

Прикладное программирование

Лекция №9



МНОГОПОТОЧНЫЕ ПРИЛОЖЕНИЯ в Java

- Что такое процессы?
- Что такое потоки выполнения?
- Как управлять потоками в Java?
- Как работать с потоками-демонами и объектами-призраками?

Процессы

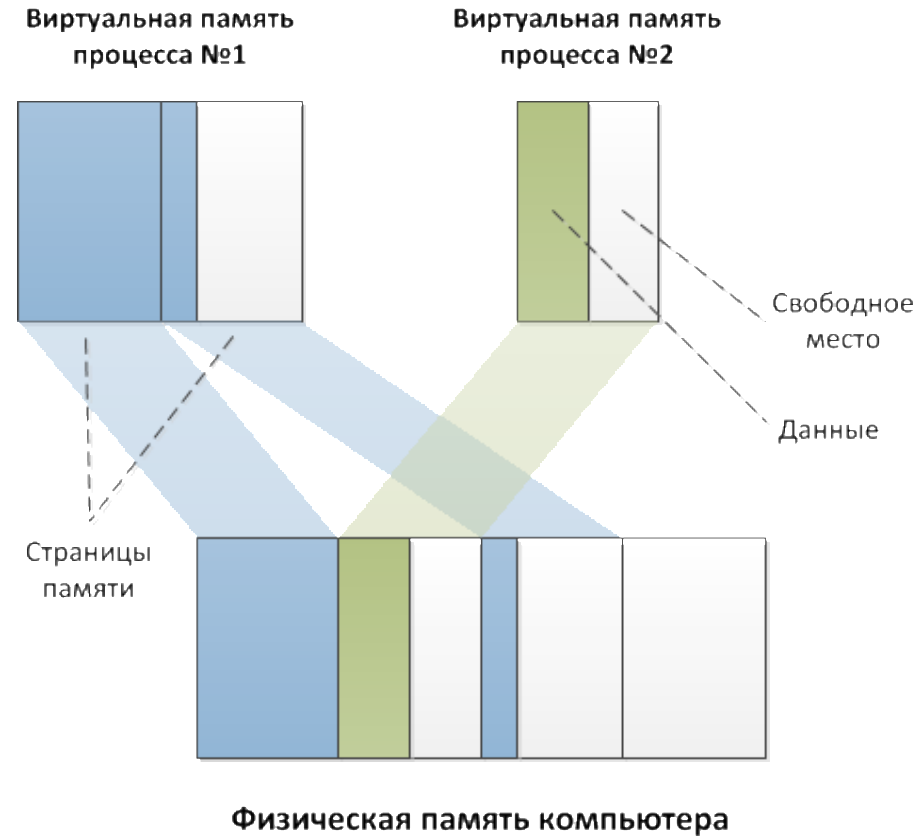
Процесс операционной системы — это совокупность кода и данных, разделяющих общее *виртуальное адресное пространство*.

Процессы ОС всегда изолированы друг от друга, поэтому прямой доступ к памяти чужого процесса невозможен (взаимодействие между процессами осуществляется с помощью специальных средств).

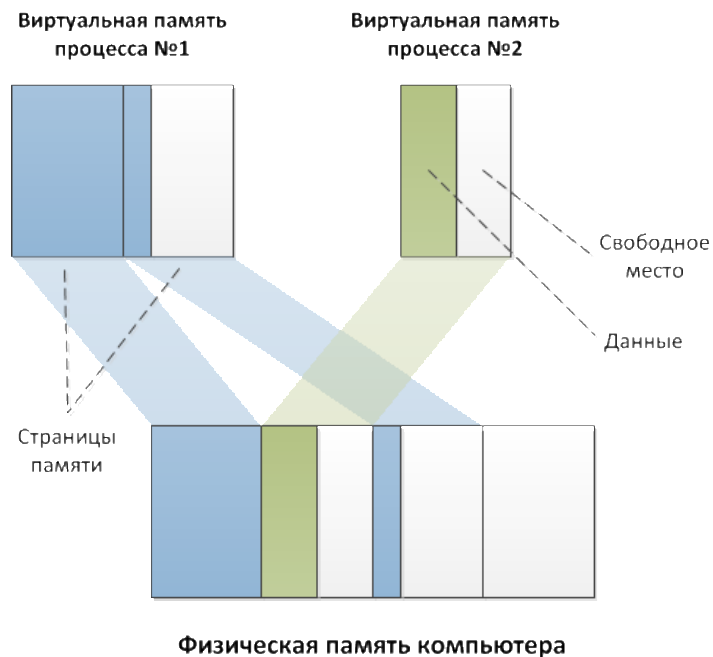
Для каждого процесса ОС создает так называемое **«виртуальное адресное пространство»**, к которому процесс имеет прямой доступ.

Это пространство **принадлежит процессу**, содержит **только его данные** и находится **в полном его распоряжении**.

Процессы



Процессы



ОС оперирует так называемыми **страницами памяти**, которые представляют собой просто **область определенного фиксированного размера**.

Если процессу становится *недостаточно памяти*, система выделяет ему **дополнительные страницы из физической памяти**.

Страницы виртуальной памяти могут проецироваться на физическую память *в произвольном порядке*.

При запуске программы ОС создает процесс, загружая в его адресное пространство *код и данные программы*, а затем запускает **главный поток созданного процесса**.

Взаимодействие процессов

Процессы изолированы друг от друга, и если вдруг у них возникает прямая необходимость для обмена данными, то для этого используется **IPC (Inter Process Communication) механизмы**, предоставляемые операционной системой.



Поток выполнения

Поток выполнения представляет собой **последовательность инструкций**, которая выполняется в контексте определенного процесса.

