это можно сделать. Для рассматриваемого примера будем снова использовать наш класс MyThread3.

Метод interrupt()

В классе Thread определен метод interrupt(), который позволяет переслать потоку просьбу прекратить работу. Обратите внимание на формулировку. Вызов этого метода ни в коем случае не прекращает выполнение потока, а просто изменяет состояние некоего флага в потоке. Поток может проверить этот флаг и увидеть, что его попросили остановиться. А может совсем и не проверять его и даже не знать о том, что его попросили прекратить работу. Все зависит от того, как разработчик потока написал код. Будет описана реакция на выставленный флаг — поток отреагирует, не будет — поток продолжит работу. Такой режим остановки потока можно назвать уведомительным. Потоку отправляют уведомление с просьбой остановиться, а он либо реагирует на это уведомление, либо игнорирует его. Почему это сделано именно так? Для того, чтобы в случае преждевременной остановки потока, программист выполнил бы в потоке сохранение всех внешних ресурсов, если он с таковыми работает (файлов, баз данных, сетевых подключений), а уже затем остановил поток. Обратите внимание — речь идет не о приостановке потока с возможностью последующего возобновления, а именно об остановке.

Поговорим об использовании метода interrupt() подробнее, а затем применим его в нашем приложении. Рассмотрим жизненный цикл дополнительного потока.

После запуска он выполняет код, описанный в своем потоковом методе run(). Если в методе run() встречается вызов метода sleep(), то поток на какое-то время блокируется, а затем возобновляет работу. Другими словами, жизненный цикл потока состоит из периодов активности и периодов блокировки. Блокироваться поток может не только методом sleep(), а, например, еще методом wait(), который мы рассмотрим позже.

Теперь представьте, что вы вызываете для дополнительного потока метод interrupt(). Результат такого вызова будет зависеть от того, в каком состоянии находился поток в момент вызова interrupt(). Если поток был в активном состоянии, то вызов interrupt() выставит в этом потоке свой флаг, на который поток сможет реагировать или же не реагировать. Если же поток находился в блокированном состоянии, то вызов метода interrupt() разблокирует его и выбросит исключение InterruptedException. А выброшенное исключение снимет флаг, выставленный методом interrupt(). Другими словами, выполнение потока в этом случае будет переведено в catch блок, и при этом флаг прерывания будет сброшен.

Скажите честно, всегда ли вы добросовестно относитесь к обработке исключительных ситуаций? Чаще всего в catch блоке программист ограничивается записью в лог. А эти логи почти никогда не читает. А иногда catch блок вообще оставляется пустым (как в нашем примере J) и исключения просто «проглатываются» без обработки. Так поступать нельзя. А когда вы работаете с потоками — так поступать нельзя категорически! Рассмотрим пример и подробно его обсудим.

Добавим в наш код отправку второму потоку сообщения о преждевременном завершении работы. В потоке опишем такую реакцию: если поток успел выполнить половину итераций — он остановится, если же половина итераций еще не выполнена —продолжит работу, чтобы выполнить половину итераций, а уже затем остановится.

В метод main() добавим строку, которая отправляет второму потоку уведомление о завершении работы:

```
int v = (new Random()).nextInt(10);
MyThread3 r4=new MyThread3(v);
Thread t4 = new Thread(r4);
t4.setDaemon(true);
v = (new Random()).nextInt(10);
MyThread3 r5=new MyThread3(v);
Thread t5 = new Thread(r5);
t5.setDaemon(true);
t4.start();
t5.start();
t5.interrupt();
try {
    t4.join();
    t5.join();
    } catch (InterruptedException ex) {
}
```

Теперь в потоковом методе опишем реакцию на такое уведомление:

```
public void run(){
    int limit=this.value/2;
    System.out.println("*** The thread "+
        Thread.currentThread().getName()+
        " started with " +this.value);
    while(this.value >=0){
```

```
        System.out.println("From "+
            Thread.currentThread().getName()+
            " value =" +this.value);
        this.value-=1;
        if(this.value >= limit && Thread.
            currentThread().isInterrupted())
        {
            System.out.println("*** The thread
                is interrupted");
            return;
        }
        try {
            Thread.sleep(300);
        } catch (InterruptedException ex) {

        }
    }
    System.out.println("*** The thread "+
        Thread.currentThread().getName()+
        "has finished");
}
```

Сначала мы вычисляем для потока минимальное количество итераций, которые он должен выполнить, и сохраняем его в limit. Затем на каждой итерации проверяем, выполнено ли уже это минимальное число итераций и не вызывался ли для потока метод interrupt(). Помните, мы говорили, что этот метод выставляет некоторый флаг внутри потока? Этот флаг явно увидеть нельзя, но проверить его состояние можно несколькими способами. Один из них — вызов метода isInterrupted() от имени объекта потока. Чтобы получить доступ к объекту исполняемого потока, мы использовали вызов Thread.currentThread(). Так вот, если был вызван метод interrupt() и при этом был

выставлен тот флаг внутри потока, то метод isInterrupted() вернет true. При этом поток войдет в тело if, напишет в консоль сообщение «*** *The thread is interrupted*» и завершит работу. Обратите внимание, что catch блок мы намеренно пока оставили пустым.

Запустите приложение несколько раз. Я уверен, что сообщение «*** *The thread is interrupted*» вы, скорее всего, не увидели. У меня вывод получился такой:

```
*** The thread Thread-1 started with 4
From Thread-1 value =4
*** The thread Thread-0 started with 8
From Thread-0 value =8
From Thread-1 value =3
From Thread-0 value =7
From Thread-1 value =2
From Thread-0 value =6
From Thread-1 value =1
From Thread-0 value =5
From Thread-1 value =0
From Thread-0 value =4
*** The thread Thread-1 has finished
From Thread-0 value =3
From Thread-0 value =2
From Thread-0 value =1
From Thread-0 value =0
*** The thread Thread-0 has finished
```

В этом случае второй поток Thread-1 стартовал со значением value = 4. Мы отправили ему вслед, в методе main(), вызов метода interrupt(). По нашему плану, этот поток должен выполнить две итерации и прекратить работу с сообщением «*** *The thread is interrupted*». Однако он выполнил все свои четыре итерации и только

потом завершился с сообщением «*The thread Thread-1 has finished*». Почему этот поток не вошел в блок обработки завершения?

Дело в том, что в коде потокового метода run() есть ошибка. Такую ошибку часто допускают неопытные программисты, забывая или не зная об одной особенности метода interrupt(). Внимательно прочитайте и запомните следующее. Если в момент вызова метода interrupt() поток находится в активном состоянии, то в нем выставляется соответствующий флаг, на который поток может отреагировать. Если же в момент вызова этого метода поток находится в спящем или заблокированном состоянии, например, после вызова метода sleep() или wait(), то он будет разбужен и выбросит исключение InterruptedException. А выброшенное исключение снимет флаг, выставленный методом interrupt().

Что происходит в нашем примере? Поскольку в потоковом методе мы используем метод sleep(), то очень часто в момент вызова в main() метода interrupt() наш поток будет находиться в спящем состоянии. В этом случае вызов interrupt() разбудит поток и будет выброшено исключение InterruptedException. Управление будет передано в catch блок, *и при этом будет сброшен флаг, выставленный методом* interrupt(). В такой ситуации наш дополнительный поток не попадет в блок обработки завершения и не выведет в консольное окно сообщение «**** The thread is interrupted*». Как исправить такое поведение дополнительного потока? Надо вспомнить, что выброшенное исключение сбрасывает флаг прекращения работы, поэтому достаточно в catch блоке просто снова

выставить его. Поток увидит флаг при продолжении выполнения и завершится так, как мы запланировали. Выставить этот флаг можно повторным вызовом метода interrupt() от имени объекта потока:

```
public void run(){
    int limit=this.value/2;
    System.out.println("*** The thread "+
        Thread.currentThread().getName()+
        " started with " +this.value);
    while(this.value >=0){
        System.out.println("From "+
        Thread.currentThread().getName()+
        " value =" +this.value);
        this.value-=1;
        if(this.value <= limit && Thread.
            currentThread().isInterrupted())
        {
            System.out.println("*** The thread is
                interrupted ");
            return;
        }

        try {
            Thread.sleep(300);
        } catch (InterruptedException ex) {
            Thread.currentThread().interrupt();
        }
    }
    System.out.println("*** The thread "+
        Thread.currentThread().getName()+
        " has finished");
}
```

Запустите приложение еще раз. Теперь все должно выполниться по плану. У меня получился такой вывод:

```
*** The thread Thread-0 started with 6
From Thread-0 value =6
*** The thread Thread-1 started with 6
From Thread-1 value =6
From Thread-0 value =5
From Thread-1 value =5
From Thread-1 value =4
*** The thread is interrupted
From Thread-0 value =4
From Thread-0 value =3
From Thread-0 value =2
From Thread-0 value =1
From Thread-0 value =0
*** The thread Thread-0 has finished
```

Поток Thread-1 выполнил половину итераций и завершился с ожидаемым сообщением. Это произошло потому, что в catch блоке мы снова выставили флаг, сброшенный в момент возникновения исключения, и при продолжении работы поток отреагировал на этот флаг и вошел в блок обработки.

И еще одно замечание. Никогда не оставляйте catch блок пустым! Если бы в этом блоке был прописан хотя бы вывод об исключении, это дало бы ключ к решению данной проблемы. Существует еще один способ проверки флага, выставляемого методом interrupt(). Мы использовали объектный метод isInterrupted(), а можно было еще воспользоваться статическим методом Thread.interrupted(). Этот метод также возвращает true, если флаг установлен и при этом Thread.interrupted() сбрасывает проверенный флаг, если он был установлен, в то время как isInterrupted() состояние флага не изменяет.

Синхронизация

Synchronized блоки и методы

Теперь, когда вы научились создавать в приложениях дополнительные потоки, вам надо научиться избегать проблем, которые могут возникать в многопоточных приложениях при неправильном использовании потоков.

Рассмотрим очередной пример, который продемонстрирует одну из распространенных проблем, присущих многопотоковым приложениям. Сделаем так, чтобы два потока работали одновременно с одним и тем же ресурсом, и посмотрим, как потоки будут выполнять такую работу.

Создайте новое приложение. В главном классе приложения рядом с методом main() создайте статическое поле counter. Именно с этим полем и будут работать потоки:

```
public class Treadtest {
    public static int counter=0;

    public static void main(String[] args) {
        int limit = 1000;
        IncThread t6=new IncThread(limit);
        DecThread t7=new DecThread(limit);
        t6.start();
        t7.start();
        try {
            t6.join();
            t7.join();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
```

```
            Logger.getLogger(Treadtest.class.getName()).
            log(Level.SEVERE, null, ex);
        }
        System.out.println("counter="+counter);
    }
```

Теперь добавьте два потоковых класса IncThread и DecThread, которые будут работать с полем counter. Первый класс будет увеличивать значение этого поля на 1 в цикле limit раз. Второй класс будет уменьшать значение этого поля на 1 тоже limit раз. Обратите внимание, что оба класса получают при создании одно и то же значение limit, которое определяет выполняемое классами в потоковом методе количество итераций. Когда оба потока завершат работу, метод main() выведет в консольное окно текущее значение поля counter. Вот оба потоковых класса:

```
public class IncThread extends Thread{
    int limit;
    public IncThread(int limit)
    {
        this.limit=limit;
    }
    public void run(){
        for(int i=0; i<limit; i++)
        {
            Treadtest.counter++;
        }
    }
}

public class DecThread extends Thread{
    int limit;
    public DecThread(int limit)
```

```
    {
        this.limit=limit;
    }
    public void run(){
        for(int i=0; i<limit; i++)
        {
            Treadtest.counter--;
        }
    }
}
```

Укажите начальное значение limit, равное 1000, и запустите приложение. Чему равняется значение counter после завершения приложения? У меня получилось значение 0. Все логично: один поток увеличил это поле 1000 раз на единицу, второй — его 1000 раз на единицу. А теперь увеличьте значение limit до 10000 и запустите приложение снова. Ну что, по–прежнему counter равно 0? Если да, то увеличьте значение limit до 100000 и снова запустите приложение. Вы обязательно столкнетесь с ситуацией, когда limit не будет равняться 0, а будет больше 0 или меньше 0. Причем, при каждом запуске значение будет другим. Странное поведение приложения, правда?

