Глава 11

ПОТОКИ ВЫПОЛНЕНИЯ

Соседняя очередь всегда движется быстрее.

Наблюдение Этторе

Как только вы перейдете в другую очередь,
ваша начнет двигаться быстрее.

Наблюдение О'Брайена

Класс Thread и интерфейс Runnable

К большинству современных распределенных приложений (Rich Client) и веб-приложений (Thin Client) выдвигаются требования одновременной поддержки многих пользователей, каждому из которых выделяется отдельный поток, а также разделения и параллельной обработки информационных ресурсов. Потоки — средство, которое помогает организовать одновременное выполнение нескольких задач, каждой в независимом потоке. Потоки представляют собой экземпляры классов, каждый из которых запускается и функционирует самостоятельно, автономно (или относительно автономно) от главного потока выполнения программы. Существует два способа создания и запуска потока: на основе расширения класса **Thread** или реализации интерфейса **Runnable**.

// # 1 # расширение класса Thread # TalkThread.java

```
package by.bsu.threads;
public class TalkThread extends Thread {
        @Override
        public void run() {
                for (int i = 0; i < 10; i++) {
                        System.out.println("Talking");
                        try {
                                Thread.sleep(7); // остановка на 7 миллисекунд
                        } catch (InterruptedException e) {
                                System.err.print(e);
                        }
                }
        }
}
```

При реализации интерфейса **Runnable** необходимо определить его единственный абстрактный метод **run().** Например:

/ # 2 # реализация интерфейса Runnable # WalkRunnable.java # WalkTalk.java */*

```
package by.bsu.threads;
public class WalkRunnable implements Runnable {
          @Override
          public void run() {
                    for (int i = 0; i < 10; i++) {
                              System.out.println("Walking");
                              try {
                                        Thread.sleep(7);
                              } catch (InterruptedException e) {
                                        System.err.println(e);
                              }
                    }
          }
}
package by.bsu.threads;
public class WalkTalk {
          public static void main(String[ ] args) {
                    // новые объекты потоков
                    TalkThread talk = new TalkThread();
                    Thread walk = new Thread(new WalkRunnable());
                    // запуск потоков
                    talk.start();
                    walk.start();
                    // WalkRunnable w = new WalkRunnable(); // просто объект, не поток
                    // w.run(); или talk.run(); // выполнится метод, но поток не запустится!
          }
}
```

Запуск двух потоков для объектов классов **TalkThread** непосредственно и **WalkRunnable** через инициализацию экземпляра **Thread** приводит к выводу строк: **Talking Walking**. Порядок вывода, как правило, различен при нескольких запусках приложения.

Интерфейс **Runnable** не имеет метода **start(),** а только единственный метод **run().** Поэтому для запуска такого потока, как **WalkRunnable** следует создать экземпляр класса **Thread** с передачей экземпляра **WalkRunnable** его конструктору. Однако при прямом вызове метода **run()** поток не запустится, выполнится только тело самого метода.

Жизненный цикл потока

При выполнении программы объект класса **Thread** может быть в одном из четырех основных состояний: «новый», «работоспособный», «неработоспособный» и «пассивный». При создании потока он получает состояние «новый» (**NEW**) и не выполняется. Для перевода потока из состояния «новый» в состояние «работоспособный» (**RUNNABLE**) следует выполнить метод **start(),** который вызывает метод **run()** — основной метод потока.

Поток может находиться в одном из состояний, соответствующих элементам статически вложенного перечисления **Thread.State**:

NEW — поток создан, но еще не запущен;

RUNNABLE — поток выполняется;

BLOCKED — поток блокирован;

WAITING — поток ждет окончания работы другого потока;

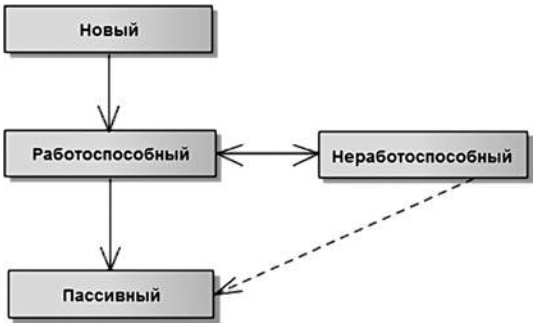

Рис. 11.1. *Состояния потока*

TIMED_WAITING — поток некоторое время ждет окончания другого потока;

TERMINATED — поток завершен.

Получить текущее значение состояния потока можно вызовом метода **getState().**

Поток переходит в состояние «неработоспособный» в режиме ожидания (**WAITING**) вызовом методов **join(), wait(),** suspend() (deprecated-метод) или методов ввода/вывода, которые предполагают задержку. Для задержки потока на некоторое время (в миллисекундах) можно перевести его в режим ожидания по времени (**TIMED_WAITING**) с помощью методов **yield(), sleep(long millis), join(long timeout)** и **wait(long timeout),** при выполнении которых может генерироваться прерывание **InterruptedException**. Вернуть потоку работоспособность после вызова метода suspend() можно методом resume() (deprecated-метод), а после вызова метода **wait()** — методами **notify()** или **notifyAll().** Поток переходит в «пассивное» состояние (**TERMINATED**), если вызваны методы **interrupt(),** stop() (deprecated-метод) или метод **run()** завершил выполнение, и запустить его повторно уже невозможно. После этого, чтобы запустить поток, необходимо создать новый объект потока. Метод **interrupt()** успешно завершает поток, если он находится в состоянии «работоспособный». Если же поток неработоспособен, например, находится в состоянии **TIMED_WAITING**, то метод инициирует исключение **InterruptedException**. Чтобы это не происходило, следует предварительно вызвать метод **isInterrupted(),** который проверит возможность завершения работы потока. При разработке не следует использовать методы принудительной остановки потока, так как возможны проблемы с закрытием ресурсов и другими внешними объектами.

Методы suspend(), resume() и stop() являются deprecated-методами и запрещены к использованию, так как они не являются в полной мере «потокобезопасными».

Управление приоритетами и группы потоков

Потоку можно назначить приоритет от **1** (константа **MIN_PRIORITY**) до **10** (**MAX_PRIORITY**) с помощью метода **setPriority(int prior).** Получить значение приоритета потока можно с помощью метода **getPriority().**

// # 3 # демонстрация приоритетов # PriorityRunner.java # PriorThread.java

```
package by.bsu.priority;
public class PriorThread extends Thread {
          public PriorThread(String name) {
                    super(name);
          }
          public void run() {
                    for (int i = 0; i < 71; i++) {
                              System.out.println(getName() + " " + i);
                              try {
                                        Thread.sleep(1); // попробовать sleep(0),sleep(10)
                              } catch (InterruptedException e) {
                                        System.err.print(e);
                              }
                    }
          }
}
package by.bsu.priority;
public class PriorityRunner {
          public static void main(String[ ] args) {
                    PriorThread min = new PriorThread("Min");
                    PriorThread max = new PriorThread("Max");
                    PriorThread norm = new PriorThread("Norm");
                    min.setPriority(Thread.MIN_PRIORITY); // 1
                    max.setPriority(Thread.MAX_PRIORITY); // 10
                    norm.setPriority(Thread.NORM_PRIORITY); // 5
                    min.start();
                    norm.start();
                    max.start();
          }
}
```

Поток с более высоким приоритетом в данном случае, как правило, монополизирует вывод на консоль.

Потоки объединяются в группы потоков. После создания потока нельзя изменить его принадлежность к группе.

```
ThreadGroup tg = new ThreadGroup("Группа потоков 1");
Thread t0 = new Thread(tg, "поток 0");
```

Все потоки, объединенные в группу, имеют одинаковый приоритет. Чтобы определить, к какой группе относится поток, следует вызвать метод **getThreadGroup()**.

Если поток до включения в группу имел приоритет выше приоритета группы потоков, то после включения значение его приоритета станет равным приоритету группы. Поток же со значением приоритета, более низким, чем приоритет группы после включения в оную, значения своего приоритета не изменит.

Управление потоками

Приостановить (задержать) выполнение потока можно с помощью метода **sleep(int millis)** класса **Thread**. Менее надежный альтернативный способ состоит в вызове метода **yield(),** который может сделать некоторую паузу и позволяет другим потокам начать выполнение своей задачи. Метод **join()** блокирует работу потока, в котором он вызван, до тех пор, пока не будет закончено выполнение вызывающего метод потока или не истечет время ожидания при обращении к методу **join(long timeout).**

// # 4 # задержка потока # JoinRunner.java

```
package by.bsu.management;
class JoinThread extends Thread {
          public JoinThread (String name) {
                    super(name);
          }
          public void run() {
                    String nameT = getName();
                    long timeout = 0;
                    System.out.println("Старт потока " + nameT);
                    try {
                              switch (nameT) {
                                  case "First":
                                        timeout = 5_000;
                                  break;
                                  case "Second":
                                         timeout = 1_000;
                              }
                              Thread.sleep(timeout);
                              System.out.println("завершение потока " + nameT);
                    } catch (InterruptedException e) {
                              e.printStackTrace();
                    }
          }
}
public class JoinRunner {
              static {
                    System.out.println("Старт потока main");
                    }
          public static void main(String[ ] args) {
                    JoinThread t1 = new JoinThread("First");
                    JoinThread t2 = new JoinThread("Second");
```

```
                t1.start();
                t2.start();
                try {
                    t1.join(); // поток main остановлен до окончания работы потока t1
                } catch (InterruptedException e) {
                     e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName()); // имя текущего потока
        }
}
```

Возможно, будет выведено:

Старт потока main
Старт потока First
Старт потока Second
завершение потока Second
завершение потока First
main

Несмотря на вызов метода **join()** для потока **t1**, поток **t2** будет работать, в отличие от потока **main**, который сможет продолжить свое выполнение только по завершении потока **t1**.

Если вместо метода **join()** без параметров использовать версию **join(long timeout),** то поток **main** будет остановлен только на указанный промежуток времени. При вызове **t1.join(500)** вывод будет другим:

Старт потока First
Старт потока Second
main
завершение потока Second
завершение потока First

Статический метод **currentThread()** возвращает ссылку на текущий поток, т. е. на поток, в котором данный метод был вызван.

Вызов статического метода **yield()** для исполняемого потока должен приводить к приостановке потока на некоторый квант времени, чтобы другие потоки могли выполнять свои действия. Например, в случае потока с высоким приоритетом после обработки части пакета данных, когда следующая еще не готова, стоит уступить часть времени другим потокам. Однако если требуется надежная остановка потока, то следует использовать его крайне осторожно или вообще применить другой способ.