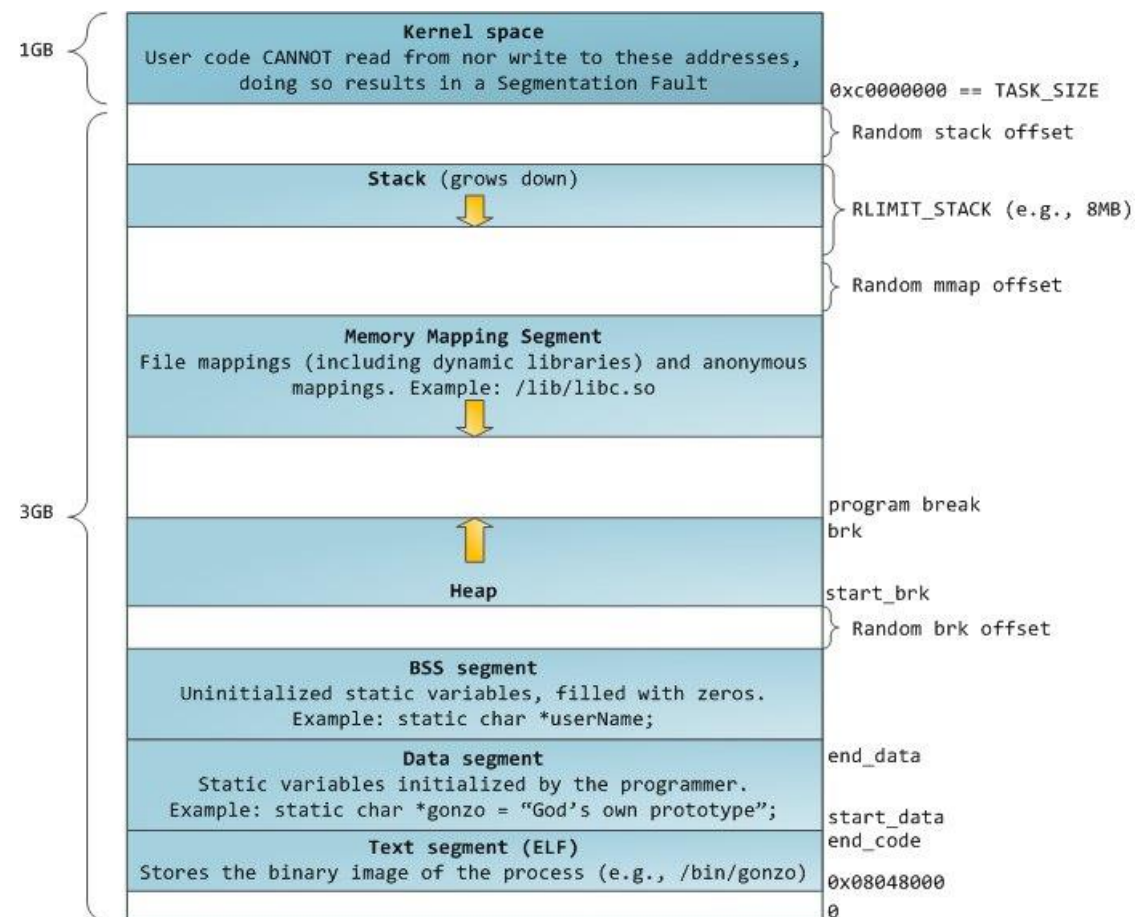
Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, *параллельно с другими потоками* этого процесса.

Другими словами, происходит *одновременное выполнение нескольких задач в независимых потоках*.

Процессы всегда выполняются в независимых адресных пространствах. В отличие от процессов, **потоки находятся в общем адресном пространстве**.

Т.е. потоки работают с одной и той же областью памяти, которая выделена конкретному процессу, что порождает необходимость их **синхронизации** при манипуляциях с данными.

Поток выполнения



А что было до многоядерных процессоров?

Даже когда существовали только одноядерные процессоры, возможно было запускать несколько параллельных потоков выполнения.

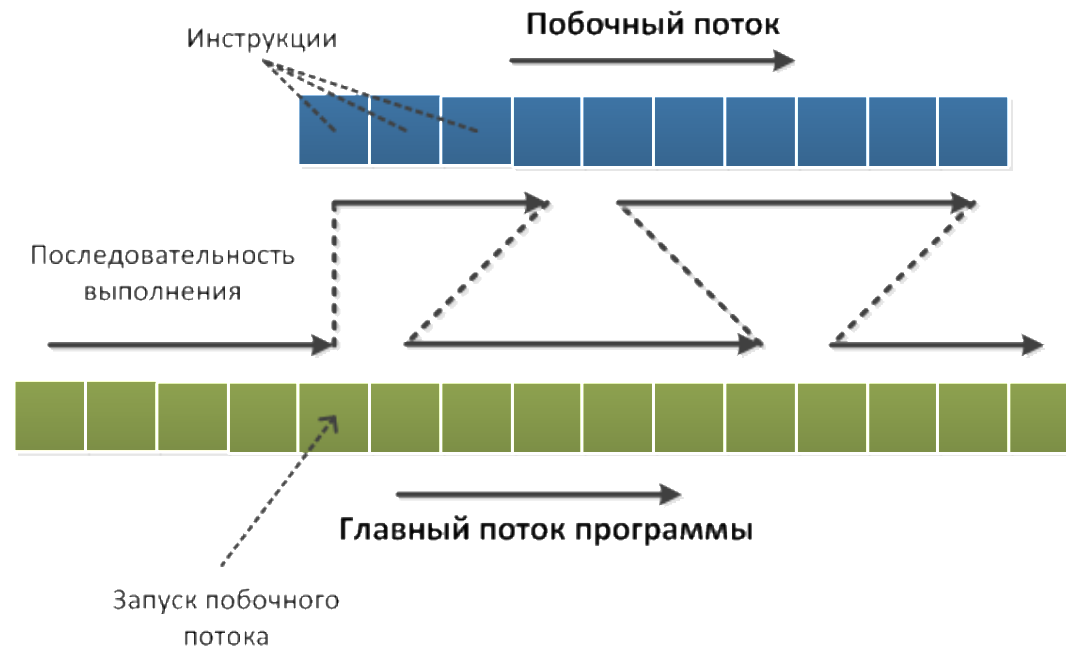
В таком случае система периодически переключается между потоками, поочередно выполняя то один, то другой поток. Такая схема называется **псевдо-параллелизмом**.

Как это работает?