Разберем, почему мы получили такое поведение. Почему при небольших значениях limit приложение ведет себя вполне ожидаемо, а при увеличении этого значения начинаются странности? Когда значение limit небольшое, каждый поток успевает выполнить свои итерации за один сеанс работы с процессором. Неважно, какой из потоков первым выполнит свою работу. Тот факт, что мы первым запускаем IncThread, вовсе не дает нам уверенности в том,

что именно этот поток первым выполнит свою работу. Важно то, что один из потоков увеличит поле counter limit раз, а второй потом уменьшит это поле limit раз за один сеанс работы с процессором.

Что произойдет, когда значение limit будет настолько большим, что поток не успеет выполнить limit итераций за один подход к процессору? Скорее всего, возникнет ситуация, при которой потоки начнут искажать результаты работы друг друга. При начале работы с процессором поток выгружает свой код из собственного стека в регистры процессора, и CPU начинает выполнять полученные инструкции. Когда время работы потока с процессором завершается, процессор выгружает данные из своих регистров обратно в стек потока, чтобы при следующей встрече с этим же потоком забрать данные из стека и продолжить работу с того места, где произошла остановка. Кроме этого, процессор должен записать изменившиеся значения в те переменные, которые были изменены в ходе выполненной работы. Иногда происходит так, что процессор успевает прочитать текущее значение переменной, добавляет к нему требуемое для изменения значение, но не успевает записать измененное значение в переменную. Тогда он запоминает, что и куда надо записать и выполняет это сразу же при следующей встрече с этим же потоком. Можно сказать, что в таком случае операция += заменяется двумя операциями: увеличением сейчас и присваиванием при следующей встрече. Такое явление описывается термином *Race condition*.

Допустим, IncThread выполнил 2000 итераций, и его время работы с процессором истекло. В нашем мыслен-

ном примере процессор не успел записать значение 2000 в counter, но запомнил это. Затем к процессору пришел поток DecThread и уменьшил значение counter на 1500, и в этот раз процессор успел записать в counter значение –1500 (минус одна тысяча пятьсот) и расстался с этим потоком. А затем к процессору снова пришел поток IncThread. У процессора хорошая память, и он вспомнил, что, расставаясь с этим потоком в прошлый раз, он не успел записать в переменную counter значение 2000. Поэтому прежде всего процессор сделает это, а затем продолжит работу с потоком. Однако записывая значение 2000, процессор затрет в counter результаты работы предыдущего потока DecThread! Процессор не может и не обязан знать, что эти потоки работают с одной и той же переменной. И чем больше подходов к процессору будет у наших потоков, тем больше вероятность возникновения таких ошибок. Такие ошибки возникают, когда несколько потоков работают одновременно с одним и тем же ресурсом. В таких случаях говорят, что потоки не синхронизированы. Давайте запомним эту ситуацию: когда разные потоки работают одновременно с одним и тем же ресурсом, они могут искажать результаты работы друг друга.

Как можно устранить такое их поведение? Надо сделать так, чтобы поток блокировал ресурс в том случае, когда он не успел завершить работу с ресурсом. Другими словами, если, например, поток IncThread не успел обновить значение нашей переменной counter, он должен заблокировать доступ к этой переменной. Когда после этого к процессору придет поток DecThread, переменная counter будет недоступна для работы, так как будет нахо-

диться в заблокированном состоянии, и поток DecThread пропустит свой сеанс работы с процессором. Конечно же, это несколько замедлит общее время работы потоков, но зато мы получим неискаженный результат. Запомните: такая блокировка не приведет к тому, что сначала полностью выполнится первый поток, а потом — второй. Нет, потоки будут работать одновременно, получая доступ к процессору в порядке очереди. При этом если после какого-то сеанса работы с процессором один из потоков не успеет занести в переменную результаты своей работы, он заблокирует переменную, уходя от процессора. И эта переменная будет никому не доступна, пока этот же поток не завершит свою работу с ней при следующем сеансе работы с процессором. Как можно выполнить такую блокировку? В Java для этого применяются synchronized блоки. Если заключить какой-то фрагмент кода в такой блок, то этот код одновременно сможет выполняться только одним потоком. Любые другие потоки, которым надо работать с этим кодом, должны будут ожидать завершения работы потока, захватившего такой блок. Чтобы синхронизировать работу нескольких потоков с помощью synchronized блока, надо иметь в своем распоряжении специальный объект, доступный всем этим потокам. Этот объект обязательно должен быть ссылочного типа. Именно по состоянию этого объекта, который обычно называют lock, потоки будут понимать, свободен или заблокирован synchronized блок. В synchronized блок можно заключать любой объем кода, но обычно надо стараться включать в такой блок только код, использующий разделяемый ресурс. Спецификатор synchronized можно использовать также для того, чтобы

сделать синхронизированным целый метод, объектный или статический. Мы рассмотрим разные способы использования этого спецификатора в примерах.

Внесем изменения в наш пример, чтобы потоки не мешали друг другу изменять значение переменной counter. Добавим в наш класс Treadtest еще одно статическое поле:

```
public static Object locker=new Object();
```

Это поле public, поэтому будет доступно всем потокам нашего приложения. Тип этого поля Object, т.е. ссылочный, что является обязательным условием. Не обязательно использовать именно тип Object. Можно было использовать String или Fish, или любой другой ссылочный тип. Значение такого объекта совершенно не важно. Обратите внимание, что это должен быть именно объект. Если бы вы объявили только ссылку:

```
public static Object locker;
```

то получили бы ошибку.

Теперь надо внести небольшие изменения в потоковые классы. В обоих потоках надо заключить в synchronized блок операции инкремента и декремента.

В потоке IncThread:

```
for(int i=0; i<limit; i++)
    {
        synchronized(Treadtest.locker){
            Treadtest.counter++;
        }
    }
```

В потоке DecThread:

```
for(int i=0; i<limit; i++)
{
    synchronized(Treadtest.locker){
        Treadtest.counter--;
    }
}
```

Запустите теперь наше приложение с любым значением limit, хоть 1000000 хоть 100000000. Вы в любом случае получите итоговое значение counter, равное 0.

Рассмотрим другой пример, в котором несколько потоков будут работать с одним и тем же методом. В этом примере продемонстрируем применение synchronized методов.

Создайте новый проект и добавьте в него такой класс:

```
public class MyResource {
    long counter = 0;
    public void add(long value){
        this.counter += value;
    }
    long getCounter(){
        return this.counter;
    }
}
```

Метод add() будет вызываться несколькими потоками одновременно, и нам надо будет сделать так, чтобы потоки не мешали друг другу.

Теперь добавим в состав приложения потоковый класс:

```
public class MyThread extends Thread{
    protected MyResource counter = null;
```

```
    public MyThread(MyResource counter){
        this.counter = counter;
    }
    public void run() {
        for(int i=0; i<=100; i++){
           counter.add(i);
        }
    }
}
```

Этот класс содержит поле типа MyResource, которое инициализируется через конструктор. Затем в потоковом методе run() вызывается метод add().

Создадим и запустим в main() два потока, а после завершения их работы проверим значение поля counter:

```
public static void main(String[] args) {
    MyResource counter = new MyResource();
    Thread  t1 = new MyThread(counter);
    Thread  t2 = new MyThread(counter);
    t1.start();
    t2.start();
    try {
        t1.join();
        t2.join();
    } catch (InterruptedException ex) {
        ex.printStackTrace();
        Logger.getLogger(Testsunchro.class.getName()).
            log(Level.SEVERE, null, ex);
    }
    System.out.println("counter="+counter.getCounter());
}
```

Обратите внимание, что оба потока работают с одним и тем же объектом класса MyResource одновременно.

Каждый из потоков увеличивает значение counter на 5050. Запустите приложение несколько раз, и вы увидите, что итоговое значение counter всякий раз будет другим. Если бы два потока выполнились без помех, то значение counter было бы равно 10100, но у нас получаются другие результаты. Это говорит о том, что потоки мешают друг другу. Справедливости ради надо отметить, что иногда проскакивают и правильные результаты, но это происходит из-за небольшого количества итераций в цикле потокового метода.

Может ли нам помочь в этом случае спецификатор synchronized? Да. Для устранения проблемы с рассинхронизацией надо просто привести метод add() к такому виду:

```
public synchronized void add(long value){
    this.counter += value;
}
```

Теперь этот метод является синхронизированным, т.е. одновременно может выполняться только одним потоком для конкретного объекта. А у нас оба потока работают с одним и тем же объектом. Запустите приложение и убедитесь в том, что сейчас итоговое значение поля counter всегда будет равно 10100, т.е. каждый поток без помех выполнил свою работу. В нашем случае синхронизация методов add() в двух потоках выполняется по объекту counter, который этот метод вызывает. Ведь оба наших потока работают с одним и тем же объектом counter. Спецификатор synchronized можно использовать и для static методов. В таком случае

синхронизация метода выполняется по свойству class того класса, в котором определен метод.

Вы увидели примеры синхронизации многопоточных приложений с помощью synchronized методов и блоков. Такая синхронизация хороша при контроле доступа к разделяемому ресурсу. С другой стороны, часто бывают ситуации, когда необходимо создать более сложное взаимодействие потоков. Например, иногда надо сделать так, чтобы один поток начинал работу сразу после выполнения работы другим потоком. Для таких случаев в Java есть другие средства синхронизации, которые мы сейчас рассмотрим.

wait() и notify()

Рассмотрим такую задачу. Есть текстовый файл. Нам надо выбирать из этого файла строки по какому-либо критерию и записывать выбранные строки в другой файл. Работу по чтению и записи надо выполнять в дополнительных потоках. Попробуем создать такое решение. Один поток читает файл, выбирает требуемую строку и заносит эту строку в String переменную line. Второй поток записывает эту переменную line в выходной файл. Надо сделать так, чтобы первый поток, занеся прочитанную строку в переменную line, сообщал второму потоку о том, что тот может выполнять запись. А второй поток, выполнив запись, должен сообщать первому, что переменную line уже можно затирать и заносить туда значение следующей строки. Таким образом оба потока должны работать, пока не выберут и не запишут в выходной файл все требуемые строки. Чтобы решить эту

задачу, рассмотрим методы wait(), notify() и notifyAll(). Они определены в классе Object.

Прежде всего, надо запомнить, что эти методы обязательно должны вызываться в synchronized блоке от имени объекта, по которому выполняется синхронизация. Это объект lock, описанный в предыдущем разделе. Тогда мы сделали этот объект открытым и статическим, чтобы он был доступен всем потокам. Так поступать не обязательно. Такой объект можно передавать в потоки, например, через конструктор. Мы так и сделаем в нашем примере. Прежде чем перейти к коду примера, рассмотрим применение этих методов схематически.

Один поток выполняет какую-то работу. Второй поток должен начать свою часть работы, когда первый поток сообщит ему об этом. Сейчас второй поток находится в режиме ожидания и не выполняется. Первый поток завершает свою часть работы и должен сообщить об этом второму потоку, чтобы тот проснулся начал работать. Первый поток при этом должен перейти в режим ожидания, чтобы подождать завершения работы второго потока. Затем цикл повторяется. Оба потока работают с одним и тем же объектом locker.