// # 5 # задержка потока # YieldRunner.java

```
package by.bsu.yield;
public class YieldRunner {
        public static void main(String[ ] args) {
```

```
                new Thread() { // анонимный класс
                        public void run() {
                                System.out.println("старт потока 1");
                                Thread.yield();
                                System.out.println("завершение 1");
                        }
                }.start(); // запуск потока
                new Thread() {
                        public void run() {
                                System.out.println("старт потока 2");
                                System.out.println("завершение 2");
                        }
                }.start();
        }
}
```

В результате может быть выведено:

старт потока 1
старт потока 2
завершение 2
завершение 1

Активизация метода **yield()** в коде метода **run()** первого объекта потока приведет к тому, что, скорее всего, первый поток будет остановлен на некоторый квант времени, что даст возможность другому потоку запуститься и выполнить свой код.

Потоки-демоны

Потоки-демоны используются для работы в фоновом режиме вместе с программой, но не являются неотъемлемой частью логики программы. Если какой-либо процесс может выполняться на фоне работы основных потоков выполнения и его деятельность заключается в обслуживании основных потоков приложения, то такой процесс может быть запущен как поток-демон. С помощью метода **setDaemon(boolean value),** вызванного вновь созданным потоком до его запуска, можно определить поток-демон. Метод **boolean isDaemon()** позволяет определить, является ли указанный поток демоном или нет.

// # 6 # запуск и выполнение потока-демона # SimpleThread.java # DaemonRunner.java

```
package by.bsu.daemons;
public class SimpleThread extends Thread {
        public void run() {
                try {
                        if (isDaemon()) {
                                System.out.println("старт потока-демона");
                                Thread.sleep(10_000); // заменить параметр на 1
```

```
                    } else {
                        System.out.println("старт обычного потока");
                    }
                } catch (InterruptedException e) {
                    System.err.print(e);
                } finally {
                    if (!isDaemon()) {
                        System.out.println("завершение обычного потока");
                    } else {
                        System.out.println("завершение потока-демона");
                    }
                }
        }
}
package by.bsu.daemons;
public class DaemonRunner {
        public static void main(String[ ] args) {
                SimpleThread usual = new SimpleThread();
                SimpleThread daemon = new SimpleThread();
                daemon.setDaemon(true);
                daemon.start();
                usual.start();
                System.out.println("последний оператор main");
        }
}
```

В результате компиляции и запуска, возможно, будет выведено:

последний оператор main
старт потока-демона
старт обычного потока
завершение обычного потока

Поток-демон (из-за вызова метода **sleep(10000))** не успел завершить выполнение своего кода до завершения основного потока приложения, связанного с методом **main().** Базовое свойство потоков-демонов заключается в возможности основного потока приложения завершить выполнение потока-демона (в отличие от обычных потоков) с окончанием кода метода **main(),** не обращая внимания на то, что поток-демон еще работает. Если уменьшать время задержки потока-демона, то он может успеть завершить свое выполнение до окончания работы основного потока.

Потоки и исключения

В процессе функционирования потоки являются в общем случае независимыми друг от друга. Прямым следствием такой независимости будет корректное продолжение работы потока **main** после аварийной остановки запущенного из него потока после генерации исключения.

/ # 7 # генерация исключения в созданном потоке # ExceptThread.java # ExceptionThreadDemo.java */*

```
package by.bsu.thread;
public class ExceptThread extends Thread {
          public void run() {
                    boolean flag = true;
                    if (flag) {
                              throw new RuntimeException();
                    }
                    System.out.println("end of ExceptThread");
          }
}
package by.bsu.thread;
public class ExceptionThreadDemo {
          public static void main(String[ ] args) throws InterruptedException {
                    new ExceptThread().start();
                    Thread.sleep(1000);
                    System.out.println("end of main");
          }
}
```

Основной поток избавлен от необходимости обрабатывать исключения в порожденных потоках.

В данной ситуации верно и обратное: если основной поток прекратит свое выполнение из-за необработанного исключения, то это не скажется на работоспособности порожденного им потока.

/ # 8 # генерация исключения в потоке main # SimpleThread.java # ExceptionMainDemo.java */*

```
package by.bsu.thread;
public class SimpleThread extends Thread {
          public void run() {
                    try {
                              Thread.sleep(1000);
                    } catch (InterruptedException e) {
                              System.err.print(e);
                    }
                    System.out.println("end of SimpleThread");
          }
}
package by.bsu.thread;
public class ExceptionMainDemo {
          public static void main(String[ ] args) {
                    new SimpleThread().start();
                    System.out.println("end of main with exception");
                    throw new RuntimeException();
          }
}
```

Атомарные типы и модификатор volatile

Все данные приложения находятся в основном хранилище данных. При запуске нового потока создается копия хранилища и именно ею пользуется этот поток. Изменения, произведенные в копии, могут не сразу находить отражение в основном хранилище, и наоборот. Для получения актуального значения следует прибегнуть к синхронизации. Наиболее простым приемом будет объявление поля класса с модификатором **volatile**. Данный модификатор вынуждает потоки производить действия по фиксации изменений достаточно быстро. То есть другой заинтересованный поток, скорее всего, получит доступ к уже измененному значению. Для базовых типов до 32 бит этого достаточно. При использовании со ссылкой на объект — синхронизировано будет только значение самой ссылки, а не объект, на который она ссылается. Синхронизация ссылки будет эффективной в случае, если она указывает на перечисление, так как все элементы перечисления существуют в единственном экземпляре. Решением проблемы с доступом к одному экземпляру из разных потоков является блокирующая синхронизация. Модификатор **volatile** обеспечивает неблокирующую синхронизацию.

Существует целая группа классов пакета **java.util.concurrent.atomic**, обеспечивающая неблокирующую синхронизацию. Атомарные классы созданы для организации неблокирующих структур данных. Классы атомарных переменных **AtomicInteger**, **AtomicLong**, **AtomicReference** и др. расширяют нотацию **volatile** значений, полей и элементов массивов. Все атомарные классы являются изменяемыми в отличие от соответствующих им классов-оболочек. При реализации классов пакета использовались эффективные атомарные инструкции машинного уровня, которые доступны на современных процессорах. В некоторых ситуациях могут применяться варианты внутреннего блокирования.

Экземпляры классов, например, **AtomicInteger** и **AtomicReference,** предоставляют доступ и разного рода обновления к одной-единственной переменной соответствующего типа. Каждый класс также обеспечивает набор методов для этого типа. В частности, класс **AtomicInteger** — атомарные методы инкремента и декремента. Инструкции при доступе и обновлении атомарных переменных, в общем, следуют правилам для **volatile**.

Не следует классы атомарных переменных использовать как замену соответствующих классов-оболочек.

Пусть имеется некоторая торговая площадка, представленная классом **Market**, работающая в непрерывном режиме и информирующая о разнонаправленных изменениях биржевого индекса (поле **index** типа **AtomicLong**) дважды за один цикл с интервалом до 500 миллисекунд. Изменения поля **index** фиксируются методом **addAndGet(long delta)** атомарного добавления переданного значения к текущему.

/ # 9 # класс с атомарным полем # Market.java */*

```
package by.bsu.market;
import java.util.Random;
import java.util.concurrent.atomic.AtomicLong;
public class Market extends Thread {
          private AtomicLong index;
          public Market(AtomicLong index) {
                    this.index = index;
          }
          public AtomicLong getIndex() {
                    return index;
          }
          @Override
          public void run() {
                    Random random = new Random();
                    try {
                              while (true) {
                                        index.addAndGet(random.nextInt(10));
                                        Thread.sleep(random.nextInt(500));
                                        index.addAndGet(-1 * random.nextInt(10));
                                        Thread.sleep(random.nextInt(500));
                              }
                    } catch (InterruptedException e) {
                              e.printStackTrace();
                    }
          }
}
```

Имеется класс **Broker**, запрашивающий значение поля **index** с некоторым интервалом в миллисекундах.

/ # 10 # получатель значения атомарного поля # Broker.java */*

```
package by.bsu.market;
import java.util.Random;
public class Broker extends Thread {
          private Market market;
          private static final int PAUSE = 500; // in millis
          public Broker(Market market) {
                    this.market = market;
          }
          @Override
          public void run() {
                    try {
                              while (true) {
                                        System.out.println("Current index: " + market.getIndex());
                                        Thread.sleep(PAUSE);
                              }
```

```
                    } catch (InterruptedException e) {
                              e.printStackTrace();
                    }
          }
}
```

Количество экземпляров класса **Broker** может быть любым, и они постоянно с заданным интервалом запрашивают текущее значение **index**.

/ # 11 # запуск потоков изменения атомарного поля и его отслеживания несколькими потоками # AtomicDemo.java */*

```
package by.bsu.market;
import java.util.concurrent.atomic.AtomicLong;
public class AtomicDemo {
          private static final int NUMBER_BROKERS = 30;
          public static void main(String[ ] args) {
                    Market market = new Market(new AtomicLong(100));
                    market.start();
                    for (int i = 0; i < NUMBER_BROKERS; i++) {
                              new Broker(market).start();
                    }
          }
}
```

Атомарность поля обеспечивает получение экземплярами класса **Broker** идентичных текущих значений поля **index**.

Методы synchronized

Нередко возникает ситуация, когда несколько потоков имеют доступ к некоторому объекту, проще говоря, пытаются использовать общий ресурс и начинают мешать друг другу. Более того, они могут повредить этот общий ресурс. Например, когда два потока записывают информацию в файл/объект/поток. Для контролирования процесса записи может использоваться разделение ресурса с применением ключевого слова **synchronized**.

В качестве примера будет рассмотрен процесс записи информации в файл двумя конкурирующими потоками. В методе **main()** класса **SynchroRun** создаются два потока. В этом же методе создается экземпляр класса **Resource**, содержащий поле типа **FileWriter**, связанное с файлом на диске. Экземпляр **Resource** передается в качестве параметра обоим потокам. Первый поток записывает строку методом **writing()** в экземпляр класса **Resource**. Второй поток также пытается сделать запись строки в тот же самый объект **Resource**. Во избежание одновременной записи такие методы объявляются как **synchronized**. Синхронизированный метод изолирует объект, после чего он становится недоступным для других потоков. Изоляция снимается, когда поток полностью

выполнит соответствующий метод. Другой способ снятия изоляции — вызов метода **wait()** из изолированного метода — будет рассмотрен позже.

В примере продемонстрирован вариант синхронизации файла для защиты от одновременной записи информации в файл двумя различными потоками.