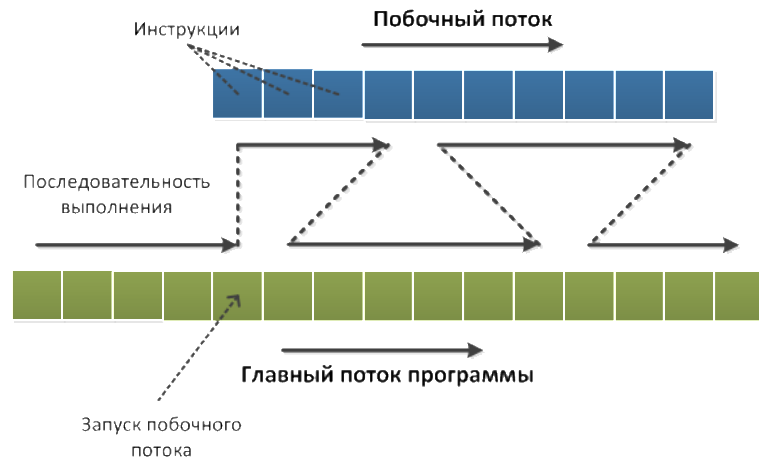
Система **запоминает состояние каждого потока**, перед тем как переключиться на другой поток, и **восстанавливает его по возвращению** к выполнению потока. В контекст (состояние) потока входят такие параметры, как **стек, набор значений регистров процессора, адрес исполняемой команды** и прочее.

Потоки

При псевдопараллельном выполнении потоков процессор постоянно переключается между выполнением нескольких потоков, выполняя по очереди часть каждого из них.



Потоки



Цветные квадраты – это инструкции процессора.

- **зеленые** – инструкции главного потока,
- **синие** – инструкции побочного потока.

После запуска побочного потока его инструкции начинают выполняться вперемешку с инструкциями главного потока.

Кол-во выполняемых инструкций за каждый подход **не определено.**

Выполнение инструкций параллельных потоков в случайном порядке в некоторых случаях может привести к конфликтам доступа к данным (*речь о синхронизации*).

Многопоточность

Многопоточность – это особое свойство платформы (OS, VM) или конкретного приложения, заключающееся в том, что процесс, запущенный в ОС, может состоять из нескольких потоков выполнения, работающих параллельно.

Подобное распределение позволяет достигать максимально эффективного использования вычислительной мощности устройства при выполнении определенных задач.

Многопоточность



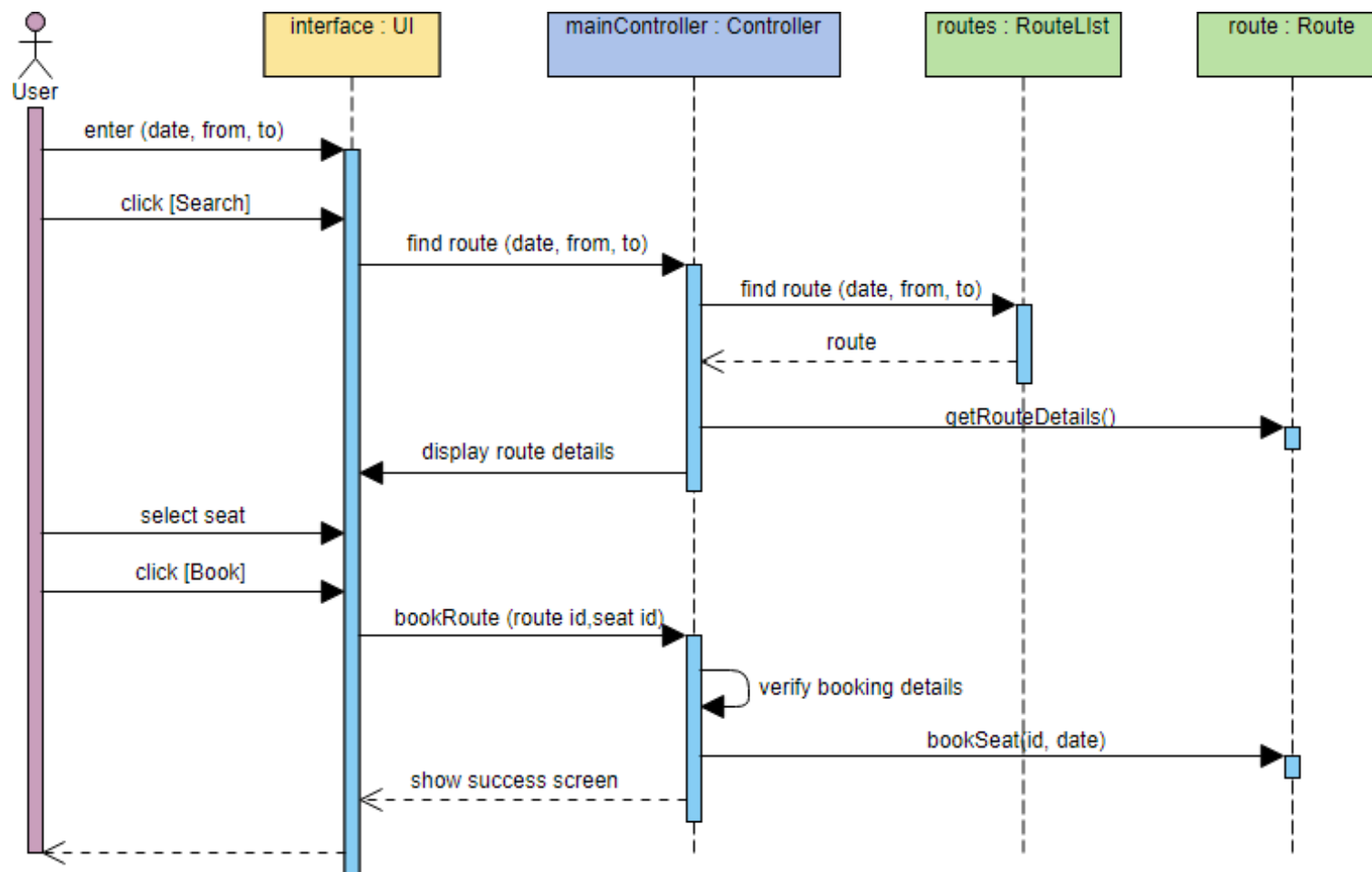
Диаграмма последовательности (sequence diagram)

Для моделирования взаимодействия объектов в языке UML используется **диаграмма последовательности**.

Взаимодействия объектов можно рассматривать во времени, и тогда для представления временных особенностей передачи и приема сообщений между объектами используется диаграмма последовательности.

Взаимодействующие объекты обмениваются между собой некоторой информацией. При этом информация принимает форму законченных сообщений. Другими словами, хотя сообщение и имеет информационное содержание, оно приобретает дополнительное свойство оказывать направленное влияние на своего получателя.