В цикле while(true) в первом потоке:

```
synchronized(locker){
    // A piece of job is being done;
    locker.notify()  // this method wake up
                     // the second thread
    locker.wait()    // this method makes to fall
                     // asleep the current thread
}
```

В цикле while(true) во втором потоке:

```
synchronized(locker){
    locker.wait() // this method makes the current
                  // thread to fall asleep,
                  // the current thread now can be
                  // awakened ONLY when some
                  // other thread calls the notify()
                  // method on the same locker object
    //A piece of job is being done;
    locker.notify() // this method wake up the first
                    //thread
}
```

Разберем этот код. Еще раз важно отметить, что объект locker в обоих потоках — это один и тот же объект. Оба потока работают в циклах. Вы понимаете, что речь идет о циклах в потоковых методах run(). Сначала отметьте, что второй поток при выполнении первой итерации выполнит метод locker.wait(). Вызов этого метода приведет к тому, что второй поток перейдет в режим ожидания, и ему не будет выделяться процессорное время, пока он не проснется. Запомните, что поток, вызвавший метод wait(), **не может разбудить себя сам.** Он активируется только в том случае, когда какой-либо другой поток вызовет метод notify() или notifyAll() на том же объекте блокировки, на котором был вызван метод wait(). Что в это время делает первый поток? Он начал свою итерацию и выполняет какую-то работу. Когда эта работа будет завершена, первый поток вызовет метод notify(), разбудив тем самым второй поток. А затем первый поток вызовет метод wait() и сам уснет, ожидая завершения второго потока. Когда второй

поток проснется, он начнет выполнять код, расположенный после вызова метод wait(), то есть будет выполнять свою часть работы. Завершив свою работу, второй поток вызовет метод notify(), чтобы разбудить первый поток, а сам перейдет к следующей итерации, в начале которой снова выполнит вызов метода wait() и уснет, ожидая, когда его разбудит первый поток. В таком режиме потоки будут выполняться, пока их работа не будет завершена каким-либо образом.

Чем отличаются методы notify() и notifyAll()? Предположим, что в данный момент времени в состоянии сна находятся несколько потоков, а не один, как в рассмотренном примере. Если активный поток вызывает метод notify(), то этот метод разбудит только один из ожидающих (спящих) потоков. Причем управлять тем, какой именно из ожидающих потоков будет активирован, возможности нет. Будет активирован какой-то поток, случайным образом выбранный системой. Если же активный поток вызовет метод notifyAll(), то будут активированы все ожидающие потоки. Однако при этом только один из всех активированных потоков захватит синхронизированный блок и начнет выполняться, остальные снова перейдут в режим ожидания. И снова у нас нет возможности управлять тем, какой именно из нескольких активированных потоков захватит синхронизированный блок.

Перейдем теперь к нашему примеру. Создайте новый проект и приведите в нем главный класс к такому виду, как показано ниже. Обратите внимание, что объект синхронизации сейчас не открытый и статический, а передается каждому потоку в конструктор. А вот строка, через

которую происходит чтение и запись файлов — открытое и статическое поле класса.

```
public class Testsynchro {
    public static String line="";
    public static void main(String[] args) {
        Object locker=new Object(); //synchronizing
                                    //object
        MyReader reader = new MyReader("lines.
                          txt",locker);
        MyWriter writer = new MyWriter("lines_out.
                          txt",locker);
        Thread t1 = new Thread(reader); // thread
                                        // reader
        Thread t2 = new Thread(writer); // thread
                                        // writer
        t1.setDaemon(true);
        t2.setDaemon(true);

        t2.start();
        try {
            Thread.sleep(500);
        } catch (InterruptedException ex) {
           Logger.getLogger(Testsynchro.
           class.getName()).
           log(Level.SEVERE, null, ex);
        }
        t1.start();
        try { // wait for both threads to complete
            t1.join();
            t2.join();
        } catch (InterruptedException ex) {
            ex.printStackTrace();
            Logger.getLogger(Testsynchro.
            class.getName()).
            log(Level.SEVERE, null, ex);
        }
    }
```

Вы видите, что входным файлом является текстовый документ с именем «*lines.txt*». У меня в этом файле располагаются такие строки:

```
The first line.
This is the second line in the file.
Here is the third line.
And the fourth one.
And this is the fifth.
Line number 6.
Next odd line number 7.
```

Задача потоков в нашем примере состоит в том, чтобы выбирать из входного файла строки с нечетными номерами и только их записывать в выходной файл «*lines_out.txt*».

Теперь добавьте в состав проекта поток, выполняющий чтение входного файл и записывающий нечетные строки в статическую строку line:

```
public class MyReader implements Runnable{
    FileReader fr = null;
    Object locker;
    public MyReader(String filePath, Object locker)
    {
        try {
            this.fr = new FileReader(filePath);
            this.locker=locker;
        } catch (FileNotFoundException ex) {
            Logger.getLogger(MyReader.class.getName()).
            log(Level.SEVERE, null, ex);
        }
    }
    @Override
    public void run() {
```

```
        int lineCounter=0;
        String str="";
        BufferedReader br = new BufferedReader(fr);
        try {
            while((str = br.readLine()) != null) {
                System.out.println("Reader:"+str);
                if( (lineCounter++) % 2 == 0) {
                    synchronized(locker){
                        Testsynchro.line=str;
                        locker.notify();
                        locker.wait();
                    }
                }
            }
            synchronized(locker){
                Testsynchro.line="exit";
                locker.notify();
                locker.wait();
            }
        } catch (IOException ex) {
            Logger.getLogger(MyReader.class.getName()).
            log(Level.SEVERE, null, ex);
        } catch (InterruptedException ex) {
            Logger.getLogger(MyReader.class.getName()).
            log(Level.SEVERE, null, ex);
        } finally {
            try {
                fr.close();
            } catch (IOException ex) {
                Logger.getLogger(MyReader.class.
                getName()).
                log(Level.SEVERE, null, ex);
            }
        }
    }
}
```

Обратите особое внимание на то, что происходит в synchronized блоках. Записывается нечетная прочитанная строка в статическую переменную line. Затем вызовом метода notify() выполняется активация потока писателя, после чего поток читателя засыпает. Во втором synchronized выполняются те же действия, но в переменную line записывается слово «exit», по которому будет завершено выполнение потока писателя. Читающий поток остановится, завершив чтение входного файла.

Теперь добавьте в проект класс писатель:

```
public class MyWriter implements Runnable{
    FileWriter fw = null;
    Object locker;

    public MyWriter(String filePath, Object locker)
    {
        try {
            this.fw = new FileWriter(filePath,true);
            this.locker=locker;
        } catch (IOException ex) {
            Logger.getLogger(MyWriter.class.getName()).
            log(Level.SEVERE, null, ex);
        }
    }

    @Override
    public void run() {
        synchronized(locker){
            while(Testsynchro.line != "exit")
            {
                try {
                    locker.wait();
                    if(Testsynchro.line != "exit"){
```

```
                        fw.write(Testsynchro.line +
                        System.getProperty("line.
                        separator"));
                    }
                    System.out.println("*** Written
                            line:" + Testsynchro.line);
                    locker.notify();
                } catch (IOException ex) {
                    Logger.getLogger(MyWriter. class.
                            getName()).
                    log(Level.SEVERE, null, ex);
                } catch (InterruptedException ex) {
                    Logger.getLogger(MyWriter.class.
                            getName()).
                    log(Level.SEVERE, null, ex);
                }
            }
        }
        try {
            fw.close();
        } catch (IOException ex) {
            Logger.getLogger(MyWriter.class.getName()).
            log(Level.SEVERE, null, ex);
        }
    }
}
```

Выполните приложение несколько раз. Каждый поток выводит в консольное окно описание своих действий, поэтому внимательно посмотрите на этот вывод. У меня получился такой вывод:

```
Reader:The first line.
*** Written line:The first line.
Reader:This is the second line in the file.
```

```
Reader:Here is the third line.
*** Written line:Here is the third line.
Reader:And the fourth one.
Reader:And this is the fifth.
*** Written line:And this is the fifth.
Reader:Line number 6.
Reader:Next odd line number 7.
*** Written line:Next odd line number 7.
*** Written line:exit
```

А в выходном файле после двух запусков приложения у меня находится такое содержимое:

```
The first line.
Here is the third line.
And this is the fifth.
Next odd line number 7.
The first line.
Here is the third line.
And this is the fifth.
Next odd line number 7.
```

Итак, поставленная перед нами задача решена благодаря использованию методов wait() и notify(). Прежде чем переходить к следующей теме, отметьте некоторые особенности использования методов wait(), notify() и notifyAll(). Возможна ситуация, при которой первый поток запишет строку из файла в переменную line до того, как второй поток будет находиться в режиме ожидания. При этом первый поток вызовом метода notify() никого не разбудит. Второй же поток, вызвав с опозданием метод wait(), не дождется, когда первый поток его разбудит — ведь первый поток тоже уже будет спать после «холостого»

вызова notify(). Чтобы этого не произошло, мы в методе main() первым запустили именно второй поток и дали ему фору в 500 миллисекунд. Еще запомните следующее: когда поток, захвативший synchronized блок, вызывает в этом блоке метод wait(), он тем самым освобождает этот блок и позволяет другим потокам захватывать его.

Semaphore

В предыдущем уроке мы с вами говорили о пакете java.util.concurrent, появившемся в Java 5. Тогда мы затронули синхронизированные коллекции, определенные в этом пакете. Сейчас нам интересен другой объект из java.util.concurrent, имеющий прямое отношение к синхронизации потоков. Это класс Semaphore. Полное описание этого класса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html.

Иногда в приложении возникает необходимость ограничить число потоков, одновременно работающих с каким-то ресурсом. В такой ситуации Semaphore будет самым лучшим решением. Создадим приложение, в котором потоковый метод сможет одновременно выполняться только пятью разными потоками. Число пять взято совершенно произвольно. Как мы выполним поставленную задачу? В объекте потокового класса мы создадим объект класса Semaphor, указав ему при создании максимальное допустимое число потоков, которые могут одновременно выполнять метод run(). Это число можно рассматривать как количество свободных мест для потоков. Каждый новый поток, входя в метод run(), будет вызывать у се-

мафора метод acquire(), уменьшая тем самым на единицу количество свободных мест. Когда будет запущено пять потоков, количество свободных мест у семафора станет равным нулю и он перестанет допускать к методу run() другие потоки. Когда какой-то из пяти потоков, уже выполняющих метод run(), завершит свою работу, он вызовет у семафора метод release(), увеличив на единицу количество свободных мест. После этого семафор допустит в метод run() один из ожидающих потоков. Так будет продолжаться до тех пор, пока все потоки не выполнят свою работу. При этом в каждый момент времени в методе run() будет выполняться не более пяти потоков. Semaphore часто сравнивают с охранником в баре, которых следит за тем, чтобы в зале было не больше определенного числа посетителей, и когда зал заполнен, он впускает новых только после того, как кто-то уйдет и освободит место.

Подобное поведение потокового класса можно, конечно же, реализовать и без Semaphore, но это вряд ли будет эффективнее и уж точно не будет так просто. Желаемое поведение потокового класса с Semaphore мы получаем вызовом всего двух методов.