/ # 12 # синхронизация записи информации в файл # SyncThread.java # Resource.java # SynchroRun.java */*

```
package by.bsu.synch;
import java.io.*;
public class Resource {
          private FileWriter fileWriter;
          public Resource (String file) throws IOException {
                    // проверка наличия файла
                    fileWriter = new FileWriter(file, true);
          }
          public synchronized void writing(String str, int i) {
                    try {
                              fileWriter.append(str + i);
                              System.out.print(str + i);
                                        Thread.sleep((long)(Math.random() * 50));
                              fileWriter.append("->" + i + " ");
                              System.out.print("->" + i + " ");
                    } catch (IOException e) {
                              System.err.print("ошибка файла: " + e);
                    } catch (InterruptedException e) {
                              System.err.print("ошибка потока: " + e);
                    }
          }
          public void close() {
                    try {
                              fileWriter.close();
                    } catch (IOException e) {
                              System.err.print("ошибка закрытия файла: " + e);
                    }
          }
}
package by.bsu.synch;
public class SyncThread extends Thread {
          private Resource rs;
          public SyncThread(String name, Resource rs) {
                    super(name);
                    this.rs = rs;
          }
          public void run() {
                    for (int i = 0; i < 5; i++) {
                              rs.writing(getName(), i); // место срабатывания синхронизации
                    }
          }
}
```

```
package by.bsu.synch;
import java.io.IOException;
public class SynchroRun {
          public static void main(String[ ] args)  {
                    Resource s = null;
                    try {
                              s = new Resource ("data\\result.txt");
                              SyncThread t1 = new SyncThread("First", s);
                              SyncThread t2 = new SyncThread("Second", s);
                              t1.start();
                              t2.start();
                              t1.join();
                              t2.join();
                    } catch (IOException e) {
                              System.err.print("ошибка файла: " + e);
                    } catch (InterruptedException e) {
                              System.err.print("ошибка потока: " + e);
                    } finally {
                              s.close();
                    }
          }
}
```

В результате в файл будет, например, выведено:

First0->0 Second0->0 First1->1 Second1->1 First2->2 Second2->2 First3->3 Second3->3 First4->4 Second4->4

Код построен таким образом, что при отключении синхронизации метода **writing()** в случае его вызова одним потоком другой поток может вклиниться и произвести запись своей информации, несмотря на то, что метод не завершил запись, инициированную первым потоком.

Вывод в этом случае может быть, например, следующим:

First0Second0->0 Second1->0 First1->1 First2->1 Second2->2 First3->3 First4->2 Second3->3 Second4->4 ->4

Инструкция synchronized

Синхронизировать объект можно не только при помощи методов с соответствующим модификатором, но и при помощи синхронизированного блока кода. В этом случае происходит блокировка объекта, указанного в инструкции **synchronized**, и он становится недоступным для других синхронизированных методов и блоков. Такая синхронизация позволяет сузить область синхронизации, т. е. вывести за пределы синхронизации код, в ней не нуждающийся. Обычные методы на синхронизацию внимания не обращают, поэтому ответственность за грамотную блокировку объектов ложится на программиста.

/ # 13 # блокировка обьекта потоком # TwoThread.java */*

```
package by.bsu.instruction;
public class TwoThread {
          public static int counter = 0;
          public static void main(String args[ ]) {
                    final StringBuilder s = new StringBuilder();
                    new Thread() {
                              public void run() {
                                        synchronized (s) {
                                                  do {
                                                            s.append("A");
                                                            System.out.println(s);
                                                  try {
                                                            Thread.sleep(100);
                                                  } catch (InterruptedException e) {
                                                            System.err.print(e);
                                                  }
                                                } while (TwoThread.counter++ < 2);
                                        } // конец synchronized
                              }
                    }.start();
                    new Thread() {
                              public void run() {
                                        synchronized (s) {
                                                  while (TwoThread.counter++ < 6) {
                                                            s.append("B");
                                                            System.out.println(s);
                                                  }
                              } // конец synchronized
                                        }
                    }.start();
          }
}
```

В результате компиляции и запуска, скорее всего (например, второй поток может заблокировать объект первым), будет выведено:

A
AA
AAA
AAAB
AAABB
AAABBB

Один из потоков блокирует объект, и до тех пор, пока он не закончит выполнение блока синхронизации, в котором производится изменение значения объекта, ни один другой поток не может вызвать синхронизированный блок для этого объекта.

Если в коде убрать синхронизацию объекта **s**, то вывод будет другим, так как другой поток сможет получить доступ к объекту и изменить его раньше, чем первый закончит выполнение цикла.

Данный пример можно немного изменить для демонстрации «потокобезопасности» класса **StringBuffer** при вызове метода **append()** на синхронизированном экземпляре.

/ # 14 # потокобезопасность класса StringBuffer # BufferThread.java */*

```
package by.bsu.synchro;
public class BufferThread {
          static int counter = 0;
          static StringBuffer s = new StringBuffer(); // заменить на StringBuilder
          public static void main(String args[ ]) throws InterruptedException {
                    new Thread() {
                              public void run() {
                                        synchronized (s) {
                                                  while (BufferThread.counter++ < 3) {
                                                            s.append("A");
                                                            System.out.print("> " + counter + " ");
                                                            System.out.println(s);
                                                            Thread.sleep(500);
                                                  }
                                        } // конец synchronized-блока
                              }
                    }.start();
                    Thread.sleep(100);
                    while (BufferThread.counter++ < 6) {
                              System.out.print("< " + counter + " ");
                    // в этом месте поток main будет ждать освобождения блокировки объекта s
                              s.append("B");
                              System.out.println(s);
                    }
          }
}
```

Вызов метода на синхронизированном другим потоком объекте класса **StringBuffer** приведет к остановке текущего потока до тех пор, пока объект не будет разблокирован. То есть выведено будет следующее:

> 1 A
< 2 > 3 AA
AAB
< 5 AABB
< 6 AABBB

Если заменить **StringBuffer** на **StringBuilder**, то остановки потока на заблокированном объекте не произойдет и вывод будет таким:

> 1 A
< 2 AB
< 3 ABB
< 4 ABBB
< 5 ABBBB
< 6 ABBBBB

Монитор

Контроль за доступом к объекту-ресурсу обеспечивает понятие монитора. Монитор экземпляра может иметь только одного владельца. При попытке конкурирующего доступа к объекту, чей монитор имеет владельца, желающий заблокировать объект-ресурс поток должен подождать освобождения монитора этого объекта и только после этого завладеть им и начать использование объекта-ресурса. Каждый экземпляр любого класса имеет монитор. Методы **wait(), wait(long inmillis), notify(), notifyAll()** корректно срабатывают только на экземплярах, чей монитор уже кем-то захвачен. Статический метод захватывает монитор экземпляра класса **Class**, того класса, на котором он вызван. Существует в единственном экземпляре. Нестатический метод захватывает монитор экземпляра класса, на котором он вызван.

Методы wait(), notify() и notifyAll()

Эти методы никогда не переопределяются и используются только в исходном виде. Вызываются только внутри синхронизированного блока или метода на объекте, монитор которого захвачен текущим потоком. Попытка обращения к данным методам вне синхронизации или на несинхронизированном объекте (со свободным монитором) приводит к генерации исключительной ситуации **IllegalMonitorStateException**. В примере #15 рассмотрено взаимодействие методов **wait()** и **notify()** при освобождении и возврате блокировки в **synchronized** блоке. Эти методы используются для управления потоками в ситуации, когда необходимо задать определенную последовательность действий без повторного запуска потоков.

Метод **wait(),** вызванный внутри синхронизированного блока или метода, останавливает выполнение текущего потока и освобождает от блокировки захваченный объект. Возвратить блокировку объекта потоку можно вызовом метода **notify()** для одного потока или **notifyAll()** для всех потоков. Если ожидающих потоков несколько, то после вызова метода **notify()** невозможно определить, какой поток из ожидающих потоков заблокирует объект. Вызов может быть осуществлен только из другого потока, заблокировавшего в свою очередь тот же самый объект.

Проиллюстрировать работу указанных методов можно с помощью примера, когда инициализация полей и манипуляция их значениями производится в различных потоках.

/ # 15 # взаимодействие wait() и notify() # Payment.java # PaymentRunner.java */*

```
package by.bsu.synchro;
import java.util.Scanner;
public class Payment {
          private int amount;
          private boolean close;
          public int getAmount() {
                    return amount;
          }
          public boolean isClose() {
                    return close;
          }
          public synchronized void doPayment() {
                    try {
                              System.out.println("Start payment:");
                    while (amount <= 0) {
                              this.wait(); //  остановка потока и освобождение блокировки
                              // после возврата блокировки выполнение будет продолжено
                    }
                    // код выполнения платежа
                    close = true;
                    } catch (InterruptedException e) {
                              e.printStackTrace();
                    }
                    System.out.println("Payment is closed : " + close);
          }
          public void initAmount() {
                    Scanner scan = new Scanner(System.in);
                    amount = scan.nextInt();
          }
}
package by.bsu.synchro;
public class PaymentRunner {
          public static void main(String[] args) throws InterruptedException {
                    final Payment payment = new Payment();
                    new Thread() {
                              public void run() {
                                        payment.doPayment(); // вызов synchronized метода
                              }
                    }.start();
                    Thread.sleep(200);
                    synchronized (payment) { // 1-ый блок
                              System.out.println("Init amount:");
                              payment.initAmount();
```

```
                    payment.notify(); // уведомление о возврате блокировки
                }
                synchronized (payment) { // 2-ой блок
                    payment.wait(1_000);
                    System.out.println("ok");
                }
        }
}
```

В результате компиляции и запуска при вводе корректного значения для инициализации поля **amount** будет запущен процесс проведения платежа.

Задержки потоков методом **sleep()** используются для точной демонстрации последовательности действий, выполняемых потоками. Если же в коде приложения убрать все блоки синхронизации, а также вызовы методов **wait()** и **notify(),** то результатом вычислений, скорее всего, будет ноль, так как вычисление будет произведено до инициализации полей объекта.

Новые способы управления потоками

Java всегда предлагала широкие возможности для многопоточного программирования: потоки — это основа языка. Однако очень часто использование этих возможностей вызывало трудности: создать и отладить для корректного функционирования многопоточную программу достаточно сложно.

В версии Java SE 5 языка добавлен пакет **java.util.concurrent**, возможности классов которого обеспечивают более высокую производительность, масштабируемость, построение потокобезопасных блоков concurrent классов. Кроме этого усовершенствован вызов утилит синхронизации, добавлены классы семафоров и блокировок.