Диаграмма последовательности (sequence diagram)



Общее представление о потоках выполнения

В приложении всегда имеется **главный (основной) поток выполнения**. Если он **закрывается** – закрываются **все** пользовательские потоки приложения.

Java Specific

В Java кроме пользовательских потоков возможно создание **потоков-демонов (daemons)**, которые могут продолжать работу и после окончания работы главного потока выполнения.

Как правило, потоки-демоны являются вспомогательными потоками и часто используются системой.

Особенности потоков в Java

Любая программа в Java неявно использует **потоки выполнения**.

В главном потоке JVM запускает **метод main()** приложения, а также все методы, которые вызываются из него, при этом главному потоку автоматически присваивается имя **main**.

Кроме главного потока в фоновом режиме (т.е. с малым приоритетом) запускается **дочерний поток**, который занимается **сборкой мусора**.

Пример необходимости потоков

последовательный алгоритм ходьбы

Представим программу, описывающую **процесс неторопливой ходьбы человека**.

Возможны две идеологии работы программы – **последовательная** и **параллельная**.

При последовательном подходе относительную частоту выполнения таких действий, как **вдох-выдох** и **шаги левой-правой ногами** необходимо задавать самостоятельно.

Пример необходимости потоков

последовательный алгоритм ходьбы

Предположим, что в случае неторопливой ходьбы на каждые 2 шага приходится **1 вдох-выдох**.

```
1
2 while (!finish) {
3     inhale();
4     makeStepLeftFoot();
5     makeStepRightFoot();
6     exhale();
7 }
8
```

Пример необходимости потоков

последовательный алгоритм бега

Представим, что необходимо описать **алгоритм быстрого бега**, при котором **на каждый вдох-выдох** приходится **4 шага**:

```
1
2 while (!finish) {
3     inhale();
4     makeStepLeftFoot();
5     makeStepRightFoot();
6     exhale();
7     makeStepLeftFoot();
8     makeStepRightFoot();
9 }
10
```

Пример необходимости потоков

последовательный алгоритм быстрого шага

Представим, что необходимо описать **алгоритм быстрого шага**, при котором на каждые **2 вдоха-выдоха** приходится **5 шагов**:

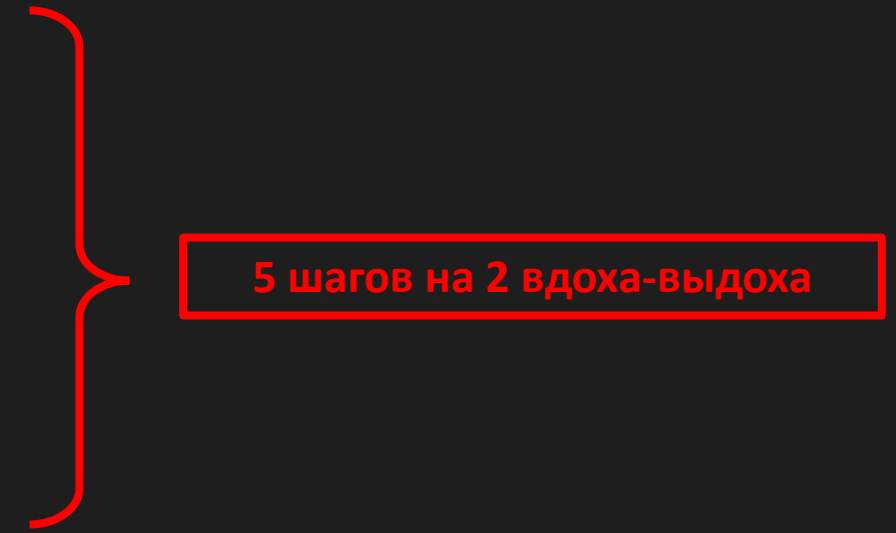
```
2 while (!finish) {  
3     inhale();  
4     makeStepLeftFoot();  
5     makeStepRightFoot();  
6     exhale();  
7     makeStepLeftFoot();  
8     makeStepRightFoot();  
9     inhale();  
10    makeStepLeftFoot();  
11    makeStepRightFoot();  
12    exhale();  
13    makeStepLeftFoot();  
14 }  
15  
16
```

Пример необходимости потоков

последовательный алгоритм быстрого шага

Проблемы возникают при описании **сложных действий**, частота происхождения составляющих компонентов которых **не является кратной друг другу**.

```
2 while (!finish) {  
3     inhale();  
4     makeStepLeftFoot();  
5     makeStepRightFoot();  
6     exhale();  
7     makeStepLeftFoot();  
8     makeStepRightFoot();  
9     inhale();  
10    makeStepLeftFoot();  
11    makeStepRightFoot();  
12    exhale();  
13    makeStepLeftFoot();  
14 }  
15  
16
```



5 шагов на 2 вдоха-выдоха

Пример необходимости потоков

последовательный алгоритм быстрого шага

Код программы перестает быть структурным.

```
2 while (!finish) {  
3     inhale();  
4     makeStepLeftFoot();  
5     makeStepRightFoot();  
6     exhale();  
7     makeStepLeftFoot();  
8     makeStepRightFoot();  
9     inhale();  
10    makeStepLeftFoot();  
11    makeStepRightFoot();  
12    exhale();  
13    makeStepLeftFoot();  
14 }  
15
```

Нарушается принцип инкапсуляции:

- «Независимые вещи должны быть независимы».
- Независимые по логике решаемой проблемы алгоритмы оказываются перемешаны друг с другом.

Такой код **ненадежный** и **плохо модифицируемый**, при этом он достаточно просто отлаживается, т.к. **последовательность выполнения операторов программы однозначно определена.**

Пример необходимости потоков

параллельный алгоритм шага

При описании процесса ходьбы с точки зрения параллельного подхода, мы имеем **два действия**, происходящих **одновременно**:

- Перемещение ног
- Дыхание (вдох-выдох)

1		10	
2	<code>while (!finish) {</code>	11	<code>while (!finish) {</code>
3	<code>inhale();</code>	12	<code>makeStepLeftFoot();</code>
4	<code>sleep(XXX);</code>	13	<code>sleep(YYY);</code>
5	<code>exhale();</code>	14	<code>makeStepRightFoot();</code>
6	<code>sleep(XXX);</code>	15	<code>sleep(YYY);</code>
7	<code>}</code>	16	<code>}</code>
8	<div style="border: 2px solid red; padding: 2px; display: inline-block;">Поток №1</div>	17	<div style="border: 2px solid red; padding: 2px; display: inline-block;">Поток №2</div>
9		18	

Пример необходимости потоков

параллельный алгоритм шага

В таком случае **относительная частота** происхождения событий задается **приоритетом потоков***, а также константами **XXX** и **YYY**, на которые засыпают потоки.