Создайте новое приложение и добавьте в его состав такой потоковый класс:

```
public class MySemaphore implements Runnable{
    Semaphore sem = new Semaphore(5); //5 is number
                                      //of free rooms
    int counter;
    public MySemaphore(int c){
        this.counter=c; //just a random value to
                        //simulate a job
    }
```

```
    @Override
    public void run() {
    try {
        sem.acquire(); //decrease number of free
                       //rooms by 1
        System.out.println(Thread.currentThread().
               getName()+ " is working... "+
               this.counter);
        Thread.currentThread().sleep(counter);
        System.out.println(Thread.currentThread().
               getName()+" is finished!");
        sem.release(); //increase number of free
                       //rooms by 1
        } catch (InterruptedException ex) {
            Logger.getLogger(MySemaphore.
               class.getName()).
               log(Level.SEVERE, null, ex);
        }
    }
}
```

В методе main() организуем создание 20 потоков этого класса и запустим их друг за другом:

```
Random r = new Random();
MySemaphore ms=new MySemaphore(r.nextInt(2000)+1000);
for(int i=0;i<20; i++){
    Thread t=new Thread(ms);
    t.start();
}
```

Вывод этого кода будет таким:

```
Thread-0 is working... 2432
Thread-1 is working... 2432
```

```
Thread-2 is working... 2432
Thread-3 is working... 2432
Thread-4 is working... 2432
Thread-2 is finished!
Thread-5 is working... 2432
Thread-0 is finished!
Thread-6 is working... 2432
Thread-1 is finished!
Thread-7 is working... 2432
Thread-3 is finished!
Thread-4 is finished!
Thread-8 is working... 2432
Thread-9 is working... 2432
Thread-5 is finished!
Thread-10 is working... 2432
Thread-6 is finished!
Thread-12 is working... 2432
Thread-7 is finished!
Thread-11 is working... 2432
Thread-8 is finished!
Thread-9 is finished!
Thread-13 is working... 2432
Thread-14 is working... 2432
Thread-10 is finished!
Thread-15 is working... 2432
Thread-12 is finished!
Thread-16 is working... 2432
Thread-11 is finished!
Thread-17 is working... 2432
Thread-13 is finished!
Thread-18 is working... 2432
Thread-14 is finished!
Thread-19 is working... 2432
Thread-15 is finished!
Thread-16 is finished!
Thread-17 is finished!
Thread-18 is finished!
Thread-19 is finished!
```

Внимательно рассмотрите этот листинг. Друг за другом запустились первые пять потоков. Затем добавление потоков приостановилось, потому что семафор не позволяет большему количеству потоков получить доступ к потоковому методу. И только когда один из пяти потоков завершил работу в методе run(), его место занял следующий из ожидающих. И так далее. Очень красивой получилась ситуация после запуска потока Thread-7: работу одновременно завершили потоки Thread-3 и Thread-4, а их места заняли сразу два потока, Thread-8 и Thread-9.

В этом примере мы создали новый поток с помощью объекта класса MySemaphore, который наследует интерфейсу Runnable. Вы должны помнить, что в Java можно использовать анонимные классы для создания таких объектов. Посмотрим, как может выглядеть решение этой задачи без создания отдельного класса MySemaphore.

Закомментируйте в методе main() то, что было написано для предыдущего примера, и вставьте взамен такой код:

```
Runnable task = new Runnable() {
    Semaphore sem = new Semaphore(5);
    Random r = new Random();
    int counter = r.nextInt(2000)+500;
    @Override
    public void run() {
        try {
            sem.acquire();
            System.out.println(Thread.currentThread().
               getName()+ " is working... "+this.counter);
            Thread.currentThread().sleep(counter);
            System.out.println(Thread.currentThread().
               getName()+ " is finished!");
            sem.release();
```

```
            } catch (InterruptedException ex) {
                Logger.getLogger(MySemaphore.class.
                    getName()).log(Level.SEVERE, null, ex);
            }
        }
};
for(int i=0;i<20; i++){
    new Thread(task).start();
}
```

Запустите приложение и убедитесь, что поведение потоков не изменилось. Этот код выглядит компактнее первого варианта и не требует создания в приложении нового класса. Вы, конечно же, обратили внимание, что в последнем примере мы совсем не используем класс MySemaphore. Вместо него мы просто создали анонимный объект, который выполняет такие же функции, что и класс MySemaphore.

Executors

До сих пор всякий раз, когда нам был нужен новый поток, мы создавали его с помощью инструкции new. При этом мы либо использовали прямое наследование от Thread, либо Runnable объект. Однако такой способ создания новых потоков имеет известные недостатки, связанные с тем, что создание нового потока с помощью new — очень ресурсоемкая операция. Если вы помните, в .NET Framework есть полезный класс ThreadPool, который создает новые потоки, запускает их, но не удаляет после того, как потоки завершат выполнение. ThreadPool хранит созданные объекты потоков, завершивших работу. И когда от приложения поступает запрос на новый поток, он активирует такой объект потока, экономя, таким образом, на создании нового объекта. Есть ли что-то похожее в Java? Да. В том же пакете java.util.concurrent есть интерфейс ExecutorService, позволяющий выполнять подобные действия. Этот интерфейс предоставляет фабричные методы для эффективного создания новых потоков. Полное описание этого интерфейса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html.

Познакомимся с ExecutorService поближе. Вы можете создавать объекты, производные от этого интерфейса разными способами. Если вам надо создать потоки, чтобы запустить в них асинхронное выполнение каких-либо задач, вы можете поступить одним из следующих способов:

```
ExecutorService executor1 =
        Executors.newSingleThreadExecutor();
ExecutorService executor2 =
        Executors.newFixedThreadPool(10);
ExecutorService executor3 =
        Executors.newScheduledThreadPool(10);
```

Здесь для создания новых потоков используются статические методы фабричного класса Executors. Теперь использование созданных объектов executor позволит нам запускать созданные потоки. Если executor создан методом newSingleThreadExecutor(), он создает один поток. Этому потоку можно назначить для выполнения много задач, но выполняться они будут последовательно, друг за другом. Если же executor создан методом newFixedThreadPool(), то он создает столько потоков, сколько указано в его параметре. Все эти потоки будут выполняться параллельно. А если executor создан методом newScheduledThreadPool(), то он позволяет запускать потоки по расписанию.

Каким способом можно использовать потоки, предоставляемые ExecutorService? Для запуска созданных потоком можно использовать целый ряд методов: execute(), submit(), invokeAny(), invokeAll(). Рассмотрим использование некоторых из них сейчас, а остальных — в следующем разделе урока, после того, как познакомимся с интерфейсом Callable.

Рассмотрим использование метода execute() с объектом Runnable в качестве параметра. Этот метод создает поток и асинхронно запускает его метод run(). Создайте новый проект и вставьте в его метод main() такой код:

```
ExecutorService executor =
        Executors.newSingleThreadExecutor();
executor. execute(() -> {
    String threadName = Thread.currentThread().getName();
    try {
        Thread.currentThread().sleep(5000);
    } catch (InterruptedException ex) {
        Logger.getLogger(Testsynchro.class.getName()).
        log(Level.SEVERE, null, ex);
    }
    System.out.println("Hello from " + threadName);
});
executor. execute(() -> {
    String threadName = Thread.currentThread().getName();

    System.out.println("Hello again from " + threadName);
});
executor.shutdown();
```

Мы создали объект ExecutorService с именем executor. Для создания использовали метод newSingleThreadExecutor(). Это означает, что для нас был создан только один поток. Затем мы дважды вызвали метод execute(), передавая ему в качестве параметра лямбда выражение, инициализирующее объект Runnable. В первом случае поток выводит фразу «*Hello from*» и свое имя, во втором — фразу «*Hello again from*» и свое имя. Обратите внимание, что первый поток делает паузу на 5 секунд перед выводом своей фразы. Это сделано для того, чтобы было видно, что второй поток не начнет выполнение, пока не выполнится первый. Затем мы закрываем сервис методом shutdown().

Запустите приложение и проследите, как оно себя поведет. Сначала возникнет пауза в 5 секунд, затем в консольное

окно будет выведено сообщение из первого потокового метода, а затем — из второго. Обратите внимание, что имя потока одно и то же. Это значит, что единственный поток, созданный методом newSingleThreadExecutor(), может выполнять много асинхронных задач, но последовательно, одну за другой. У меня вывод этого кода получился таким:

```
Hello from pool-1-thread-1
Hello again from pool-1-thread-1
```

Отметьте, что не существует способа получить результат выполнения метода run(), запущенного таким образом. О возможности получать результаты выполнения из другого потока поговорим в следующем разделе урока.

Для выполнения потоков можно также использовать метод submit(), который принимает в качестве параметра объект Runnable и асинхронно запускает его метод run(), но при этом возвращает объект типа Future. В классе Future определен метод get(), позволяющий узнать статус завершения асинхронной задачи. Если этот метод вернет null, значит, асинхронная задача успешно завершилась. Замените код в методе main() на следующий код, использующий метод submit() и проверку статуса завершения асинхронной задачи. Обратите внимание, что в этом случае, для разнообразия, мы создаем объект Runnable с помощью анонимного класса:

```
ExecutorService executor =
        Executors.newSingleThreadExecutor();
Future future = executor.submit(new Runnable() {
```

```
    public void run() {
        String threadName = Thread.currentThread().
            getName();
        System.out.println("This code is deing
            executed "
            + "asynchronously in thread " + threadName);
        }
});
try {
    //returns null if the task completed successfuly
    if(future.get() == null)
        System.out.println("Success!");
} catch (InterruptedException ex) {
    Logger.getLogger(Testsynchro.class.getName()).
         log(Level.SEVERE, null, ex);
} catch (ExecutionException ex) {
    Logger.getLogger(Testsynchro.class.getName()).
         log(Level.SEVERE, null, ex);
}
executor.shutdown();
```

Запустите это приложение и посмотрите на результат выполнения. У меня вывод этого кода получился таким:

```
This code is deing executed asynchronously
in thread pool-1-thread-1
Success!
```

Если вам надо, чтобы какое-либо действие выполнялось в дополнительном потоке периодически, можете использовать ScheduledExecutorService. Например, так:

```
ScheduledExecutorService executor =
    Executors.newScheduledThreadPool(1);
Runnable task = new Runnable() {
```

```
    public void run() {
        System.out.println("Hello from "+Thread.
            currentThread().getName());
    }
};

ScheduledFuture sf =
    executor.scheduleAtFixedRate(task, 0, 2,
             TimeUnit.SECONDS);

try {
    Thread.currentThread().sleep(5000);
} catch (InterruptedException ex) {
    Logger.getLogger(Testsynchro.class.getName()).
        log(Level.SEVERE, null, ex);
}
sf.cancel(true);
```

Обратите внимание, что в этом случае мы используем для объекта executor тип ScheduledExecutorService. В этом типе определен метод scheduleAtFixedRate(), позволяющий выполнять задачу с заданной периодичностью. В первом параметре в этот метод передается задача в виде Runnable объекта, во втором — начальная задержка, в третьем — период повторения, а в последнем — единица измерения временных интервалов. В данном коде запускается задача task с нулевой начальной задержкой и с периодичностью две секунды. После запуска этой задачи главный поток останавливается на пять секунд, а затем просыпается и останавливает выполнение периодичной задачи вызовом метода cancel().

Запустите этот пример и посмотрите на результат выполнения. У меня вывод этого кода получился таким:

```
Hello from pool-1-thread-1
Hello from pool-1-thread-1
Hello from pool-1-thread-1
```

Задача в дополнительном потоке успела выполниться три раза, а затем главный поток (после паузы в 5 секунд) отменил ее.

Вы уже заметили такую особенность объекта ExecutorService: будучи запущенным любым из рассмотренных способов, он не прекращает работу, так как ожидает получения новых задач. Поэтому для прекращения работы его надо останавливать явно. Это можно сделать вызовом метода executor.shutdown(). Есть еще метод executor.shutdownNow(), позволяющий завершить выполнение сразу, но его вызов может завершить выполнение какого-либо потока преждевременно. Поэтому пользоваться этим методом надо аккуратно. Например, выполнять, так называемое «мягкое завершение»:

```
try {
    System.out.println("attempt to shutdown executor");
    executor.shutdown();
    //wait 3 seconds for completion
    executor.awaitTermination(3, TimeUnit.SECONDS);
}

catch (InterruptedException e) {
    System.err.println("tasks interrupted");
}

finally {
    //if service is not completed yet
    if (!executor.isTerminated()) {
```

```
            executor.shutdownNow();
            System.err.println("Make it to stop");
        }
        System.out.println("shutdown finished");
}
```

Остальные способы выполнения асинхронных задач с помощью ExecutorService используют интерфейс Callable, который мы рассмотрим в следующем разделе урока. Там же рассмотрим и остальные примеры.