Возможности синхронизации существовали и ранее. Практически это означало, что синхронизированные экземпляры блокировались, хотя необходимо это было далеко не всегда. Например, поток, изменяющий одну часть объекта **Hashtable**, блокировал работу других потоков, которые хотели бы прочесть (даже не изменить) совсем другую часть этого объекта. Поэтому введение дополнительных возможностей, связанных с синхронизаций потоков и блокировкой ресурсов, довольно логично.

Ограниченно потокобезопасные (thread safe) коллекции и вспомогательные классы управления потоками сосредоточены в пакете **java.util.concurrent**. Среди них можно отметить:

- параллельные аналоги существующих синхронизированных классов-коллекций **ConcurrentHashMap**, **ConcurrentLinkedQueue** — эффективные аналоги **Hashtable** и **LinkedList**;
- классы **CopyOnWriteArrayList** и **CopyOnWriteArraySet**, копирующие свое содержимое при попытке его изменения, причем ранее полученный итератор будет корректно продолжать работать с исходным набором данных;

- блокирующие очереди **BlockingQueue** и **BlockingDeque**, гарантирующие остановку потока, запрашивающего элемент из пустой очереди до появления в ней элемента, доступного для извлечения, а также блокирующего поток, пытающийся вставить элемент в заполненную очередь, до тех пор, пока в очереди не освободится позиция;
- механизм управления заданиями, основанный на возможностях класса **Executor**, включающий организацию запуска пула потоков и службы их планирования;
- классы-барьеры синхронизации, такие как **CountDownLatch** (заставляет потоки ожидать завершения заданного числа операций, по окончании чего все ожидающие потоки освобождаются), **Semaphore** (предлагает потоку ожидать завершения действий в других потоках), **CyclicBarrier** (предлагает нескольким потокам ожидать момента, когда они все достигнут какой-либо точки, после чего барьер снимается), **Phaser** (барьер, контракт которого является расширением возможностей **CyclicBarrier**, а также частично согласовывается с возможностями **CountDownLatch**);
- класс **Exchanger** позволяет потокам обмениваться объектами.

В дополнение к перечисленному выше в пакете **java.util.concurrent.locks** содержатся дополнительные реализации моделей синхронизации потоков, а именно:

- интерфейс **Lock**, поддерживающий ограниченные ожидания снятия блокировки, прерываемые попытки блокировки, очереди блокировки и установку ожидания снятия нескольких блокировок посредством интерфейса **Condition**;
- класс семафор **ReentrantLock,** добавляющий ранее не существующую функциональность по отказу от попытки блокировки объекта с возможностью многократного повторения запроса на блокировку и отказа от нее;
- класс **ReentrantReadWriteLock** позволяет изменять объект только одному потоку, а читать в это время — нескольким.

Перечисление TimeUnit

Представляет различные единицы измерения времени. В **TimeUnit** реализован ряд методов по преобразованию между единицами измерения и по управлению операциями ожидания в потоках в этих единицах. Используется для информирования методов, работающих со временем, о том, как интерпретировать заданный параметр времени.

Перечисление **TimeUnit** может представлять время в семи размерностях-значениях: **NANOSECONDS**, **MICROSECONDS**, **MILLISECONDS**, **SECONDS**, **MINUTES**, **HOURS**, **DAYS**.

Кроме методов преобразования единиц времени представляют интерес методы управления потоками:

void timedWait(Object obj, long timeout) — выполняет метод **wait(long time)** для объекта **obj** класса **Object,** используя данные единицы измерения;

void timedJoin(Thread thread, long timeout) — выполняет метод **join(long time)** на потоке **thread**, используя данные единицы измерения.

void sleep(long timeout) — выполняет метод **sleep(long time)** класса **Thread**, используя данные единицы измерения.

Блокирующие очереди

Реализации интерфейсов **BlockingQueue** и **BlockingDeque** предлагают методы по добавлению/извлечению элементов с задержками, а именно:

void put(E e) — добавляет элемент в очередь. Если очередь заполнена, то ожидает, пока освободится место;

boolean offer(E e, long timeout, TimeUnit unit) — добавляет элемент в очередь. Если очередь заполнена, то ожидает время **timeout**, пока освободится место; если за это время место не освободилось, то возвращает **false**, не выполнив действия по добавлению;

boolean offer(E e) — добавляет элемент в очередь. Если очередь заполнена, то возвращает **false**, не выполнив действия по добавлению;

E take() — извлекает и удаляет элемент из очереди. Если очередь пуста, то ожидает, пока там появится элемент;

E poll(long timeout, TimeUnit unit) — извлекает и удаляет элемент из очереди. Если очередь пуста, то ожидает время **timeout**, пока там появится элемент, если за это время очередь так и осталась пуста, то возвращает **null**;

E poll() — извлекает и удаляет элемент из очереди. Если очередь пуста, то возвращает **null.**

Максимальный размер очереди должен быть задан при ее создании, а именно, все конструкторы класса **ArrayBlockingQueue** принимают в качестве параметра **capacity** длину очереди. Пусть объявлена очередь из пяти элементов. Изначально в ней размещены три элемента. В первом потоке производится попытка добавления трех элементов. Два добавятся успешно, а при попытке добавления третьего поток будет остановлен до появления свободного места в очереди. Только когда второй поток извлечет один элемент и освободит место, первый поток получит возможность добавить свой элемент.

/ # 16 # демонстрация возможностей блокирующей очереди # RunBlocking.java */*

```
package by.bsu.blocking;
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;
public class RunBlocking {
          public static void main(String[] args) {
                    final BlockingQueue<String> queue = new ArrayBlockingQueue<String>(2);
                    new Thread() {
```

```
                    public void run() {
                              for (int i = 1; i < 4; i++) {
                                 try {
                                     queue.put("Java" + i); // добавление 3-х
                                     System.out.println("Element " + i + " added");
                                 }catch (InterruptedException e) {
                                         e.printStackTrace();
                                   }
                              }
                    }
               }.start();
               new Thread() {
                         public void run() {
                            try {
                               Thread.sleep(1_000);
                               // извлечение одного
                               System.out.println("Element " + queue.take() + " took");
                            }catch (InterruptedException e) {
                                     e.printStackTrace();
                             }
                         }
               }.start();
          }
}
```

В результате будет выведено:

Element 1 added
Element 2 added
Element Java1 took
Element 3 added

Семафоры

Семафор позволяет управлять доступом к ресурсам или просто работой потоков на основе запрещений-разрешений. Семафор всегда устанавливается на предельное положительное число потоков, одновременное функционирование которых может быть разрешено. При превышении предельного числа все желающие работать потоки будут приостановлены до освобождения семафора одним из работающих по его разрешению потоков. Уменьшение счетчика доступа производится методами **void acquire()** и его оболочки **boolean tryAcquire()**. Оба метода занимают семафор, если он свободен. Если же семафор занят, то метод **tryAcquire()** возвращает ложь и пропускает поток дальше, что позволяет при необходимости отказаться от дальнейшей работы потоку, который не смог получить семафор. Метод **acquire()** при невозможности захвата семафора остановит поток до тех пор, пока хотя бы другой поток не освободит семафор. Метод **boolean tryAcquire(long timeout,TimeUnit unit)** возвращает

ложь, если время ожидания превышено, т. е. за указанное время поток не получил от семафора разрешение работать и пропускает поток дальше. Метод **release()** освобождает семафор и увеличивает счетчик на единицу. Простое надежное стандартное взаимодействие методов **acquire()** и **release()** демонстрирует следующий фрагмент.

/ # 17 # базовое решение при использовании семафора */*

```
public void run() {
          try {
                    semaphore.acquire();
                    // код использования защищаемого ресурса
          } catch (InterruptedException e) {
          } finally {
                    semaphore.release(); // освобождение семафора
          }
}
```

Методы **acquire()** и **release()** в общем случае могут и не вызываться в одном методе кода. Тогда за корректное и своевременное возвращение семафора будет ответственен разработчик. Метод **acquire()** не пропустит поток до тех пор, пока счетчик семафора имеет значение ноль.

Методы **acquire(), tryAquire()** и **release()** имеют перегруженную версию с параметром типа **int**. В такой метод можно передать число, на которое изменится значение счетчика семафора при успешном выполнении метода, в отличие от методов без параметров, которые всегда изменяют значение счетчика только на единицу.

Для демонстрации работы семафора предлагается задача о пуле ресурсов с ограниченным числом, в данном случае аудиоканалов, и значительно бóльшим числом клиентов, желающих воспользоваться одним из каналов. Каждый клиент получает доступ к каналу, причем пользоваться можно только одним каналом. Если все каналы заняты, то клиент ждет в течение заданного интервала времени. Если лимит ожидания превышен, генерируется исключение и клиент уходит, так и не воспользовавшись услугами пула.

Класс **ChannelPool** объявляет семафор и очередь из каналов. В методе **getResource()** производится запрос к семафору, и в случае успешного его прохождения метод извлекает из очереди канал и выдает его в качестве возвращаемого значения метода. Метод **returnResource()** добавляет экземпляр-канал к очереди на выдачу и освобождает семафор.

Реализация принципов пула предоставляет возможность повторного использования объектов в ситуациях, когда создание нового объекта — дорогостоящая процедура с точки зрения задействованных для этого ресурсов виртуальной машины. Поэтому при возможности следует объект после использования не уничтожать, а возвратить его в так называемый «пул объектов» для повторного использования. Данная стратегия широко используется при организации пула соединений с базой данных. Реализаций организации пулов существует

достаточно много с различающимися способами извлечения и возврата объектов, а также способа контроля за объектами и за заполняемостью пула. Поэтому выбрать какое-либо решение как абсолютно лучшее для всех случаев невозможно.

// # 18 # пул ресурсов # ChannelPool.java

```
package by.bsu.resource.pool;
import java.util.Queue;
import java.util.concurrent.Semaphore;
import java.util.concurrent.TimeUnit;
import java.util.LinkedList;
import by.bsu.resource.exception.ResourceException;
public class ChannelPool <T> {
          private final static int POOL_SIZE = 5; // размер пула
          private final Semaphore semaphore = new Semaphore(POOL_SIZE, true);
          private final Queue<T> resources = new LinkedList<T>();
          public ChannelPool(Queue<T> source) {
                    resources.addAll(source);
          }
          public T getResource(long maxWaitMillis) throws ResourceException {
                    try {
                              if (semaphore.tryAcquire(maxWaitMillis, TimeUnit.MILLISECONDS)) {
                                        T res = resources.poll();
                                        return res;
                              }
                    } catch (InterruptedException e) {
                              throw new ResourceException(e);
                    }
                    throw new ResourceException(":превышено время ожидания");
          }
          public void returnResource(T res) {
                    resources.add(res); // возвращение экземпляра в пул
                    semaphore.release();
          }
}
```

Класс **AudioChannel** предлагает простейшее описание канала и его использования.