1		10	
2	<code>while (!finish) {</code>	11	<code>while (!finish) {</code>
3	<code>inhale();</code>	12	<code>makeStepLeftFoot();</code>
4	<code>sleep(XXX);</code>	13	<code>sleep(YYY);</code>
5	<code>exhale();</code>	14	<code>makeStepRightFoot();</code>
6	<code>sleep(XXX);</code>	15	<code>sleep(YYY);</code>
7	<code>}</code>	16	<code>}</code>
8	<div style="border: 2px solid red; padding: 2px; display: inline-block;">Поток №1</div>	17	<div style="border: 2px solid red; padding: 2px; display: inline-block;">Поток №2</div>
9		18	

Создание потоков в Java

Имеется два способа создать класс, экземплярами которого будут потоки выполнения:

- Унаследовать класс от `java.lang.Thread`;
- Реализовать интерфейс `java.lang.Runnable`.

Интерфейс `Runnable` имеет декларацию единственного метода

```
public void run() { ... },
```

который обеспечивает последовательность действий при работе потока.

Создание потоков в Java

Класс **Thread** уже реализует интерфейс **Runnable**, но с пустой реализацией метода **run()**, поэтому при создании экземпляра **Thread** автоматически создается поток (но не запускается!!!).

В потомках **метод run()** необходимо переопределять, поместив в него реализацию алгоритмов, которые должны выполняться в данном потоке.

После выполнения метода **run()** поток **прекращает существование**.

Способ №1

создание потока на основе класса Thread

Объект-поток создается с помощью конструктора. Имеется несколько перегруженных вариантов конструкторов, самый простой из них – с пустым списком параметров.

Например, в классе **Thread** их заголовки выглядят так:

- **public Thread()** – конструктор по умолчанию. Поток получает имя **system**.
- **public Thread (String name)** – поток получает имя **name**.

Способ №1

создание потока на основе класса Thread

Создание и запуск потока осуществляется следующим образом:

```
2 public class T1 extends Thread {  
3     public void run() {  
4         // Реализация метода run  
5     }  
6  
7     // Прочие методы класса  
8     ...  
9  
10    Thread thread1 = new T1();  
11    thread1.start();  
12 }  
13
```

Способ №2

создание потока на основе интерфейса Runnable

Такой способ позволяет создавать **потоки на основе классов**, уже включенных в иерархию наследования (т.е. тех, которые не могут выбрать **Thread** в качестве своего предка).

Но при таком подходе **затрудняется доступ ко внутренним полям потока**, т.к. формально их внутри нашего класса нет.

Способ №2

создание потока на основе интерфейса Runnable (1/2)

Эту особенность приходится исправлять добавлением во внутреннее поле данных экземпляра класса ссылки на объект потока, в котором он выполняется.

```
1
2 public class R1 implements Runnable {
3     public void run() {
4         // Реализация метода run
5     }
6
7     // Прочие методы класса
8     ...
9
10    Runnable myRunnable = new Runnable();
11    Thread thread2 = new Thread(myRunnable);
12    thread2.start();
13 }
14
15
```

Способ №2

создание потока на основе интерфейса Runnable (2/2)

С помощью ссылки на объект потока можно обращаться к его полям данных и методам, например, считывая текущий приоритет потока:

containerThread.getPriority();

```
1
2 public class R1 implements Runnable {
3     Thread containerThread;
4     public void run() {
5         // Реализация метода run
6     }
7
8     public void setThread(Thread containerThread) {
9         this.containerThread = containerThread;
10    }
11
12    // Прочие методы класса
13    ...
14
15    Runnable myRunnable = new Runnable();
16    Thread thread2 = new Thread(myRunnable);
17    myRunnable.setThread(thread2);
18    thread2.start();
19 }
20
21
```


Жизненный цикл потока

При выполнении программы **объект класса Thread** может быть в одном из **четырёх основных состояний**:

- «Новый»,
- «Работоспособный»,
- «Неработоспособный»,
- «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.