Callable

Теперь поговорим подробнее об интерфейсе Callable, который мы затронули в предыдущем разделе. Этот интерфейс подобен интерфейсу Runnable в том, что позволяет выполнять какие-либо действия в дополнительном потоке, но при этом он позволяет получить из такого потока результат, возвращаемый потоковым методом. Полное описание этого интерфейса можете посмотреть на официальном сайте по адресу: http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html.

Метод, аналогичный методу run(), в интерфейсе Callable называется call() и может возвращать результат любого типа. Рассмотрим, как это все работает.

Создадим приложение, которое будет вычислять в дополнительном потоке сумму чисел из заданного диапазона и возвращать полученный результат. Для решения такой задачи надо создать класс, производный от интерфейса Callable, параметризированного типом результата, который мы хотим получить. В нашем случае — Callable<Integer>.

В созданное новое приложение добавьте класс, в котором будет асинхронно вычисляться сумма чисел:

```
public class MyCallable implements Callable<Integer>{
    int beg;
    int end;
    public MyCallable(int beg, int end){
        this.beg=beg;
        this.end=end;
    }
```

```
    @Override
    public Integer call() throws Exception {
        Integer sum=0;
        for(int i=this.beg; i<=this.end;i++){
            sum+=i;

        }
        return sum;
    }
}
```

Обратите внимание, что метод call() сейчас возвращает результат типа Integer, и сам базовый интерфейс Callable параметризирован этим типом. Теперь проверим, как этот класс работает. Добавим в main() такой код:

```
int num=1;  //number of threads
ThreadPoolExecutor executor =
    (ThreadPoolExecutor) Executors.
    newFixedThreadPool(num);

MyCallable mc  = new MyCallable (1, 10);
Future<Integer> result = executor.submit(mc);
try {
    System.out.println("Result is: " + result.get());
    System.out.println("And task completion status is" +
          result.isDone());
    executor.shutdown();
} catch (InterruptedException ex) {
    Logger.getLogger(TestCallable.class.getName()).
          log(Level.SEVERE, null, ex);
} catch (ExecutionException ex) {
    Logger.getLogger(TestCallable.class.
          getName()).log(Level.SEVERE, null, ex);
}
```

Запустите этот пример и посмотрите на результат выполнения. У меня вывод этого кода получился таким:

```
Result is: 55
And task completion status is true
```

Обсудим выполненную работу и полученные результаты. Используя фабричный метод newFixedThreadPool(num) класса Executors, мы создали объект executor. Для запуска дополнительного потока применяется метод submit(), который принимает параметр типа Callable. Поэтому мы создаем объект типа MyCallable и передаем его методу submit(). Создавая объект MyCallable, мы задали интервал от 1 до 10, планируя вычислить в потоке сумму чисел из этого диапазона. Вызывая метод submit(), мы принимаем возвращаемое значение типа Future<Integer>. Вы уже догадались, что возвращаемое значение метода call() передается в типе Future, параметризированном типом результата, в нашем случае — Future<Integer>. Получить из класса Future возвращенный результат можно вызовом метода get(), а вызовом метода isDone() можно узнать, завершился ли поток. Все это продемонстрировано в нашем примере.

Вы спрашиваете, почему мы использовали только один поток и можно ли запустить сразу несколько вычислений для разных диапазонов? Да, это можно сделать. Изменим наш пример так, чтобы сразу запускалось несколько потоков — например, 10.

Приведите код в методе main() к такому виду:

```
int num=10; //number of threads
ThreadPoolExecutor executor =
(ThreadPoolExecutor) Executors.newFixedThreadPool(num);
```

```
//create list to keep the results from all the threads
List<Future<Integer>> results = new ArrayList<>();
Random r = new Random();
//process all the 10 threads in the loop
for (int i=0; i<num; i++)
{
    int b=r.nextInt(10); //generate random interval
    int e=r.nextInt(100)+10; //
    System.out.println("Limits:"+b+" to "+e);
    MyCallable mc = new MyCallable (b,e); //create
                              //Callable
    Future<Integer> result = executor.submit(mc); //
                              //launch the thread
    results.add(result); //add received result in the list
}

System.out.println("Results: ");
try {
    //show the results from all the threads
    for(Future<Integer> result : results){
        System.out.println("Result is: " + result.get());
    }
    executor.shutdown();
    } catch (InterruptedException ex) {
    Logger.getLogger(TestCallable.class.getName()).
           log(Level.SEVERE, null, ex);
    } catch (ExecutionException ex) {
    Logger.getLogger(TestCallable.class.getName()).
           log(Level.SEVERE, null, ex);
    }
```

Прежде всего, мы создали List<Future<Integer>> results, чтобы собирать в нем возвращаемые результаты от всех потоков. Затем в цикле создали десять объектов Callable со случайными числовыми диапазонами и в этом же цикле запустили все десять потоков вызовом метода

submit(), сохраняя результат каждого потока в списке results. Затем в следующем цикле просто выводим все результаты. Обратите внимание, что в потоковом классе ничего изменять не надо. Запустите этот пример и посмотрите на результат выполнения. У меня вывод этого кода получился таким:

```
Limits: 0 to 33
Limits: 7 to 86
Limits: 5 to 28
Limits: 8 to 47
Limits: 8 to 63
Limits: 5 to 58
Limits: 7 to 13
Limits: 9 to 98
Limits: 4 to 86
Limits: 6 to 46

Results:
Result is: 561
Result is: 3720
Result is: 396
Result is: 1100
Result is: 1988
Result is: 1701
Result is: 70
Result is: 4815
Result is: 3735
Result is: 1066
```

Вы должны понимать, что данный пример схематичный. В реальном приложении надо было бы предпринять меры, позволяющие убедиться в том, что все потоки завершились, прежде чем выводить результаты. Обычно для такой проверки используется метод isDone().

В двух предыдущих примерах для активации потоков мы использовали метод submin(), а сейчас рассмотрим еще два метода, позволяющих запускать потоки сервиса ExecutorService. В следующих примерах нам надо будет видеть явно, какие потоки будут выполняться, поэтому добавьте в состав приложения еще один класс Callable:

```
public class MyStringCallable implements
       Callable<String> {
    private long wait;
    public MyStringCallable(int timeInMillis){
        this.wait=timeInMillis;
    }

    @Override
    public String call() throws Exception {
        Thread.sleep(wait);
        return Thread.currentThread().getName();
    }
}
```

А в методе main() замените код на такой:

```
int num=10; //number of threads
ThreadPoolExecutor executor =
     (ThreadPoolExecutor) Executors.
     newFixedThreadPool(num);
//collection to keep created callables
Set<Callable<String>> callables =
     new HashSet<Callable<String>>();
Random r = new Random();
//fill the collection in the loop
for (int i=0; i<num; i++){
    MyStringCallable mc  =
          new MyStringCallable(r.nextInt(1000));
```

```
    callables.add(mc);
}
String result="";
try {
    //launch a random thread out of 10
    result = executor.invokeAny(callables);
} catch (InterruptedException ex) {
    Logger.getLogger(TestCallable.class.getName()).
        log(Level.SEVERE, null, ex);
} catch (ExecutionException ex) {
    Logger.getLogger(TestCallable.class.getName()).
        log(Level.SEVERE, null, ex);
}

finally{
    executor.shutdown();
}
//show the name of the launched thread
System.out.println("Received from callable: " + result);
```

В этот раз мы создали коллекцию для хранения объектов типа Callable<String> и заполнили ее десятью объектами нашего класса MyStringCallable. Коллекция нам нужна потому, что мы будем использовать метод invokeAny(), который в качестве параметра принимает именно коллекцию объектов Callable и запускает какой-то один, случайно выбранный из потоков. Управлять тем, какой именно поток будет активирован, мы не можем. Дальше все уже вам известно. Запустите это приложение несколько раз и убедитесь, что всякий раз активироваться будет другой поток.

А теперь еще раз немного изменим код в методе main(), чтобы продемонстрировать использование еще одного

метода для вызова потоков. Снова изменим код в методе main() и приведем его к такому виду:

```
int num=10; //number of threads
ThreadPoolExecutor executor =
      (ThreadPoolExecutor) Executors.
      newFixedThreadPool(num);
//collection to keep created callables
Set<Callable<String>> callables =
      new HashSet<Callable<String>>();
Random r = new Random();
//fill the collection in the loop
for (int i=0; i<num; i++){
    MyStringCallable mc =
          new MyStringCallable(r.nextInt(1000));
    callables.add(mc);
}

//another collection to keep the results from all
//the threads
List<Future<String>> results=null;
try {
    results = executor.invokeAll(callables);
    for(Future<String> result:  results){
        System.out.println("Received from callable:
              " + result.get());
    }
} catch (InterruptedException ex) {
    Logger.getLogger(TestCallable.class.getName()).
        log(Level.SEVERE, null, ex);
} catch (ExecutionException ex) {
    Logger.getLogger(TestCallable.class.getName()).
        log(Level.SEVERE, null, ex);
}
finally{
    executor.shutdown();
}
```

Обратите внимание, что в этом примере для активации потоков вызывается метод invokeAll(), который активирует все потоки и принимает их результаты в коллекцию. У меня вывод этого кода выглядит так:

```
Received from callable: pool-1-thread-1
Received from callable: pool-1-thread-2
Received from callable: pool-1-thread-3
Received from callable: pool-1-thread-4
Received from callable: pool-1-thread-5
Received from callable: pool-1-thread-6
Received from callable: pool-1-thread-7
Received from callable: pool-1-thread-8
Received from callable: pool-1-thread-9
Received from callable: pool-1-thread-10
```

Это говорит о том, что все десять потоков были запущены и выполнились.

Сетевое взаимодействие

Теперь поговорим о том, какие инструменты Java предлагает разработчику для написания клиент-серверных приложений. Такие приложения состоят из нескольких частей, взаимодействующих между собой по сети. Нам необходимо рассмотреть, каким образом осуществляется взаимодействие приложений, выполняющихся на разных компьютерах в пределах локальной сети. Для начала надо ознакомиться с такими понятиями, как протоколы межсетевого взаимодействия и сокеты. Объекты, с которыми мы будем работать в этом разделе, определены в пакете java.net.

Протоколы

Рассмотрим правила или, скорее, наборы правил сетевого общения. Такие наборы правил называются протоколами. Пакет java.net обеспечивает поддержку двух основных протоколов: TCP и UDP. Рассмотрим особенности каждого из них.

Протокол **TCP** (*Transmission Control Protocol*) обеспечивает устойчивую связь между двумя приложениями. Принцип его работы такой. Если одно приложение–отправитель хочет отправить данные другому приложению, оно сначала проверяет, активно ли приложение–получатель. Для этого отправитель шлет получателю запрос вида «ты включен?» и ожидает ответа. Если приходит положительный ответ, отправитель шлет первую порцию данных и ожидает ответа об успешном их получении получателем. Если такой ответ получен, отправляется

вторая порция, и т.д. Если уведомления об успешном получении данных нет, отправитель делает определенную паузу и повторяет отправку тех же данных, снова ожидая уведомления об успешном получении. Так может повторяться несколько раз, и если данные отправить не удается — сеанс прерывается. Это очень грубое описание работы TCP, но оно помогает понять методику его работы. Какие основные характеристики этого протокола? Если сеанс связи завершился без ошибок, TCP гарантирует 100% успешную доставку ваших данных приложению–получателю. Это достоинство этого протокола. У TCP него есть и недостатки: низкая скорость передачи и возможность диалога только между двумя приложениями. Еще отметим, что при использовании TCP между отправителем и получателем можно создать поток ввода–вывода и пересылать данные с использованием потокового интерфейса. Это тоже является достоинством протокола.