// # 19 # канал — ресурс: обычный класс с некоторой информацией # AudioChannel.java

```
package by.bsu.resource.pool;
public class AudioChannel {
          private int channellId;
          public AudioChannel(int id) {
                    super();
                    this.channellId = id;
          }
          public int getChannellId() {
                    return channellId;
```

```
        }
        public void setChannellId(int id) {
                this.channellId = id;
        }
        public void using() {
                try {
                        // использование канала
                        Thread.sleep(new java.util.Random().nextInt(500));
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

Класс **ResourceException** желателен в такого рода задачах, чтобы точно описать возникающую проблему при работе ресурса, используемого конкурирующими потоками.

// # 20 # исключение, информирующее о сбое в поставке ресурса # ResourceException.java

```
package by..resource.exception;
public class bsu ResourceException extends Exception {
        public ResourceException() {
                super();
        }
        public ResourceException(String message, Throwable cause) {
                super(message, cause);
        }
        public ResourceException(String message) {
                super(message);
        }
        public ResourceException(Throwable cause) {
                super(cause);
        }
}
```

Класс **Client** представляет поток, запрашивающий ресурс из пула, использующий его некоторое время и возвращающий его обратно в пул.

// # 21 # поток, работающий с ресурсом # Client.java

```
package by.bsu.resource.pool;
import by.bsu.resource.exception.ResourceException;
public class Client extends Thread {
        private boolean reading = false;
        private ChannelPool<AudioChannel> pool;
        public Client (ChannelPool<AudioChannel> pool) {
                this.pool = pool;
        }
        public void run() {
```

```
                AudioChannel channel = null;
                try {
                        channel = pool.getResource(500); // изменить на 100
                        reading = true;
                        System.out.println("Channel Client #" + this.getId()
                                + " took channel #" + channel.getChannellId());
                        channel.using();
                } catch (ResourceException e) {
                        System.out.println("Client #" + this.getId() + " lost ->"
                                + e.getMessage());
                } finally {
                        if (channel != null) {
                                reading = false;
                        System.out.println("Channel Client #" + this.getId() + " : "
                                + channel.getChannellId() + " channel released");
                                pool.returnResource(channel);
                        }
                }
        }
        public boolean isReading() {
                return reading;
        }
}
```

Класс **Runner** демонстрирует работу пула ресурсов аудиоканалов. При заполнении очереди каналов в данном решении необходимо следить, чтобы число каналов, передаваемых списком в конструктор класса **ChannelPool**, совпадало со значением константы **POOL_SIZE** этого же класса. Константа используется для инициализации семафора и при большем или меньшем размерах передаваемого списка возникают коллизии, которые, кстати, есть смысл спровоцировать и разобраться в причинах и следствиях.

// # 22 # запуск и использование пула # Runner.java

```
package by.bsu.resource.main;
import java.util.LinkedList;
import by.bsu.resource.pool.AudioChannel;
import by.bsu.resource.pool.ChannelPool;
import by.bsu.resource.pool.Client;
public class Runner {
        public static void main(String[] args) {
                LinkedList<AudioChannel> list = new LinkedList<AudioChannel>() {
                        {
                                this.add(new AudioChannel(771));
                                this.add(new AudioChannel(883));
                                this.add(new AudioChannel(550));
                                this.add(new AudioChannel(337));
                                this.add(new AudioChannel(442));
                        }
```

```
                    };
                    ChannelPool<AudioChannel> pool = new ChannelPool<>(list);
                    for (int i = 0; i < 20; i++) {
                              new Client(pool).start();
                    }
          }
}
```

Результатом может быть вывод:

Channel Client #8 took channel #771
Channel Client #10 took channel #550
Channel Client #12 took channel #337
Channel Client #14 took channel #442
Channel Client #9 took channel #883
Channel Client #9 : 883 channel released
Channel Client #16 took channel #883
Channel Client #12 : 337 channel released
Channel Client #18 took channel #337
Channel Client #10 : 550 channel released
Channel Client #11 took channel #550
Channel Client #11 : 550 channel released
Channel Client #13 took channel #550
Channel Client #18 : 337 channel released
Channel Client #15 took channel #337
Channel Client #14 : 442 channel released
Channel Client #17 took channel #442
Channel Client #8 : 771 channel released
Channel Client #20 took channel #771
Client #19 lost ->:превышено время ожидания
Client #26 lost ->:превышено время ожидания
Client #24 lost ->:превышено время ожидания
Client #22 lost ->:превышено время ожидания
Client #23 lost ->:превышено время ожидания
Client #25 lost ->:превышено время ожидания
Client #27 lost ->:превышено время ожидания
Client #21 lost ->:превышено время ожидания
Channel Client #16 : 883 channel released
Channel Client #13 : 550 channel released
Channel Client #17 : 442 channel released
Channel Client #15 : 337 channel released
Channel Client #20 : 771 channel released

Барьеры

Многие задачи могут быть разделены на подзадачи и выполняться параллельно. По достижении некоторой данной точки всеми параллельными потоками подводится итог и определяется общий результат. Если стоит задача задержать заданное число потоков до достижения ими определенной точки синхронизации, то используются классы-барьеры. После того, как все потоки достигли этой самой точки, они будут разблокированы и могут продолжать выполнение. Класс **CyclicBarrier** определяет минимальное число потоков, которое может быть остановлено барьером. Кроме этого барьер сам может быть проинициализирован потоком, который будет запускаться при снятии барьера. Методы **int await()** и **int await(long timeout, TimeUnit unit)** останавливают поток, использующий барьер до тех пор, пока число потоков достигнет заданного числа в классе-барьере. Метод **await()** возвращает порядковый номер достижения потоком барьерной точки. Метод **boolean isBroken()** проверяет состояние барьера. Метод **reset()** сбрасывает состояние барьера к моменту инициализации. Метод **int getNumberWaiting()** позволяет определить число ожидаемых барьером потоков до его снятия. Экземпляр **CyclicBarrier** можно использовать повторно.

Процесс проведения аукциона подразумевает корректное использование класса **CyclicBarrier**. Класс **Auction** определяет список конкурирующих предложений от клиентов и размер барьера. Чтобы приложение работало корректно, необходимо, чтобы размер списка совпадал со значением константы **BIDS_NUMBER**. Барьер инициализируется потоком определения победителя торгов, который запустится после того, как все предложения будут объявлены. Если потоков будет запущено больше чем размер барьера, то «лишние» предложения могут быть не учтены при вычислении победителя, если же потоков будет меньше, то приложение окажется в состоянии deadlock. Для предотвращения подобных ситуаций следует использовать метод **await()** с параметрами.

// # 23 # определение барьера и действия по его окончании # Auction.java

```
package by.bsu.auction;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.concurrent.CyclicBarrier;
public class Auction {
  private ArrayList<Bid> bids;
  private CyclicBarrier barrier;
  public final int BIDS_NUMBER = 5;
  public Auction() {
    this.bids = new ArrayList<Bid>();
      this.barrier = new CyclicBarrier(this.BIDS_NUMBER, new Runnable() {
        public void run() {
          Bid winner = Auction.this.defineWinner();
```

```
            System.out.println("Bid #" + winner.getBidId() + ", price:" + winner.getPrice() + " win!");
          }
        });
      }
      public CyclicBarrier getBarrier() {
        return barrier;
      }
      public boolean add(Bid e) {
        return bids.add(e);
      }
      public Bid defineWinner() {
          return Collections.max(bids, new Comparator<Bid>() {
                              @Override
                              public int compare(Bid ob1, Bid ob2) {
                                        return ob1.getPrice() - ob2.getPrice();
                              }
          });
     }
}
```

Класс **Bid** определяет предложение клиента на аукционе и запрашивает барьер, после которого клиент либо заплатит за лот, либо будет продолжать работать дальше.

// # 24 # поток, использующий барьер # Bid.java

```
package by.bsu.auction;
import java.util.Random;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
public class Bid extends Thread {
          private Integer bidId;
          private int price;
          private CyclicBarrier barrier;
          public Bid(int id, int price, CyclicBarrier barrier) {
                    this.bidId = id;
                    this.price = price;
                    this.barrier = barrier;
          }
          public Integer getBidId() {
                    return bidId;
          }
          public int getPrice() {
                    return price;
          }
          @Override
          public void run() {
                    try {
                              System.out.println("Client " + this.bidId + " specifies a price.");
                              Thread.sleep(new Random().nextInt(3000)); // время на раздумье
                              // определение уровня повышения цены
```

```
                    int delta = new Random().nextInt(50);
                    price += delta;
                    System.out.println("Bid " + this.bidId + " : " + price);
                    this.barrier.await(); // остановка у барьера
                    System.out.println("Continue to work..."); // проверить кто выиграл
                                        // и оплатить в случае победы ставки
                } catch (BrokenBarrierException e) {
                    e.printStackTrace();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
        }
}
```

// # 25 # инициализация аукциона и его запуск # AuctionRunner.java

```
package by.bsu.auction;
import java.util.Random;
public class AuctionRunner {
        public static void main(String[ ] args) {
                Auction auction = new Auction();
                int startPrice = new Random().nextInt(100);
                for (int i = 0; i < auction.BIDS_NUMBER; i++) {
                    Bid thread = new Bid(i, startPrice, auction.getBarrier());
                    auction.add(thread);
                    thread.start();
                }
        }
}
```

Результаты работы аукциона:

Client 0 specifies a price.
Client 2 specifies a price.
Client 1 specifies a price.
Client 3 specifies a price.
Client 4 specifies a price.
Bid 4 : 87
Bid 0 : 81
Bid 1 : 93
Bid 2 : 81
Bid 3 : 96
Bid #3, price:96 win!
Continue to work...
Continue to work...
Continue to work...
Continue to work...
Continue to work...

«Щеколда»

Еще один вид барьера представляет класс **CountDownLatch**. Экземпляр класса инициализируется начальным значением числа ожидающих снятия «щеколды» потоков. В отличие от **CyclicBarrier**, метод **await()** просто останавливает поток без всяких изменений значения счетчика. Значение счетчика снижается вызовом метода **countDown()**, т. е. «щеколда» сдвигается на единицу. Когда счетчик обнулится, барьеры, поставленные методом **await()**, снимаются для всех ожидающих разрешения потоков. Крайне желательно, чтобы метод **await()** был вызван раньше, чем метод **countDown()**. Последнему безразлично, вызывался метод **await()** или нет, счетчик все равно будет уменьшен на единицу. Если счетчик равен нулю, то «лишние» вызовы метода **countDown()** будут проигнорированы.

Демонстрацией возможностей класса **CountDownLatch** может служить задача выполнения студентами набора заданий (тестов). *Студенту* предлагается для выполнения набор заданий. Он выполняет их и переходит в режим ожидания оценок по всем заданиям, чтобы вычислить среднее значение оценки. *Преподаватель* (**Tutor**) проверяет задание и после каждого проверенного задания сдвигает «щеколду» на единицу. Когда все задания студента проверены, счетчик становится равным нулю и барьер снимается, производятся необходимые вычисления в классе **Student**.

// # 26 # поток-студент, выполняющий задания и ожидающий их проверки # Student.java