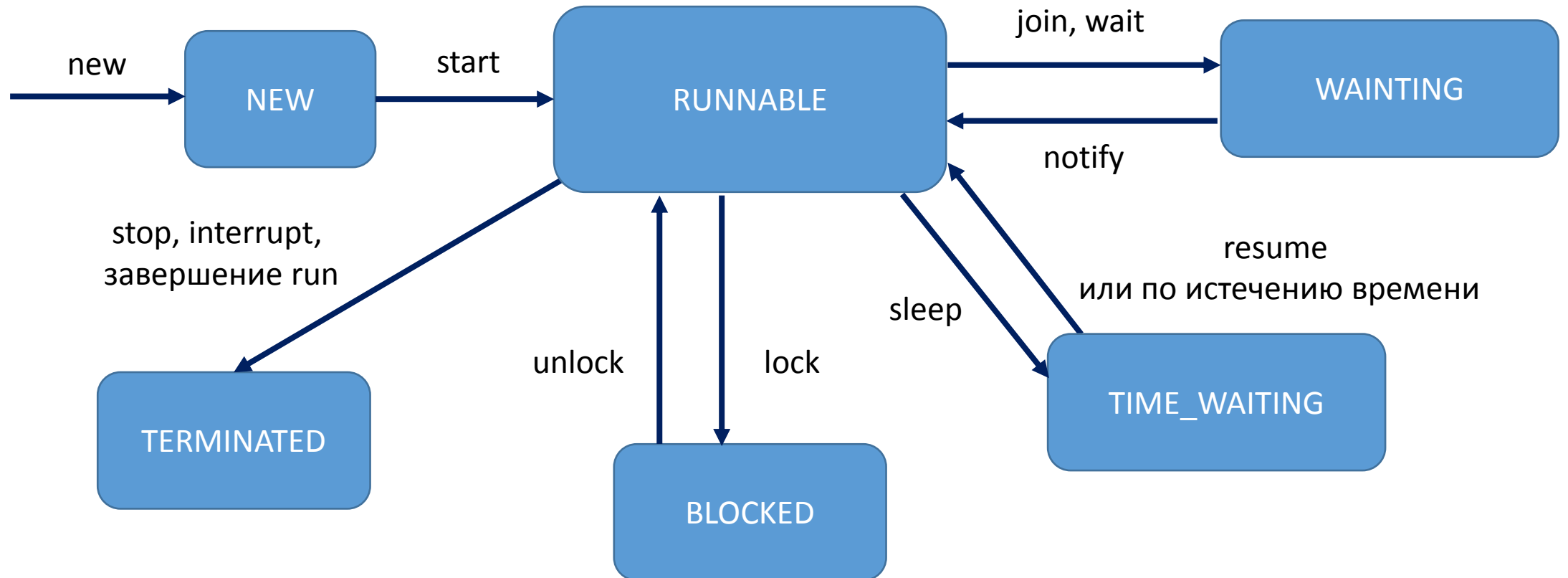
Жизненный цикл потока

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;
- **TIMED_WAITING** — поток некоторое время ждет окончания другого потока;
- **TERMINATED** — поток завершен.

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

Жизненный цикл потока



Управление приоритетами потоков

Каждый поток в системе имеет свой приоритет.

Приоритет – это некоторое число в объекте потока.

Чем выше значение, тем больший приоритет присвоен потоку. Система в первую очередь выполняет потоки с высшим приоритетом, а потоки с меньшим приоритетом получают процессорное время только тогда, когда более привилегированные потоки простаивают.

- **void setPriority(int priority)** – потоку назначается приоритет:

- от 1 (**MIN_PRIORITY**)
- 5 (**NORM_PRIORITY**)
- до 10 (**MAX_PRIORITY**)

** Прочие значения задаются целочисленным значением (от 1 до 10).*

- **int getPriority()** – получить текущее значение приоритета конкретного потока.

Группы потоков

Часто потоки объединяются в **группы потоков**.

После создания потока **нельзя** изменить его принадлежность к группе. 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 со значением приоритета, **более низким**, чем приоритет группы после включения в группу, **не изменит** значения своего приоритета.*

Потоки-демоны (Java Specific)

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.

Потоки-демоны (на практике)

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 все еще работает.

Если уменьшать время задержки потока-демона, то он может успеть завершить свое выполнение до окончания работы основного потока.

Поток-демон – это поток, который не предотвращает выход JVM, когда программа завершается, а поток все еще работает. Примером для потока демона является **сборщик мусора**.

Потоки-демоны (на практике)

- Когда создается новый поток, он наследует **статус демона** своего родителя.
- Когда все потоки-не-демоны завершают выполнение, **JVM** **останавливается**, блоки больше не выполняются, оставшиеся **потоки-демоны также останавливаются** и, следовательно, **JVM** просто завершает свою работу.

!!! Потоки-демоны должны использоваться **экономно!**

Их **опасно** использовать для задач, которые могут выполнять какие-либо операции ввода-вывода.

Потоки и исключения

В процессе выполнения программы потоки представляются 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.

```
2  throw new RuntimeException()  
3  // try-catch(InterruptedException ex)-finally  
4  
5      или  
6  
7  public static void main(String[ ] args) throws InterruptedException {  
8      // ...  
9  }  
10
```

Потоки и исключения

Основной поток **избавлен** от необходимости *обрабатывать исключения в порожденных потоках.*

В данной ситуации верно и обратное утверждение:

если основной поток прекратит свое выполнение из-за необработанного исключения, то это не скажется на работоспособности порожденного им потока.

Коллизия доступа

Часто возникают ситуации, когда несколько потоков пытаются использовать общий ресурс либо получают доступ к одним и тем же данным. При этом они начинают друг другу мешать.

Есть риск, что таким образом потоки могут **повредить** этот ресурс. Например, когда несколько потоков записывают информацию в **файл/объект/поток**.

Следовательно, для них требуется обеспечить **разграничение доступа**.

Синхронизация потоков

Пока один из потоков имеет доступ к данным, никакой другой поток не должен иметь возможности и полномочий их читать или изменять. Он должен дожидаться завершения первого потока.

Речь идет о **«синхронизации потоков»**.

Еще на лекции по работе с потоками ввода-вывода мы выяснили, что нельзя допускать ситуации, когда выполняются одновременные операции чтения и записи. Это может привести к повреждению целостности данных.

Для контролирования процесса чтения/записи может использоваться разделение ресурса с применением ключевого слова **synchronized**.

Синхронизация потоков

синхронизация на ресурсах

Синхронизация на ресурсах – это тип синхронизации, который обеспечивает **блокировку данных** на то время, которое необходимо потоку для выполнения тех или иных действий.

Имеется два способа синхронизации на ресурсах:

- **Синхронизация на объектах**
- **Синхронизация на методах.**

Синхронизация потоков

синхронизация на ресурсах

Блокировка такого рода осуществляется на основе **концепции мониторов*** и заключается в следующем:

- Под **монитором** понимается некая **управляющая конструкция**, обеспечивающая монопольный доступ к объекту.
- Если во время выполнения синхронизованного метода объекта другой поток попытается обратиться к методам или данным этого объекта, он будет заблокирован до тех пор, пока не закончится выполнение синхронизованного метода.
- При **запуске синхронизованного метода** говорят, что объект **входит в монитор**.
- При **завершении** – что объект **выходит из монитора**.

При этом поток, внутри которого вызван синхронизованный метод, считается владельцем данного монитора.

Синхронизация на ресурсах

синхронизация на объектах

Синхронизация на объекте **object** осуществляется следующим образом:

```
2 synchronized(object) {  
3     method1(object);  
4     object.method2();  
5  
6     // Прочие синхронизированные действия  
7 }  
8
```

Синхронизация на ресурсах

синхронизация на объектах

В данном случае в качестве **синхронизованного оператора** выступает участок кода в фигурных скобках.

Во время выполнения этого участка доступ к **объекту object** блокируется для всех других потоков, т.е. пока будет выполняться вызов оператора, выполнение вызова любого синхронизованного метода или синхронизованного оператора для этого объекта будет приостановлено до окончания работы оператора.

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

Синхронизация на ресурсах

синхронизация на методах

Второй способ синхронизации на ресурсах используется **при разработке классов**, рассчитанных на взаимодействия в **многопоточной среде**.