Работа второго протокола **UDP** (*User Datagram Protocol*) сильно отличается от поведения TCP. Этот протокол оформляет пересылаемые данные особым образом (в виде дейтаграмм) и просто отправляет их в сеть. Каждая дейтаграмма знает, куда ей надо добраться, и делает это самостоятельно. Никакой установки канала связи или потока ввода-вывода между отправителем и получателем этот протокол не выполняет. Для него даже не важно, получены его данные или нет, и активен ли получатель вообще. Конечно же, ни о какой 100% гарантии доставки речь здесь не идет. Однако у UDP есть существенное преимущество — быстрота доставки. Кроме того, этот

протокол позволяет выполнять массовую доставку данных. Поэтому он очень широко применяется в различных приложениях.

Сокеты

Вы уже знаете, что каждый компьютер в сети имеет собственный адрес. Казалось бы, этого достаточно, чтобы идентифицировать компьютер в сети. Если вы задумаетесь, то поймете, что с сетью взаимодействует не компьютер в целом, а какое-либо приложение, запущенное на этом компьютере. К тому же, на одном компьютере может быть несколько приложений, работающих в сети. Поэтому идентифицировать в сети надо не просто компьютер, а конкретное приложение. Для этого используются сокеты. Можно сказать, что сокет представляет собой объединение адреса компьютера в сети и идентификатора конкретного приложения. Идентификаторы приложений называют портами.

Сокет = Адрес + Номер_порта

Каждому приложению, которое выходит в сеть присваивается номер порта в диапазоне от 1 до 65565. Причем отметьте, что у протоколов TCP и UDP у каждого свои 65565 портов и они друг другу не мешают.

TCP протокол

InetAddress

Для работы с адресами в Java используется класс InetAddress. Полное описание этого класса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/net/InetAddress.html.

Особенностью этого класса является то, что его объекты создаются не инструкцией new, а с помощью статических методов класса, например, так:

```
InetAddress addr1 = InetAddress.getByName("127.0.0.1");
InetAddress addr2 =
    InetAddress.getByName("www.google.com");
InetAddress addr3 = InetAddress.getLocalHost();
```

Объекты InetAddress очень часто используются в клиент–серверных приложениях и мы рассмотрим их применение в дальнейшем.

ServerSocket и Socket

Главным действующим звеном в любом сетевом взаимодействии является сокет. Давайте будем придерживаться правильной терминологии и называть отправителя клиентом, а получателя сервером. Хотя, это вовсе не означает, что сервер не умеет отправлять данные, а клиент не умеет их получать.

Так вот, поведения клиентского и серверного сокетов отличаются друг от друга. Задача клиентского сокета —

установить связь с серверным сокетом, организовать и отправить данные. Задача серверного сокета — перейти в режим прослушивания и ожидать поступления данных от клиентов. Говорят, что клиентский сокет — активный, а серверный — пассивный. Поэтому реализация сокетов отличается в зависимости от того, клиентский это сокет или серверный, и от того — для какого протокола эти сокеты должны использоваться.

Для работы с сокетами по протоколу TCP в Java используется несколько классов. Например, для создания серверного сокета — класс ServerSocket, а клиентского — Socket. Полное описание этих классов можете посмотреть на официальном сайте на страницах:

https://docs.oracle.com/javase/8/docs/api/java/net/ServerSocket.html.

http://docs.oracle.com/javase/8/docs/api/java/net/Socket.html

Рассмотрим создание серверного и клиентского сокетов с помощью указанных классов. Структура кода, создающего серверный сокет, может быть такой:

```
ServerSocket listener = new ServerSocket(12345);

while (true) {
    Socket client = null;
    while (client == null) {
        client = listener.accept();
        //communication with the client
        //takes place here
   }
}
```

Создается объект класса ServerSocket, которому при создании назначается номер порта. Это значит, что данное приложение сможет принимать по сети данные от других компьютеров, отправленные на адрес текущего компьютера и на номер порта 12345. Этот порт теперь закреплен за этим приложением и другие приложения на этом компьютере, работающие в сети по протоколу TCP, не должны забирать этот номер порта себе. Однако, другое приложение на этом компьютере может забрать себе порт с таким же номером, но для использования по протоколу UDP. После создания объекта серверного сокета начинается очень интересная работа — прослушивание или ожидание подключений от клиентов. Чаще всего прослушивание выполняется в бесконечном цикле, как на приведенной схеме. Обратите внимание на вызов метода accept(). Это — блокирующий метод. Другими словами этот цикл не будет выполняться с сумасшедшей скоростью, как может показаться с первого взгляда. На каждой итерации вызов метода accept() будет останавливать выполнение этого цикла и всего серверного приложения. Возобновление выполнения будет происходить только тогда, когда к нашему серверу подключится какой-нибудь клиент. Метод accept() вернет клиентский сокет подключившегося клиента, и приложение продолжит выполнение. Теперь, когда в наличии есть два сокета: серверный listener и клиентский client, серверное приложение может установить связь с подключившимся клиентом и начать прием–передачу данных. Вы должны понимать, что поскольку в работе серверного сокета

используется блокирующий метод, прослушивание лучше выполнять в дополнительном потоке, чтобы не блокировать все приложение.

Для создания клиентского сокета надо знать адрес сервера и номер порта, к которому привязан серверный сокет.

```
Socket client = new Socket("10.3.6.14", 12345);
```

Выполнив подключение к серверному сокету, клиент обычно отправляет ему данные и сразу переходит в режим прослушивания, ожидая ответные данные от сервера.

Однопоточный сервер

Рассмотрим простой пример использования сокетов по протоколу TCP, позволяющий обмениваться текстовыми сообщениями между сервером и клиентом. Этот пример будет состоять из двух приложений. Назовем их SimpleServer и SimpleClient. Создайте проект SimpleServer и приведите его главный класс к такому виду:

```
public class ServerChat {
    ServerSocket listener = null;
    Socket client = null;
    ObjectInputStream in = null;
    ObjectOutputStream out = null;
    int port = 8888;
    String msg = "";

    public static void main(String[] args) {
        ServerChat server = new ServerChat();
```

```
        while(true)
        {
            server.listen();
        }
    }

    void listen()
    {
        try {
            listener = new ServerSocket(port);
            System.out.println("Waiting for connection");
            //app is blocked by accept() call until
            //a client connection
            client = listener.accept();

            System.out.println("Client connected " +
                client.getInetAddress().getHostName());
            out = new ObjectOutputStream(client.
                getOutputStream());
            out.flush();

            in = new ObjectInputStream(client.
                getInputStream());

            //start dialog
            do{
                try {
                    msg = (String)in.readObject();
                    //echoed received message
                    System.out.println("client> " + msg);
                    DateFormat df = new
                    SimpleDateFormat("yyyy/MM/dd
                         HH:mm:ss");
                    Date d = new Date();
                    sendMessage("Message received;" +
                         df.format(d));
                    } catch (ClassNotFoundException ex) {
                        Logger.getLogger(ServerChat.
```

```
                            class.getName()).
                            log(Level.SEVERE, null, ex);
                    }
                }while(!msg.equals("exit"));
            } catch (IOException ex) {
                Logger.getLogger(ServerChat.
                    class.getName()).
                    log(Level.SEVERE, null, ex);
            }
            finally{
                try {
                    if(in != null)
                        in.close();
                    if(out != null)
                        out.close();
                    if(listener != null)
                        listener.close();
                } catch (IOException ex) {
                    Logger.getLogger(ServerChat.
                        class.getName()).
                        log(Level.SEVERE, null, ex);
                }
            }
        }

    void sendMessage(String msg)
    {
        try {
            out.writeObject(msg);
            out.flush();
        } catch (IOException ex) {
            Logger.getLogger(ServerChat.class.getName()).
            log(Level.SEVERE, null, ex);
        }
    }
}
```

Сначала мы создаем в классе ссылку listener для серверного сокета и ссылку client для того, чтобы принимать в нее сокеты подключающихся клиентов. Затем создаем входной и выходной потоки ввода–вывода, чтобы выполнять пересылку данных с использованием потоков. Всю логику диалога между сервером и подключающимися клиентами выносим в метод listen(), который вызываем в цикле в методе main(). В методе listen() создаем объект серверного сокета и вызываем метод accept(). После вызова accept() выполнение приложения приостанавливается, до тех пор, пока не подключится какой-нибудь клиент. При подключении клиента accept() возвращает сокет этого клиента, который мы забираем в объект client. Теперь с помощью этого объекта мы создаем потоки ввода вывода между нашим приложением и подключившимся клиентом. Для создания этих потоков мы используем методы client.getOutputStream() и client.getInputStream(). После создания потоков начинаем новый цикл, в котором и будет происходить диалог. Клиент будет присылать серверу текстовые сообщения. Сервер будет выводить полученные сообщения в консольное окно, и после каждого полученного сообщения отправлять клиенту ответ с уведомлением о получении сообщения и указанием точного момента времени получения сообщения. Отправка сообщения клиенту оформлена в виде отдельного метода sendMessage(). Диалог будет продолжаться до тех пор, пока клиент не пришлет сообщение «exit».

Теперь создайте клиентское приложение SimpleClient и приведите его к такому виду:

```
public class ClientChat {
    Socket client = null;
    ObjectInputStream in = null;
    ObjectOutputStream out = null;
    String msg = "";
    int port = 8888;

    public static void main(String[] args) {
        ClientChat chat = new ClientChat();
        chat.setConnection();
    }

    void setConnection()
    {
        try {
            client = new Socket("127.0.0.1", port);
            system.out.println("Connected to server");
            out = new ObjectOutputStream(client.
                  getOutputStream());
            out.flush();
            in = new ObjectInputStream(client.
                  getInputStream());

            do {
                msg = JOptionPane.showInputDialog(this,
                      "Enter your message:");
                if(msg == null)
                    msg = "";
                sendMessage(msg);
                if(!msg.equals("exit"))
                {
                    try {
                        msg = (String)in.readObject();
                    } catch (ClassNotFoundException ex) {
                        Logger.getLogger(ClientChat.
                           class.getName()).
                           log(Level.SEVERE, null, ex);
```

```
                    }
                    System.out.println("server> " + msg);
                }
            } while (!msg.equals("exit"));
        } catch (IOException ex) {
            Logger.getLogger(ClientChat.class.getName()).
                log(Level.SEVERE, null, ex);
        }
        finally
        {
            try {
                if(in != null)
                    in.close();
                if(out != null)
                    out.close();
                if(client != null)
                    client.close();
            } catch (IOException ex) {
                Logger.getLogger(ClientChat.
                     class.getName()).
                     log(Level.SEVERE, null, ex);
            }
        }
    }

    void sendMessage(String msg)
    {
        try {
            out.writeObject(msg);
            out.flush();
        } catch (IOException ex) {
            Logger.getLogger(ClientChat.class.getName()).
                 log(Level.SEVERE, null, ex);
        }
    }
}
```

В этом приложении выполняется много действий, подобных тем, что мы только-что рассмотрели. Логика диалога с сервером вынесена в метод setConnection(). Там выполняется создание объекта клиентского сокета и подключение к серверу. Затем создается диалоговое окно для ввода текстового сообщения и это сообщение отправляется серверу. После отправки клиент автоматически переходит в режим ожидания ответа от сервера: клиент вызывает метод in.readObject(). Этот метод тоже блокирующий, поэтому клиент останавливает выполнение, ожидая ответ от сервера. Именно поэтому мы сделали так, чтобы сервер после каждого принятого сообщения отправлял клиенту уведомление о получении и тем самым выводил клиентское приложение из состояния блокировки.

Запустите серверное приложение, оставьте его активным, а затем запустите клиентское приложение, отправьте несколько сообщений и завершите сеанс связи отправкой сообщения «exit».