```
package by.bsu.learning;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CountDownLatch;
import java.util.Random;
public class Student extends Thread {
          private Integer idStudent;
          private List<Task> taskList;
          private CountDownLatch countDown;
          public Student(Integer idStudent, int numberTasks) {
                    this.idStudent = idStudent;
                    this.countDown = new CountDownLatch(numberTasks);
                    this.taskList = new ArrayList<Task>(numberTasks);
          }
          public Integer getIdStudent() {
                    return idStudent;
          }
          public void setIdStudent(Integer idStudent) {
                    this.idStudent = idStudent;
          }
          public CountDownLatch getCountDownLatch() {
                    return countDown;
          }
```

```
        public List<Task> getTaskList() {
                return taskList;
        }
        public void addTask(Task task) {
                        taskList.add(task);
        }
        public void run() {
                int i = 0;
                for (Task inWork : taskList) {
                        // на выполнение задания требуется некоторое время
                        try {
                                Thread.sleep(new Random().nextInt(100));
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                        // отправка ответа
                        inWork.setAnswer("Answer #" + ++i);
                        System.out.println("Answer #" + i + " from " + idStudent);
                }
                try {
                        countDown.await(); // ожидание проверки заданий
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                // подсчет средней оценки за все задачи
                float averageMark = 0;
                for (Task inWork : taskList) {
                        // выполнение задания
                        averageMark += inWork.getMark(); // отправка ответа
                }
                averageMark /= taskList.size();
                System.out.println("Student " + idStudent + ": Average mark = "
                                + averageMark);
        }
}
```

// # 27 # поток-тьютор, проверяющий задания # Tutor.java

```
package by.bsu.learning;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
public class Tutor extends Thread {
        private Integer idTutor;
        private List<Student> list;
        public Tutor() {
                this.list = new ArrayList<>();
        }
        public Tutor(List<Student> list) {
                this.list = list;
        }
```

```
        public Integer getIdTutor() {
                return idTutor;
        }
        public void setIdTutor(Integer id) {
                this.idTutor = id;
        }
        public void run() {
                for (Student st : list) {
                        // проверить, выданы ли студенту задания
                        List<Task> tasks = st.getTaskList();
                        for (Task current : tasks) {
                                // проверить наличие ответа!
                                int mark = 3 + new Random().nextInt(7);
                                current.setMark(mark);
                                System.out.println(mark + " for student N "
                                                + st.getIdStudent());
                                st.getCountDownLatch().countDown();
                        }
                        System.out.println("All estimates made for " + st.getIdStudent());
                }
        }
}
```

// # 28 # класс-носитель информации # Task.java

```
package by.bsu.learning;
public class Task {
        private String content;
        private String answer;
        private int mark;
        public Task(String content) {
                this.content = content;
        }
        public String getContent() {
                return content;
        }
        public void setContent(String content) {
                this.content = content;
        }
        public String getAnswer() {
                return answer;
        }
        public void setAnswer(String answer) {
                this.answer = answer;
        }
        public int getMark() {
                return mark;
        }
        public void setMark(int mark) {
                this.mark = mark;
        }
}
```

/ # 29 # запуск формирования группы студентов и выполнения проверки их заданий # RunLearning.java */*

```
package by.bsu.learning;
import java.util.ArrayList;
public class RunLearning {
          public static void main(String[] args) {
                    final int NUMBER_TASKS_1 = 5;
                    Student student1 = new Student(322801, NUMBER_TASKS_1);
                    for (int i = 0; i < NUMBER_TASKS_1; i++) {
                              Task t = new Task("Task #" + i);
                              student1.addTask(t);
                    }
                    final int NUMBER_TASKS_2 = 4;
                    Student student2 = new Student(322924, NUMBER_TASKS_2);
                    for (int i = 0; i < NUMBER_TASKS_2; i++) {
                              Task t = new Task("Task ##" + i);
                              student2.addTask(t);
                    }
                    ArrayList<Student> lst = new ArrayList<Student>();
                    lst.add(student1);
                    lst.add(student2);
                    Tutor tutor = new Tutor(lst);
                    student1.start();
                    student2.start();
                    try { // поток проверки стартует с задержкой
                              Thread.sleep(5000);
                    } catch (InterruptedException e) {
                              e.printStackTrace();
                    }
                    tutor.start();
          }
}
```

В результате будет выведено:

Answer #1 from 322801
Answer #2 from 322801
Answer #1 from 322924
Answer #2 from 322924
Answer #3 from 322801
Answer #4 from 322801
Answer #5 from 322801
Answer #3 from 322924
Answer #4 from 322924
3 for student N%322801
6 for student N%322801
9 for student N%322801
3 for student N%322801

6 for student N%322801
All mark for 322801 accepted
9 for student N%322924
8 for student N%322924
3 for student N%322924
9 for student N%322924
All mark for 322924 accepted
Student 322924: Average mark = 7.25
Student 322801: Average mark = 5.4

Следует отметить, что в этом и предыдущих заданиях не полностью выполнены все условия по обеспечению корректной работы приложений во всех возможных ситуациях, возникающих в условиях многопоточности. Следует рассматривать данный пример в качестве каркаса для построения работоспособного приложения.

Обмен блокировками

Существует возможность безопасного обмена объектами, в том числе и синхронизированными. Функционал обмена представляет класс **Exchanger** с его единственным методом **T exchange(T ob)**. Возвращаемый параметр метода — объект, который будет принят из другого потока, передаваемый параметр **ob** метода — собственный объект потока, который будет отдан другому потоку.

Поток **Producer** представляет информацию о количестве произведенного товара, поток **Consumer** — о количестве проданного. В результате обмена производитель снизит план производства, если количество проданного товара меньше произведенного. Потребитель, к тому же, снижает цену на товар, так как поступления товара больше, чем продано за время, предшествующее обмену.

/ # 30 # содержит Exchanger и представляет основу для производителя и потребителя # Subject.java */*

```
package by.bsu.exchanger;
import java.util.concurrent.Exchanger;
public class Subject {
          protected static Exchanger<Item> exchanger = new Exchanger<>();
          private String name;
          protected Item item;
          public Subject(String name, Item item) {
                    this.name = name;
                    this.item = item;
          }
          public String getName() {
                    return name;
          }
```

```
        public Item getItem() {
                return item;
        }
        public void setName(String name) {
                this.name = name;
        }
        public void setItem(Item item) {
                this.item = item;
        }
}
```

/ # 31 # поток-производитель товара, обменивающегося информацией о планах производства с потребителем # Producer.java */*

```
package by.bsu.exchanger;
public class Producer extends Subject implements Runnable {
    public Producer(String name, Item item) {
            super(name, item);
    }
    public void run() {
            try {
            synchronized(item) { // блок синхронизации не нужен, но показателен
                    int proposedNumber = this.getItem().getNumber();
                    //  обмен синхронизированными экземплярами
                    item  =  exchanger.exchange(item);
                    if (proposedNumber <= item.getNumber()) {
                            System.out.println("Producer " + getName()
                            + " повышает план производства товара");
                    } else  {
                            System.out.println("Producer " + getName()
                            + " снижает план производства товара");
                    }
            }
            } catch (InterruptedException e) {
                    e.printStackTrace();
            }
    }
}
```

/ # 32 # поток-потребитель товара, обменивающийся уровнем продаж с производителем # Consumer.java */*

```
package by.bsu.exchanger;
public class Consumer extends Subject implements Runnable {
    public Consumer(String name, Item item) {
            super(name, item);
    }
    public void run() {
            try {
            synchronized(item) { // блок синхронизации не нужен, но показателен
```

```
                    int requiredNumber = item.getNumber();
                    item = exchanger.exchange(item); // обмен
                    if (requiredNumber >= item.getNumber()) {
                              System.out.println("Consumer " + getName()
                              + " повышает стоимость товара");
                    } else  {
                              System.out.println("Consumer " + getName()
                              + " снижает стоимость товара");
                    }
               }
          } catch  (InterruptedException e) {
                    e.printStackTrace();
          }
     }
}
```

/ # 33 # класс-описание товара # Item.java */*

```
package by.bsu.exchanger;
public class Item {
          private Integer id;
          private Integer number;
          public Item(Integer id, Integer number) {
                    super();
                    this.id = id;
                    this.number = number;
          }
          public Integer getId() {
                    return id;
          }
          public Integer getNumber() {
                    return number;
          }
}
```

/ # 34 # процесс обмена # RunExchange.java */*

```
package by.bsu.exchanger;
public class RunExchange {
          public static void main(String[ ] args) {
                    Item ss1 = new Item(34, 2200);
                    Item ss2 = new Item(34, 2100);
                    new Thread(new Producer("HP ", ss1)).start();
                    new Thread(new Consumer("RETAIL Trade", ss2)).start();
          }
}
```

В результате будет получено:

Consumer RETAIL Trade снижает стоимость товара
Producer HP снижает план производства товара

Альтернатива synchronized

Синхронизация ресурса ключевым словом **synchronized** накладывает достаточно жесткие правила на освобождение этого ресурса. Интерфейс **Lock** представляет собой некоторое обобщение синхронизации. Появляется возможность провести опрос о блокировании, установить время ожидания блокировки и условия ее прерывания. Интерфейс также оптимизирует работу JVM с процессами конкурирования за освобождаемые ресурсы.

Класс **ReentrantLock** представляет два основных метода:

void lock() — получает блокировку экземпляра. Если экземпляр блокирован другим потоком, то поток отключается и бездействует до освобождения экземпляра;

void unlock() — освобождает блокировку экземпляра. Если текущий поток не является обладателем блокировки, генерируется исключение **IllegalMonitorStateException**.

Шаблонное применение этих методов после объявления экземпляра **locking** класса **ReentrantLock**:

```
locking.lock();
try {
   // some code here
} finally {
   locking.unlock();
}
```

Данная конструкция копирует функциональность блока **synchronized**. Гибкость классу предоставляют методы:

boolean tryLock() — получает блокировку экземпляра. Если блокировка выполнена другим потоком, то метод сразу возвращает **false**;

boolean tryLock(long timeout, TimeUnit unit) — получает блокировку экземпляра. Если экземпляр блокирован другим потоком, то метод приостанавливает поток на время **timeout**, и если блокировка становится возможна в течение этого интервала, то поток ее получает, если же блокировка недоступна, то метод возвращает **false**.

Метод **lock()** и оба метода **tryLock()** при повторном успешном получении блокировки в одном и том же потоке увеличивают счетчик блокировок на единицу, что требует соответствующего числа снятий блокировки.

Класс **Condition** предназначен для управления блокировкой. Ссылку на экземпляр можно получить только из объекта типа **Lock** методом **newCondition()**. Расширение возможностей происходит за счет методов **await()** и **signal()**, функциональность которых подобна действию методов **wait()** и **notify()** класса **Object**.

Пусть необходим нереляционный способ сохранения информации в коллекции, когда неделимым квантом информации считается пара или более следующих друг за другом элементов. То есть добавление и удаление элементов может

осуществляться только парами и другой поток не может добавить/удалить свои элементы, пока заблокировавший коллекцию поток полностью не выполнит свои действия.

/ # 35 # ресурсы добавляются и удаляются только парами # DoubleResource.java */*

```
package by.bsu.lock;
import java.util.Deque;
import java.util.LinkedList;
import java.util.Random;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class DoubleResource {
          private Deque<String> list = new LinkedList<String>();
          private Lock lock = new ReentrantLock();
          private Condition isFree = lock.newCondition();
          public void adding(String str, int i) {
                    try {
                              lock.lock();
                              list.add(i + "<" + str);
                              TimeUnit.MILLISECONDS.sleep(new Random().nextInt(50));
                              list.add(str + ">" + i);
                              isFree.signal();
                    } catch (InterruptedException e) {
                              e.printStackTrace();
                    } finally {
                              lock.unlock();
                    }
          }
          public String deleting() {
                    lock.lock();
                    String s = list.poll();
                    s += list.poll();
                    isFree.signal();
                    lock.unlock();
                    return s;
          }
          public String toString() {
                    return list.toString();
          }
}
```

/ # 36 # поток доступа к ресурсу # ResThread.java */*

```
package by.bsu.lock;
import java.util.Random;
public class ResThread extends Thread {
          private DoubleResource resource;
          public ResThread(String name, DoubleResource rs) {
```

```
                super(name);
                resource = rs;
        }
        public void run() {
                for (int i = 1; i < 4; i++) {
                        if (new Random().nextInt(2) > 0) {
                                resource.adding(getName(), i);
                        } else {
                                resource.deleting();
                        }
                }
        }
}
```

/ # 37 # запуск процессов доступа к ресурсу # SynchroMain.java */*

```
package by.bsu.lock;
import java.util.concurrent.TimeUnit;
public class SynchroMain {
        public static void main(String[ ] args) throws InterruptedException {
                for (int i = 0; i < 5; i++) {
                        DoubleResource resource = new DoubleResource();
                        new ResThread("a", resource).start();
                        new ResThread("Z", resource).start();
                        new ResThread("#", resource).start();
                        TimeUnit.MILLISECONDS.sleep(1_000);
                        System.out.println(resource);
                }
        }
}
```

В результате может быть получено:

[1<a, a>1, 2<a, a>2, 1<Z, Z>1, 1<#, #>1, 3<Z, Z>3, 3<#, #>3]
[]
[2<a, a>2, 3<a, a>3, 1<Z, Z>1, 3<Z, Z>3, 2<#, #>2]
[3<a, a>3, 3<Z, Z>3, 3<#, #>3]
[1<a, a>1, 2<a, a>2, 3<a, a>3, 3<#, #>3]

где результат с пустыми скобками свидетельствует, что попыток изъятия пар было больше, чем попыток добавления. Нерезультативные попытки не фиксировались.

ExecutorService и Callable

В альтернативной системе управления потоками разработан механизм исполнителей, функции которого заключаются в запуске отдельных потоков и их групп, а также в управлении ими: принудительной остановке, контроле числа работающих потоков и планирования их запуска.

Класс **ExecutorService** методом **execute(Runnable thread)** запускает традиционные потоки, метод же **submit(Callable<T> task)** запускает потоки с возвращаемым значением. Метод **shutdown()** останавливает все запущенные им ранее потоки и прекращает действие самого исполнителя. Статические методы **newSingleThreadExecutor()** и **newFixedThreadPool(int numThreads)** класса **Executors** определяют правила, по которым работает **ExecutorService**, а именно первый позволяет исполнителю запускать только один поток, второй — не более чем указано в параметре **numThreads**, ставя другие потоки в очередь ожидания окончания уже запущенных потоков.

/ # 38 # поток с возвращением результата # CalcCallable.java */*

```
package by.bsu.future;
import java.util.Random;
import java.util.concurrent.Callable;
public class CalcCallable implements Callable<Number> {
          @Override
          public Number call() throws Exception {
                    Number res = new Random().nextGaussian(); // имитация вычислений
                    return res;
          }
}
```

/ # 39 # запуск потока и извлечение результата его выполнения # CalcRunner.java */*

```
package by.bsu.future;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
public class CalcRunner {
          public static void main(String[] args) {
                    ExecutorService es = Executors.newSingleThreadExecutor();
                    Future<Number> future = es.submit(new CalcCallable());
                    es.shutdown();
                    try {
                              System.out.println(future.get());
                    } catch (InterruptedException | ExecutionException e) {
                              e.printStackTrace();
                    }
          }
}
```

Интерфейс **Callable** представляет поток, возвращающий значение вызывающему потоку. Определяет один метод **V call() throws Exception**, в код реализации которого и следует поместить решаемую задачу. Результат выполнения метода **call()** может быть получен через экземпляр класса **Future**, методами **V get()** или **V get(long timeout, TimeUnit unit)**, как и продемонстрировано в предыдущем

примере. Перед извлечением результатов работы потока **Callable** можно проверить, завершилась ли задача успешно, методами **isDone()** и **isCancelled()** соответственно.

/ # 40 # список обрабатываемых объектов # ProductList.java */*

```
package by.bsu.future;
import java.util.ArrayDeque;
public class ProductList {
          private static ArrayDeque<String> arr = new ArrayDeque<String>() {
                    {
                              this.add("Product 1");
                              this.add("Product 2");
                              this.add("Product 3");
                              this.add("Product 4");
                              this.add("Product 5");
                    }
          };
          public static String getProduct() {
                    return arr.poll();
          }
}
```

/ # 41 # поток обработки экземпляра продукта # BaseCallable.java */*

```
package by.bsu.future;
import java.util.concurrent.Callable;
import java.util.concurrent.TimeUnit;
public class BaseCallable implements Callable<String> {
          @Override
          public String call() throws Exception {
                    String product = ProductList.getProduct();
                    String result = null;
                    if (product != null) {
                              result = product + " done";
                    } else {
                              result = "productList is empty";
                    }
                    TimeUnit.MILLISECONDS.sleep(100);
                    System.out.println(result);
                    return result;
          }
}
```

/ # 42 # запуск пула потоков и извлечение результатов их работы # RunExecutor.java */*

```
package by.bsu.future;
import java.util.ArrayList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
```

```
import java.util.concurrent.Future;
public class RunExecutor {
          public static void main(String[] args) throws Exception {
                    ArrayList<Future<String>> list = new ArrayList<Future<String>>();
                    ExecutorService es = Executors.newFixedThreadPool(3);
                    for (int i = 0; i < 7; i++) {
                              list.add(es.submit(new BaseCallable()));
                    }
                    es.shutdown();
                    for (Future<String> future : list) {
                              System.out.println(future.get() + " result fixed");
                    }
          }
}
```

В результате выполнения будет, возможно, выведено:

Product 3 done
Product 1 done
Product 2 done
Product 1 done result fixed
Product 3 done result fixed
Product 2 done result fixed
productList is empty
Product 5 done
Product 4 done
Product 4 done result fixed
Product 5 done result fixed
productList is empty result fixed
productList is empty
productList is empty result fixed

Phaser

Более сложное поведение этого синхронизатора **Phaser** напоминает поведение **CyclicBarrier**, однако число участников синхронизации может меняться. Участвующие стороны сначала должны зарегистрироваться phaser-объектом. Регистрация осуществляется с помощью методов **register()**, **bulkRegister(int parties)** или подходящего конструктора. Выход из синхронизации phaser-объектом производит метод **arriveAndDeregister()**, причем выход из числа синхронизируемых сторон может быть и в случае, когда поток завершил выполнение, и в случае, когда поток все еще выполняется. Основным назначением класса **Phaser** является синхронизация потоков, выполнение которых требуется разбить на отдельные этапы (фазы), а эти фазы, в свою очередь, необходимо синхронизовать. **Phaser** может как задержать поток, пока другие потоки не достигнут конца

текущей фазы методом **arriveAndAwaitAdvance()**, так и пропустить поток, отметив лишь окончание какой-либо фазы методом **arrive()**.

/ # 43 # поток # Truck.java */*

```
package by.bsu.phaser;
import java.util.ArrayDeque;
import java.util.Queue;
import java.util.Random;
import java.util.concurrent.Phaser;
public class Truck implements Runnable {
          private Phaser phaser;
          private String number;
          private int capacity;
          private Storage storafeFrom;
          private Storage storageTo;
          private Queue<Item> bodyStorage;
          public Truck(Phaser phaser, String name, int capacity, Storage stFrom,
                              Storage stTo) {
                    this.phaser = phaser;
                    this.number = name;
                    this.capacity = capacity;
                    this.bodyStorage = new ArrayDeque<Item>(capacity);
                    this.storafeFrom = stFrom;
                    this.storageTo = stTo;
                    this.phaser.register();
          }
          public void run() {
                    loadTruck();
                    phaser.arriveAndAwaitAdvance();

                    goTruck();
                    phaser.arriveAndAwaitAdvance();

                    unloadTruck();
                    phaser.arriveAndDeregister();
          }
          private void loadTruck() {
                    for (int i = 0; i < capacity; i++) {
                              Item g = storafeFrom.getGood();
                              if (g == null) { // если в хранилище больше нет товара,
                                                  // загрузка грузовика прерывается
                                        return;
                              }
                              bodyStorage.add(g);
                              System.out.println("Грузовик " + number + " загрузил товар №"
                                                  + g.getRegistrationNumber());
                              try {
                                        Thread.sleep(50);
                              } catch (InterruptedException e) {
                                        e.printStackTrace();
```