При этом методы, **критичные к атомарности операций** с данными (обычно – требующие согласованное изменение нескольких полей данных), объявляются как **синхронизованные** с помощью модификатора **synchronized**:

```
9 public class MyClass {  
10     public synchronized void someMethod() {  
11         // Реализация метода  
12     }  
13 }  
14
```

Синхронизация на ресурсах

синхронизация на методах

Вызов **object.someMethod()** приведет к вхождению объекта в монитор, и, пока он будет выполняться, доступ из других потоков к объекту будет блокирован – выполнение любого синхронизованного метода или оператора для этого объекта будет приостановлено до окончания работы метода.

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

Синхронизация потоков

синхронизация на событиях

Кроме синхронизации на данных имеется **синхронизация на событиях**, когда параллельно выполняющиеся потоки приостанавливаются вплоть до наступления **некоторого события**, о котором **им сигнализирует другой поток**.

Основными операциями при таком типе синхронизации являются **ждать** и **оповестить**.

Синхронизация потоков

синхронизация на событиях

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

- **wait()** – поток, внутри которого какой-либо объект вызвал данный метод, приостанавливает работу своего **метода run()** вплоть до поступления уведомления объекту, вызвавшему остановку, через методы **notify()** или **notifyAll()**. При неправильной попытке «разбудить» поток соответствующий код компилируется, но при запуске инициирует исключение **IllegalMonitorStateException**.
- **join()** – позволяет одному потоку (thread1) ждать завершения выполнения другого (thread2). Как только поток thread2 завершится, метод join() вернет управление потоку thread1, и он сможет продолжить выполнение.

Синхронизация потоков

синхронизация на событиях

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

- **notify()** – оповещение, приводящее к возобновлению работы потока, ожидающего выхода данного объекта из монитора. Если таких потоков несколько, выбирается один из них (какой именно – зависит от реализации JVM).
- **notifyAll()** – оповещение, приводящее к возобновлению работы всех потоков, ожидающих выхода данного объекта из монитора.

Реализация синхронизации на событиях (способ 1)

Метод **wait()** для любого объекта следует использовать следующим образом – организовать цикл **while**, в котором следует выполнять оператор **wait()**:

```
1
2 synchronized(object) {
3     while(!<some_condition>) {
4         object.wait();
5
6         // Выполнение операторов после разблокирования;
7     }
8 }
9
```

Реализация синхронизации на событиях (способ 1)

Не следует беспокоиться, что постоянно работающий цикл **while** будет занимать ресурсы процессора.

После вызова **object.wait()** поток, в котором находится указанный код, засыпает и перестает работать.

На время сна метод **wait()** снимает блокировку с объекта **object**, задаваемую оператором **synchronized(object)**, что позволяет другим потокам обращаться к объекту с вызовом **object.notify()** или **object.notifyAll()**.

Реализация синхронизации на событиях (способ 2)

Альтернативным способом реализации синхронизации на событиях является замена используемого в предыдущем фрагменте кода синхронизации на объекте на синхронизацию на его методах.

В этом случае мы можем сконструировать специальный объект, контролирующий и ограничивающий одновременный доступ потоков:

Реализация синхронизации на событиях (способ 2)

```
1
2 public class Lock {
3     private boolean locked;
4
5     public synchronized void checkIfLocked() throws InterruptedException {
6         while(locked) wait();
7     }
8
9     public synchronized void lock() {
10         locked = true;
11     }
12
13     public synchronized void unlock() {
14         locked = false;
15         notifyAll();
16     }
17 }
18
19
```

Важные константы и методы класса Thread

Имя константы или метода	Описание
MIN_PRIORITY	Минимально возможный приоритет потоков. Зависит от операционной системы и версии JVM .
NORM_PRIORITY	Нормальный приоритет потоков. Главный поток создается с нормальным приоритетом, а затем приоритет может быть изменен.
MAX_PRIORITY	Максимально возможный приоритет потока.
static Thread currentThread()	Возвращает ссылку на текущий поток .
boolean holdsLock(Object obj)	Возвращает true в случае, когда текущий поток блокирует объект obj .

Важные методы экземпляров класса Thread

Имя константы или метода	Описание
void run()	Метод, который обеспечивает последовательность действий во время жизни потока.
void start()	Вызывает выполнение текущего потока , в том числе запуск его метода run() в нужном контексте. Может быть вызван всего один раз .
void yield()	Вызывает временную приостановку потока , с передачей управления другим потокам.
long getId()	Возвращает уникальный (только во время жизни) идентификатор потока
String getName()	Возвращает имя потока, которое ему было задано при создании или методом setName()
void setName(String name)	Устанавливает новое имя потока .

Важные методы экземпляров класса Thread

Имя константы или метода	Описание
int <code>getPriority()</code>	Возвращает приоритет потока.
void <code>setPriority(int newPriority)</code>	Устанавливает приоритет потока.
String <code>toString()</code>	Возвращает строковое представление объекта потока, в том числе – его имя, группу, приоритет.
void <code>sleep(long millis)</code>	Вызывает приостановку (“засыпание”) потока на millis миллисекунд. При этом все блокировки потока сохраняются.
void <code>interrupt()</code>	Прерывает сон потока, вызванный вызовами wait() или sleep() , устанавливая ему статус прерванного. При этом возбуждается проверяемая исключительная ситуация InterruptedException .

Важные методы экземпляров класса Thread

Имя константы или метода	Описание
boolean <code>isInterrupted()</code>	Возвращает текущее состояние статуса прерывания потока без изменения значения статуса.
void <code>join()</code>	Переводит поток в режим умирания – ожидания завершения (смерти). Обычно используется для завершения главным потоком работы всех дочерних пользовательских потоков (“слияния” их с главным потоком).
boolean <code>isAlive()</code>	Возвращает true в случае, когда текущий поток жив (не умер). Даже если поток завершился, от него остается объект-“призрак” , отвечающий на запрос <code>isAlive()</code> значением false .

Прочие способы управления потоками

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**.

Ограниченно **потокобезопасные (thread safe) коллекции** и вспомогательные классы управления потоками тоже сосредоточены в пакете **java.util.concurrent**.

Пакет `java.util.concurrent` (1/3)

- 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;

Пакет `java.util.concurrent` (2/3)

- 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;

Пакет `java.util.concurrent` (3/3)

- **Классы-барьеры синхронизации:**
 - **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.