В серверном приложении у меня получился такой вывод:

```
Waiting for connection
Client connected 127.0.0.1
client> Hi, server!
client> I send you my messages
client> I see that you received them all
client> exit

Waiting for connection
```

А в клиентском — такой:

```
Connected to server
server> Message received; 2017/05/26 17:17:32
server> Message received; 2017/05/26 17:17:51
server> Message received; 2017/05/26 17:18:14
```

Обратите внимание, что после окончания диалога с этим клиентом, сервер снова перешел в режим ожидания подключений от других клиентов. Об этом говорит вывод *Waiting for connection* после получения сообщения «exit».

Многопоточный сервер

Какой главный недостаток приведенного выше примера? Конечно же — это выполнение сервера в одном потоке. Такой сервер не может общаться с несколькими клиентами одновременно. Давайте перепишем сервер таким образом, чтобы он выполнял общение с каждым подключившимся клиентом в отдельном потоке. Для этого создадим новое серверное приложение MultiServer и добавим в состав этого приложения потоковый класс, в котором будет выполняться общение с подключившимися клиентами.

```
public class MyListener implements Runnable {
    Socket socket = null;
    ObjectOutputStream out = null;
    ObjectInputStream in = null;
    String msg = "";

    public MyListener(Socket s)
    {
        this.socket = s;
```

```
}

@Override
public void run() {
try {
    System.out.println("Client connected " +
    this.socket.getInetAddress().getHostName());
    out = new ObjectOutputStream(socket.
          getOutputStream());
    out.flush();
    in = new ObjectInputStream(socket.
          getInputStream());
    do {
        try {
            msg = (String)in.readObject();
            System.out.println("client> " +msg);
            if(msg.equals("exit"))
            {
                break;
            }
            DateFormat df =
                new SimpleDateFormat("yyyy/MM/dd
                HH:mm:ss");
            Date d = new Date();
            sendMessage("Message received " +
                df.format(d));
        } catch (ClassNotFoundException ex) {
                Logger.getLogger(MyListener.
                    class.getName()).
                    log(Level.SEVERE, null, ex);
        }
    } while (!msg.equals("exit"));
} catch (IOException ex) {
    Logger.getLogger(MyListener.class.getName()).
        log(Level.SEVERE, null, ex);
}
}
```

```
    void sendMessage(String m)
    {
        try {
            out.writeObject(m);
            out.flush();
        } catch (IOException ex) {
            Logger.getLogger(MyListener.
                 class.getName()).
                 log(Level.SEVERE, null, ex);
        }
    }
}
```

В главном классе этого приложения будет выполняться прослушивание клиентов. При подключении очередного клиента, приложение будет извлекать клиентский сокет клиента и передавать его в приведенный выше потоковый класс, в котором будет выполняться диалог с этим клиентом. Вставьте в главный класс такой код:

```
public class ChatServer {
    ServerSocket listener = null;
    Socket client = null;
    int maxCount = 10; //max number of clients
    int count = 0;     //current client number
    int port = 8888;

    public static void main(String[] args) {
        ChatServer chat = new ChatServer();
        chat.createConnection();
    }

    void createConnection()
    {
```

```
        try {
            listener = new ServerSocket(port, maxCount);
            while(count <= maxCount)
            {
                count++;
                client = listener.accept();
                MyListener ml = new MyListener(client);
                Thread t = new Thread(ml);
                t.setDaemon(true);
                t.start();
            }
        } catch (IOException ex) {
            Logger.getLogger(ChatServer.class.getName()).
               log(Level.SEVERE, null, ex);
        }
    }
}
```

Запустите новое серверное приложение и оставьте его активным. Теперь запустите несколько копий нашего клиентского приложения и поочередно отправляйте серверу сообщения от каждого клиента. Вы увидите, что сервер без проблем будет одновременно вести диалог с каждым клиентом.

Самое показательное приложение, демонстрирующее передачу данных по сети — это чат. Конечно же, наши примеры не являются чатами, позволяющими вести обмен текстовыми сообщениями. В реальном чате сервер должен выполнять очень много разнообразной работы, а не просто отвечать эхом на каждое поступившее сообщение. Он должен запоминать в коллекции всех подключившихся клиентов и пересылать сообщение каждого клиента, либо всем остальным клиентам, если

сообщение общедоступное, либо только некоторым, если сообщение приватное. Также сервер должен отслеживать подключение к чату новых клиентов и выход клиентов из чата, и сообщать об этом всем участникам чата. Обычно в таком приложении пересылается не просто строка текста, а специальный класс, инкапсулирующий в себе, само сообщение, отправителя, возможно, получателей и много чего еще. Такой класс может включать в себя методы, которые будут выполнять обработку связанных с сообщением событий. Это очень интересная задача, но мы не будем ее решать в рамках этого урока. Однако, мы все же создадим полнофункциональный чат, при знакомстве с протоколом UDP.

UDP протокол

DatagramPacket

При использовании UDP протокола, данные пересылаемые по сети, упаковываются специальным образом и называются дейтаграммами. Для создания дейтаграмм применяется класс DatagramPacket. Полное описание этого класса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/net/DatagramPacket.html.

Этот класс представляет собой обертку вокруг байтового массива. Такой массив является контейнером для пересылаемых данных. Вообще говоря, входящие и исходящие дейтаграммы отличаются по заполнению своих полей, хотя и те и другие являются объектами одного и того же класса DatagramPacket. Исходящая дейтаграмма содержит в своем байтовом массиве отправляемые данные, и, кроме этого, — адрес и номер порта получателя. Входящая дейтаграмма содержит пустой байтовый массив, в который будут приняты пришедшие по сети данные. Входящие дейтаграммы создаются таким конструктором:

```
public DatagramPacket(byte[] data, int length)
```

Исходящие дейтаграммы таким:

```
public DatagramPacket(byte[] data, int length,
       InetAddress host, int port)
```

здесь: host это адрес получателя, а port — его номер порта.

Запомните: байтовый массив data, переданный конструктору, хранится в дейтаграмме не по значению, а по ссылке. Если вы измените этот массив после передачи его в дейтаграмму, содержимое дейтаграммы изменится тоже.

Для получения и изменения данных в дейтаграмме существует ряд геттеров и сеттеров:

```
public InetAddress getAddress()
public int getPort()
public byte[] getData()
public int getLength()

public void setAddress(InetAddress host)
public void setPort(int port)
public void setData(byte buffer[])
public void setLength(int length)
```

Мы рассмотрим использование этого класса в наших примерах. Однако уже сейчас вы должны понимать, что любые пересылаемые по протоколу UDP данные, должны преобразовываться в байтовый массив, а после получения преобразовываться обратно к своему исходному виду. Размер пересылаемого пакета в UPD протокле ограничен 65535 байтами. Однако для непосредственно для данных в дейтаграмме отводится только 65,507 байт, поскольку 8 байт занимает UDP заголовок и 20 байт — IP заголовок.

DatagramSocket

При использования TCP протокола, для создания серверных и клиентских сокетов применяются разные классы. В случае использования UDP протокола для создания клиентских и серверных сокетов используется

один класс — DatagramSocket. Полное описание этого класса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/net/DatagramSocket.html.

Для создания клиентского сокета используется такой конструктор:

```
public DatagramSocket() throws SocketException
```

Отправка дейтаграммы может выглядеть так:

```
try {
    InetAddress ia = new InetAddress("10.3.60.112");
    int port = 19999;
    String str = "A text message";
    byte[] data = str.getBytes();
    DatagramPacket dp = new DatagramPacket(data,
         data.length, ia, port);
    DatagramSocket sender = new DatagramSocket();
    sender.send(dp);
}
catch (UnknownHostException ex) {
    System.err.println(ex);
}
catch (IOException ex) {
    System.err.println(ex);
}
```

Рассмотрим алгоритм этого примера.

1. Готовим информацию с координатами получателя (ia и port) и текстовое сообщение для отправки (str):

```
InetAddress ia = new InetAddress("10.3.60.112");
int port = 19999;
String str = "A text message";
```

2. Преобразовываем строку в байтовый массив:

```
byte[] data = str.getBytes();
```

3. Создаем исходящую дейтаграмму:

```
DatagramPacket dp = new DatagramPacket(data,
data.length, ia, port);
```

4. Создаем клиентский сокет:

```
DatagramSocket sender = new DatagramSocket();
```

5. Отправляем дейтаграмму:

```
sender.send(dp);
```

Для создания серверного сокета применяются такие конструкторы:

```
public DatagramSocket(int port) throws SocketException
public DatagramSocket(int port, InetAddress laddr)
       throws SocketException
```

Получение дейтаграммы может выглядеть так:

```
try {
    DatagramSocket ds = new DatagramSocket(19999);
    byte buffer = new byte[10240];
    DatagramPacket idp = new DatagramPacket(buffer,
                         buffer.length);
    ds.receive(idp);
    byte[] data = idp.getData();
    String s = new String(data, 0, data.getLength());
    System.out.println(s);
}
```

```
catch (IOException ex) {
    System.err.println(ex);
}
```

Рассмотрим алгоритм этого примера.

1. Создаем серверный сокет:

```
DatagramSocket ds = new DatagramSocket(19999);
```

2. Создаем байтовый массив для входящей дейтаграммы:

```
byte buffer = new byte[10240];
```

3. Создаем входящую дейтаграмму:

```
DatagramPacket idp = new DatagramPacket(buffer,
                     buffer.length);
```

4. Принимаем входящую дейтаграмму:

```
ds.receive(idp);
```

5. Извлекаем байтовый массив из пришедшей дейтаграммы:

```
byte[] data = idp.getData();
```

6. Извлекаем текстовые данные из полученного массива:

```
String s = new String(data, 0, data.getLength());
```

В рассмотренных примерах мы пересылали строку. Посмотрим, как можно пересылать произвольные объекты. Алгоритм использования сокетов и дейтаграмм

в этом случае будет точно таким же, как и в предыдущих примерах. Однако для преобразования объекта в байтовый массив перед отправкой и для преобразования байтового массива в объект после получения, надо будет использовать потоки ввода вывода.

Перед отправкой объект записывается в поток ByteArrayOutputStream с помощью потока обертки ObjectOutputStream. Затем поток легко преобразовывается в байтовый массив с помощью метода toByteArray().

```
class Fish implements Serializable {
    String name;
    double weight;
    double price;
    ...
}

Fish f = new Fish("salmon", 3, 170);

ByteArrayOutputStream bos = new
ByteArrayOutputStream(2048)
ObjectOutputStream oos = new ObjectOutputStream(bos);
oos.writeObject(f);

DatagramPacket packet = new DatagramPacket(bos.
                        toByteArray(), bos.size());
datagramSocket.send(packet);
```

При извлечении данных из дейтаграммы надо будет выполнить обратные преобразования. После извлечения байтового массива из дейтаграммы, он заносится в поток ByteArrayInputStream. Затем вокруг этого потока создается обертка ObjectInputStream. С помощью метода readObject() этого потока обертки выполняется преоб-

разование массива байт в объект типа Object, а затем — явное преобразование к требуемому типу.

```
try {
    DatagramSocket ds = new DatagramSocket(12345);
    byte buffer = new byte[2048];
    DatagramPacket idp = new DatagramPacket(buffer,
        buffer.length);
    ds.receive(idp);
    byte[] data = idp.getData();
    ByteArrayInputStream in =
        new ByteArrayInputStream(data);
    ObjectInputStream is = new ObjectInputStream(in);
    try {
        Fish f = (Fish) is.readObject();
        System.out.println("Fish object: "+ f);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}
catch (IOException ex) {
    System.err.println(ex);
}
```

Текстовый UDP чат

Напишем приложение, которое позволит обмениваться текстовыми сообщениями в пределах локальной сети. У этого приложения будет несколько особенностей. Во-первых, не будет отдельно серверной и клиентской частей приложения. Одно приложение будет отправлять и получать сообщения. Во-вторых, сообщения будут отправляться не конкретному компьютеру в локальной сети, а всем членам локальной сети. Такая рассылка называется *броадкаст*. По правде говоря, массовая рассылка здорово

грузит сеть и ее надо применять в редких случаях. У нас именно такой случай — мы учимся, поэтому нам можно использовать *броадкаст*. Приложение будет с графическим интерфейсом. Мы не будем приводить полный код приложения, он доступен в дополнении к уроку. В тексте урока мы приведем фрагменты когда, отвечающие за работу с дейтаграммами. Чтобы получить полное представление о работе этого приложения, его надо запускать в локальной сети. Я полагаю, что адрес вашей локальной сети начинается с 10.3.XXX.XXX, а остальная часть адреса нам не важна. Вы поймете это по дальнейшему изложению материала.