```
                    }
                }
        }
        private void unloadTruck() {
                int size = bodyStorage.size();
                for (int i = 0; i < size; i++) {
                        Item g = bodyStorage.poll();
                        storageTo.setGood(g);
                        System.out.println("Грузовик " + number + " разгрузил товар №"
                                        + g.getRegistrationNumber());
                        try {
                                Thread.sleep(50);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
        private void goTruck() {
                try {
                        Thread.sleep(new Random(100).nextInt(500));
                        System.out.println("Грузовик " + number + " начал поездку.");
                        Thread.sleep(new Random(100).nextInt(1000));
                        System.out.println("Грузовик " + number + " завершил поездку.");
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}
```

/ # 44 # перевозимый товар # Item.java */*

```
package by.bsu.phaser;
public class Item {
        private int registrationNumber;
        public Item(int number) {
                this.registrationNumber = number;
        }
        public int getRegistrationNumber() {
                return registrationNumber;
        }
}
```

/ # 45 # склад-коллекция # Storage.java */*

```
package by.bsu.phaser;
import java.util.Iterator;
import java.util.List;
import java.util.Queue;
import java.util.concurrent.LinkedBlockingQueue;
public class Storage implements Iterable<Item> {
```

```
        public static final int DEFAULT_STORAGE_CAPACITY = 20;
        private Queue<Item> goods = null;
        private Storage() {
                goods =
                new LinkedBlockingQueue<Item>(DEFAULT_STORAGE_CAPACITY);
        }
        private Storage(int capacity) {
                goods = new LinkedBlockingQueue<Item>(capacity);
        }
        public static Storage createStorage(int capacity) {
                Storage storage = new Storage(capacity);
                return storage;
        }
        public static Storage createStorage(int capacity, List<Item> goods) {
                Storage storage = new Storage(capacity);
                storage.goods.addAll(goods);
                return storage;
        }
        public Item getGood() {
                return goods.poll();
        }
        public boolean setGood(Item good) {
                return goods.add(good);
        }
        @Override
        public Iterator<Item> iterator() {
                return goods.iterator();
        }
}
```

/ # 46 # запуск процесса # PhaserDemo.java */*

```
package by.bsu.phaser;
import java.util.Arrays;
import java.util.Iterator;
import java.util.List;
import java.util.concurrent.Phaser;
public class PhaserDemo {
        public static void main(String[] args) {
                // создание коллекцию товаров
                Item[] goods = new Item[20];
                for (int i = 0; i < goods.length; i++) {
                        goods[i] = new Item(i + 1);
                }
                List<Item> listGood = Arrays.asList(goods);
                // создание склада, из которого забирают товары
                Storage storageA = Storage.createStorage(listGood.size(), listGood);
                // создание склада, куда перевозят товары
                Storage storageB = Storage.createStorage(listGood.size());
                // создание фазера для синхронизации движения колонны грузовиков
```

```
                Phaser phaser = new Phaser();
                phaser.register();
                int currentPhase;
                // создание колонны грузовиков
                Thread tr1 = new Thread(new Truck(phaser, "tr1", 5, storageA, storageB));
                Thread tr2 = new Thread(new Truck(phaser, "tr2", 6, storageA, storageB));
                Thread tr3 = new Thread(new Truck(phaser, "tr3", 7, storageA, storageB));
                printGoodsToConsole("Товары на складе A", storageA);
                printGoodsToConsole("Товары на складе B", storageB);
                // запуск колонны грузовиков на загрузку на одном складе + поездку +
                // разгрузку на другом складе
                tr1.start();
                tr2.start();
                tr3.start();
                // синхронизация загрузки
                currentPhase = phaser.getPhase();
                phaser.arriveAndAwaitAdvance();
                System.out.println("Загрузка колонны завершена. Фаза " + currentPhase
                                + " завершена.");
                // синхронизация поездки
                currentPhase = phaser.getPhase();
                phaser.arriveAndAwaitAdvance();
                System.out.println("Поездка колонны завершена. Фаза " + currentPhase
                                + " завершена.");
                // синхронизация разгрузки
                currentPhase = phaser.getPhase();
                phaser.arriveAndAwaitAdvance();
                System.out.println("Разгрузка колонны завершена. Фаза " + currentPhase
                                + " завершена.");
                phaser.arriveAndDeregister();
                if (phaser.isTerminated()) {
                        System.out.println("Фазы синхронизированы и завершены.");
                }
                printGoodsToConsole("Товары на складе A", storageA);
                printGoodsToConsole("Товары на складе B", storageB);
        }
        public static void printGoodsToConsole(String title, Storage storage) {
                System.out.println(title);
                Iterator<Item> goodIterator = storage.iterator();
                while (goodIterator.hasNext()) {
                        System.out.print(goodIterator.next().getRegistrationNumber() + " ");
                }
                System.out.println();
        }
}
```

Товары на складе A
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Товары на складе B
Грузовик tr1 загрузил товар №1

Грузовик tr2 загрузил товар №2
Грузовик tr3 загрузил товар №3
Грузовик tr2 загрузил товар №4
Грузовик tr1 загрузил товар №5
Грузовик tr3 загрузил товар №6
Грузовик tr2 загрузил товар №7
Грузовик tr1 загрузил товар №8
Грузовик tr3 загрузил товар №9
Грузовик tr2 загрузил товар №10
Грузовик tr3 загрузил товар №11
Грузовик tr1 загрузил товар №12
Грузовик tr2 загрузил товар №13
Грузовик tr3 загрузил товар №14
Грузовик tr1 загрузил товар №15
Грузовик tr2 загрузил товар №16
Грузовик tr3 загрузил товар №17
Грузовик tr3 загрузил товар №18
Загрузка колонны завершена. Фаза 0 завершена.
Грузовик tr2 начал поездку.
Грузовик tr1 начал поездку.
Грузовик tr3 начал поездку.
Грузовик tr1 завершил поездку.
Грузовик tr3 завершил поездку.
Грузовик tr2 завершил поездку.
Грузовик tr2 разгрузил товар №2
Грузовик tr1 разгрузил товар №1
Грузовик tr3 разгрузил товар №3
Поездка колонны завершена. Фаза 1 завершена.
Грузовик tr2 разгрузил товар №4
Грузовик tr1 разгрузил товар №5
Грузовик tr3 разгрузил товар №6
Грузовик tr2 разгрузил товар №7
Грузовик tr1 разгрузил товар №8
Грузовик tr3 разгрузил товар №9
Грузовик tr2 разгрузил товар №10
Грузовик tr1 разгрузил товар №12
Грузовик tr3 разгрузил товар №11
Грузовик tr2 разгрузил товар №13
Грузовик tr1 разгрузил товар №15
Грузовик tr3 разгрузил товар №14
Грузовик tr2 разгрузил товар №16
Грузовик tr3 разгрузил товар №17

Грузовик tr3 разгрузил товар №18
Разгрузка колонны завершена. Фаза 2 завершена.
Фазы синхронизированы и завершены.
Товары на складе A
19 20
Товары на складе B
2 3 1 4 5 6 7 8 9 10 12 11 13 15 14 16 17 18

Задания к главе 11

Вариант A

Разработать многопоточное приложение.

Использовать возможности, предоставляемые пакетом java.util.concurrent.

Не использовать слово synchronized.

Все сущности, желающие получить доступ к ресурсу, должны быть потоками.

Использовать возможности ООП.

Не использовать графический интерфейс. Приложение должно быть консольным.

1. **Порт**. Корабли заходят в порт для разгрузки/загрузки контейнеров. Число контейнеров, находящихся в текущий момент в порту и на корабле, должно быть неотрицательным и превышающим заданную грузоподъемность судна и вместимость порта. В порту работает несколько причалов. У одного причала может стоять один корабль. Корабль может загружаться у причала, разгружаться или выполнять оба действия.
2. **Маленькая библиотека**. Доступны для чтения несколько книг. Одинаковых книг в библиотеке нет. Некоторые выдаются на руки, некоторые только в читальный зал. Читатель может брать на руки и в читальный зал несколько книг.
3. **Автостоянка**. Доступно несколько машиномест. На одном месте может находиться только один автомобиль. Если все места заняты, то автомобиль не станет ждать больше определенного времени и уедет на другую стоянку.
4. **CallCenter**. В организации работает несколько операторов. Оператор может обслуживать только одного клиента, остальные должны ждать своей очереди. Клиент может положить трубку и перезвонить еще раз через некоторое время.
5. **Автобусные остановки**. На маршруте несколько остановок. На одной остановке может останавливаться несколько автобусов одновременно, но не более заданного числа.
6. **Свободная касса**. В ресторане быстрого обслуживания есть несколько касс. Посетители стоят в очереди в конкретную кассу, но могут перейти в другую очередь при уменьшении или исчезновении там очереди.

7. **Тоннель**. В горах существует два железнодорожных тоннеля, по которым поезда могут двигаться в обоих направлениях. По обоим концам тоннеля собралось много поездов. Обеспечить безопасное прохождение тоннелей в обоих направлениях. Поезд можно перенаправить из одного тоннеля в другой при превышении заданного времени ожидания на проезд.
8. **Банк.** Имеется банк с кассирами, клиентами и их счетами. Клиент может снимать/пополнять/переводить/оплачивать/обменивать денежные средства. Кассир последовательно обслуживает клиентов. Поток-наблюдатель следит, чтобы в кассах всегда были наличные, при скоплении денег более определенной суммы, часть их переводится в хранилище, при истощении запасов наличных происходит пополнение из хранилища.
9. **Аукцион.** На торги выставляется несколько лотов. Участники аукциона делают заявки. Заявку можно корректировать в сторону увеличения несколько раз за торги одного лота. Аукцион определяет победителя и переходит к следующему лоту. Участник, не заплативший за лот в заданный промежуток времени, отстраняется на несколько лотов от торгов.
10. **Биржа.** На торгах брокеры предлагают акции нескольких фирм. На бирже совершаются действия по купле-продаже акций. В зависимости от количества проданных-купленных акций их цена изменяется. Брокеры предлагают к продаже некоторую часть акций. От активности и роста-падения котировок акций изменяется индекс биржи. Биржа может приостановить торги при резком падении индекса.
11. **Аэропорт**. Посадка/высадка пассажиров может осуществляться через конечное число терминалов и наземным способом через конечное число трапов. Самолеты бывают разной вместимости и дальности полета. Организовать функционирование аэропорта, если пунктов назначения 4–6, и зон дальности 2–3.

Вариант В

Для заданий варианта В гл. 4 организовать синхронизированный доступ к ресурсам (файлам). Для каждого процесса создать отдельный поток выполнения.