В главном классе приложения, который в нашем случае является фреймом, объявим несколько глобальных ссылок:

```
private String name=""; //user nickname
private DatagramSocket socket=null; //socket
private MyListener listener=null; //class receiver
public static InetAddress GROUP=null; // LAN address
public static int PORT=6711; //port number
```

В конструкторе этого класса мы инициализируем адрес:

```
try
{
    MyFrame.GROUP=InetAddress.getByName("10.3.255.255");
}catch (UnknownHostException ex)
{
    Logger.getLogger(MyFrame.class.getName()).
        log(Level.SEVERE, null, ex);
}
```

Обратите внимание, что мы указали не конкретный адрес, а групповой. Это значит, что сообщения отправляемые этим сокетом будут доступны всем компьютерам сети, адреса которых соответствуют этой группе.

Для входа в чат, наше приложение будет требовать от пользователя ввода имени. Когда имя будет введено, активируется соответствующая кнопка, в обработчике клика которой мы инициализируем сокет и передадим его в потоковый класс, который организует прием сообщений в дополнительном потоке.

```
try
{
    this.socket=new DatagramSocket(MyFrame.PORT);
} catch (IOException ex)
{
    Logger.getLogger(MyFrame.class.getName()).
           log(Level.SEVERE, null, ex);
}

this.listener=new MyListener(this.socket,this.
           taScreen);
Thread t=new Thread(listener);
t.setDaemon(true);
t.start();
```

Отправляемые сообщения заносятся в элемент JTextArea и отправляются нажатием на кнопку Send. Работа по упаковыванию текста в дейтаграмму и ее отправке, выполняется в обработчике клика этой кнопки:

```
String str=this.taMessage.getText().trim();
this.taMessage.setText("");
```

```
str=this.name+":""""""""""+str;
byte[] out=new byte[4096];

try
{
    out=str.getBytes("UTF-8");
    DatagramPacket pout = new
          DatagramPacket(out, out.length,
          MyFrame.GROUP, MyFrame.PORT);
    this.socket.send(pout);
} catch (UnsupportedEncodingException ex)
{
    Logger.getLogger(MyFrame.class.getName()).
          log(Level.SEVERE, null, ex);
} catch (IOException ex)
{
    Logger.getLogger(MyFrame.class.getName()).
           log(Level.SEVERE, null, ex);
}
```

Теперь рассмотрим, что происходит в классе MyListener, ответственном за получение сообщений. В потоковом методе, в бесконечном цикле выполняется прием дейтаграмм, извлечение из них текстовых сообщений и вывод этих сообщений в окно.

```
public class MyListener implements Runnable
{
    private DatagramSocket socket=null;
    private JTextArea ta=null;
    public boolean isAlive=true;

    public MyListener(DatagramSocket s,JTextArea a)
    {
        this.socket=s;
```

```
        this.ta=a;
    }

    @Override
    public void run()
    {
        System.out.println("Listening started!");
        byte[] buff=new byte[4096];
        String str="";
        DatagramPacket pin=null;

        while(this.isAlive)
        {
            buff=new byte[4096];
            pin=new DatagramPacket(buff,buff.length);

            try
            {
                this.socket.receive(pin);
            } catch (IOException ex)
            {
                Logger.getLogger(MyListener.
                    class.getName()).
                    log(Level.SEVERE, null, ex);
            }

            str=new String(pin.getData()).trim()+
                System.getProperty("line.separator");
            ta.append(str);
        }
        System.out.println("Listening is over");
    }
}
```

Разместите это приложение на нескольких компьютерах в локальной сети, убедившись, что в адресе вы

указали правильные данные именно вашей сети. Затем начинайте вводить сообщения независимо друг от друга. Сообщение каждого участника чата будет приниматься всеми остальными.

MulticastSocket

Этот класс представляет собой сокет для отправки и получения дейтаграмм по UDP протоколу. При этом MulticastSocket обладает способностью отправлять дейтаграммы сразу большому количеству клиентов, объединенных в группу. Такая группа клиентов представляет собой определенный адрес из специального диапазона адресов класса D. Этот диапазон включает в себя адреса 224.0.0.0 to 239.255.255.255 включительно. Однако адрес 224.0.0.0 зарезервирован и не должен использоваться для рассылки. Другими словами, MulticastSocket позволяет произвольному количеству клиентов подключиться к одному из адресов этого диапазона, объединив их в группу, и затем отправлять дейтаграммы всем членам такой группы. Полное описание этого класса можете посмотреть на официальном сайте по адресу: https://docs.oracle.com/javase/8/docs/api/java/net/MulticastSocket.html.

Рассмотрим приложение, которое позволяет создать такую группу и обмениваться текстовыми сообщениями между всеми ее членами. Это приложение также не будет состоять из отдельных серверной и клиентской частей. Функции отправки и получения сообщения будут реализованы в одном приложении с графическим интерфейсом. Окно приложения может выглядеть таким образом:

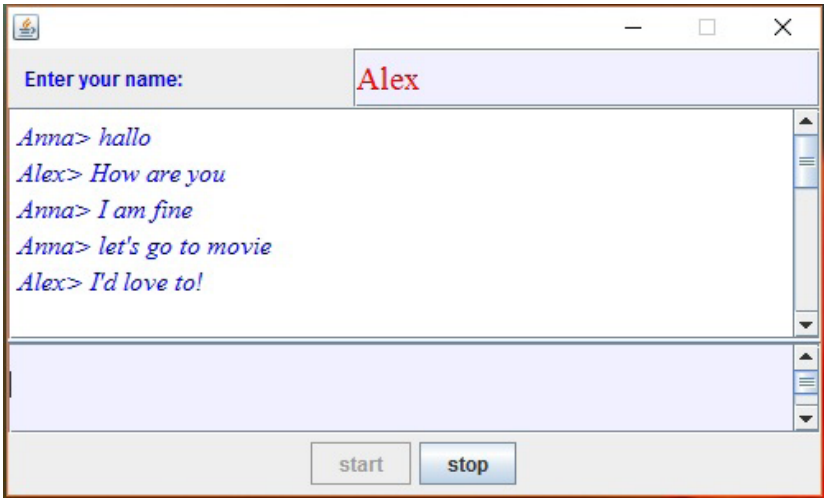

Рис. 2. Окно чата

Мы не будем в тексте урока приводить полный код приложения. Укажем только код для создания сокета, получения и отправки сообщений. Полный код всего приложения доступен в материалах к уроку. Кроме этого, вы должны понимать, что сможете обмениваться сообщениями только в том случае, если у вас есть прямой выход в Интернет.

Сначала создается ссылка для сокета и выполняется попытка инициализации адреса из указанного диапазона. Номер порта при этом произвольный.

```
MulticastSocket socket = null;
InetAddress group = null;
int port = 8888;
try {
    group = InetAddress.getByName("234.5.6.7");
} catch (UnknownHostException ex) {
}
```

Когда пользователь введет в окне чата свое имя, станет активной кнопка подключения у чату. При нажатии на эту кнопку инициализируется сокет и выполняется подключение к созданной адресной группе. Затем в отдельном потоке запускается прослушивание.

```
@Override
public void actionPerformed(ActionEvent ae) {
    try {
        socket = new MulticastSocket(port);
        socket.setReuseAddress(true);
        socket.joinGroup(group);
        MyListener ml = new MyListener();
        Thread t = new Thread(ml);
        t.setDaemon(true);
        t.start();
        } catch (IOException ex) {
        }
    }
});
```

В потоковом классе запускается бесконечный цикл приема дейтаграмм. Каждая пришедшая дейтаграмма приносит в своем байтовом массиве текстовое сообщение. Это сообщение извлекается, к нему добавляется символ перехода на новую строку и все это отображается в верхнем JTextArea.

```
public class MyListener implements Runnable {
    @Override
    public void run() {
        byte[] recv = new byte[4096];
        String msg = "";
        while(true)
```

```
        {
            DatagramPacket pin =
                new DatagramPacket(recv, recv.length);
            recv = new byte[4096];
            try {
                socket.receive(pin);
                msg = new String(pin.getData()).trim();
                msg += System.getProperty("line.
                     separator");
                chat.append(msg);
                } catch (IOException ex) {
                }
            }
}
```

Вы видите, что по логике работы это приложение практически является копией предыдущего чата. Отличия заключаются в использованных сокетах и в способах создания окон. В первом приложении использовался DatagramSocket, а в этом — MulticastSocket. Окно в первом приложении создавалось средствами NetBeans, а во втором — вручную. И еще одно замечание об этой программе. Если у вас нет прямого выхода в интернет, а вы находитесь в локальной сети за роутером, то данное приложение работать не будет, поскольку роутер не будет знать, куда пересылать поступающие сообщения внутри сети.

Домашнее задание

В качестве домашнего задания вам надо создать сетевое приложение.

Описание клиента

Клиентская часть приложения должна позволять пользователю выбрать на своем диске какой-нибудь графический файл и отправить его на сервер. Это должно быть приложение с графическим интерфейсом. Жестких требований к интерфейсу мы предъявлять не будем. Окно должно содержать кнопку, позволяющую выбрать файл, и кнопку, отправляющую файл на сервер. Будет полезным также создать возможность предварительного просмотра выбранной картинки. Структуру окна приложения продумайте и реализуйте самостоятельно. Выглядеть это может, например, так:

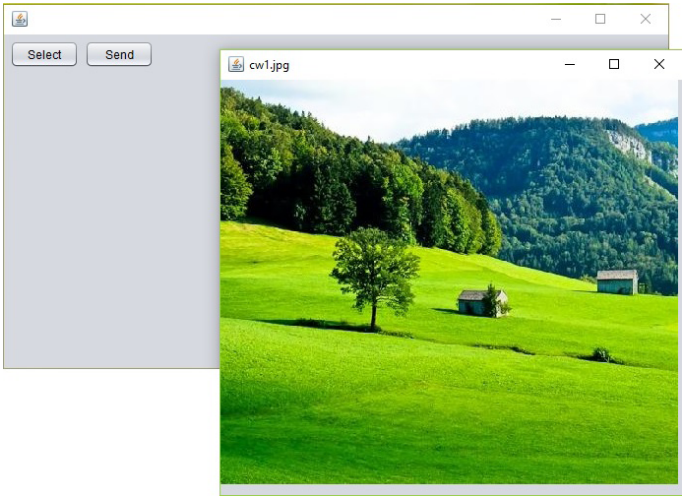

Рис. 3. Клиентская часть приложения

Описание сервера

Сервер должен получить графический файл от клиента и сохранить его в своей папке images, изменив этому файлу имя (например, добавлением временно́й метки). Сервер также должен иметь графический пользовательский интерфейс. В окне сервера в в элементе JListBox должны отображаться все файлы из папки images. При выборе в этом элементе какого-либо файла, выбранный файл должен отображаться в новом фрейме. Сервер должен выполнять загрузку файла от каждого клиента в отдельном потоке.

Подумайте хорошо, какой протокол будете использовать для передачи файлов по сети.