Пакет `java.util.concurrent.locks`

В пакете `java.util.concurrent.locks` содержатся дополнительные реализации моделей синхронизации потоков:

- 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 существовавшую 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 — несколькими.

Семафор



Семафор – переменная, которая используется для управления доступом к общему ресурсу несколькими процессами в многопоточных системах.

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.

Семафор

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 **false** 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.

Семафор

```
2 public void run() {  
3     try {  
4         semaphore.acquire();  
5         // Код использования защищаемого ресурса  
6     }  
7     catch(InterruptedException ex) {  
8         ex.printStackTrace();  
9     }  
10    finally {  
11        semaphore.release(); // Освобождение семафора  
12    }  
13 }  
14  
15
```

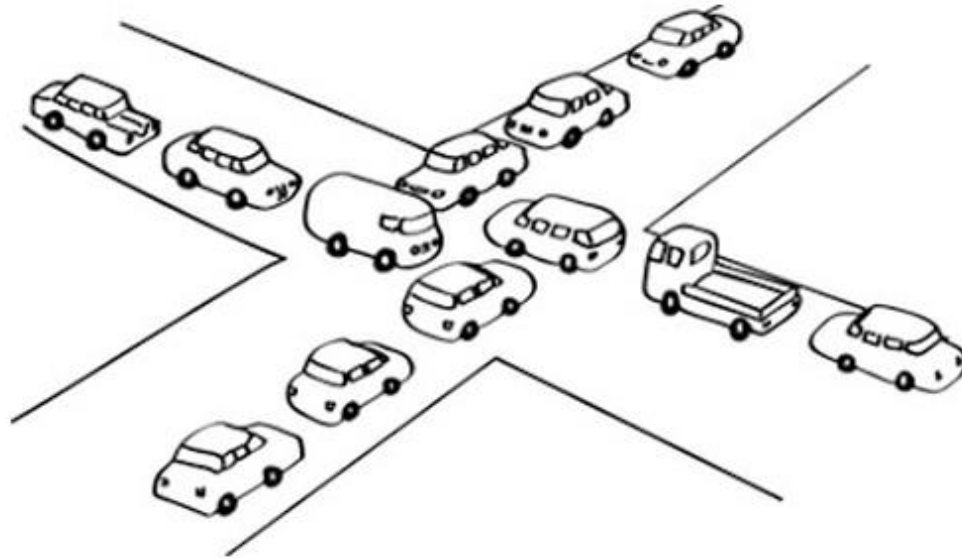
Семафор

```
2 public class HelloSynchronizedBlock extends Thread {  
3     private int num = 0;  
4     private static final Object semaphore = new Object();  
5  
6     private static void work(int number) throws InterruptedException {  
7         synchronized (semaphore) {  
8             for (int i = 0; i < 10; i++) {  
9                 Thread.sleep(100);  
10                System.out.println(i + " of " + number);  
11            }  
12        }  
13    }  
14  
15    public HelloSynchronizedBlock(int num) {  
16        this.num = num;  
17    }  
18  
19    @Override  
20    public void run() { ... }  
21  
22    public static void main(String[] args) { ... }  
23 }  
24  
25
```

Семафор

```
2 public class HelloSynchronizedBlock extends Thread {
3     private int num = 0;
4     private static final Object semaphore = new Object();
5
6     private static void work(int number) throws InterruptedException { ... }
7
8     public HelloSynchronizedBlock(int num) {
9         this.num = num;
10    }
11
12    @Override
13    public void run() {
14        try {
15            work(num);
16        }
17        catch (InterruptedException ex) {
18            ex.printStackTrace();
19        }
20    }
21
22    public static void main(String[] args) {
23        for (int i = 0; i < 10; i++) {
24            new HelloSynchronizedBlock(i).start();
25        }
26    }
27 }
28
```

DeadLock



DeadLock

Deadlock или взаимная блокировка — это ошибка, которая происходит, когда потоки имеют циклическую зависимость от пары синхронизированных объектов.

Предположим, один поток входит в монитор объекта X, а другой - в монитор объекта Y.

Если поток в объекте X попытается вызвать любой синхронизированный метод для объекта Y, он будет блокирован, как и предполагалось.

Но если поток исполнения в объекте Y, в свою очередь, попытается вызвать любой синхронизированный метод для объекта X, то этот поток будет ожидать вечно, поскольку для получения доступа к объекту X он должен снять свою блокировку с объекта Y, чтобы первый поток исполнения мог завершиться.

DeadLock

DeadLock является ошибкой, которую трудно отладить, по **двум** следующим причинам:

- Когда исполнение двух потоков **точно совпадает по времени** (возникает очень редко);
- Когда участвует **больше двух потоков исполнения и двух синхронизированных объектов**. (взаимная блокировка может произойти в результате более сложной последовательности событий)

DeadLock

```
2 public class HelloDeadLock extends Thread {
3     private Thread anotherThread;
4     private String name;
5
6     public HelloDeadLock(String name) {
7         this.name = name;
8     }
9
10    public void setAnotherThread(Thread anotherThread) {
11        this.anotherThread = anotherThread;
12    }
13
14    @Override
15    public void run() {
16        synchronized (anotherThread) {
17            System.out.println(name + " waiting.");
18            try {
19                anotherThread.wait();
20            }
21            catch (InterruptedException ex) {
22                ex.printStackTrace();
23            }
24        }
25    }
26
27    public static void main(String[] args) { ... }
28
29 }
30
```

DeadLock

```
2 public class HelloDeadLock extends Thread {
3     private Thread anotherThread;
4     private String name;
5
6     public HelloDeadLock(String name) {
7         this.name = name;
8     }
9
10    public void setAnotherThread(Thread anotherThread) {
11        this.anotherThread = anotherThread;
12    }
13
14    @Override
15    public void run() { ... }
16
17    public static void main(String[] args) {
18        HelloDeadLock first = new HelloDeadLock("First");
19        HelloDeadLock second = new HelloDeadLock("Second");
20
21        first.setAnotherThread(second);
22        second.setAnotherThread(first);
23        first.setDaemon(true);
24        first.start();
25        second.start();
26    }
27 }
28
29
```

DeadLock

```
2 public class HelloDeadLock extends Thread {
3     private Thread anotherThread;
4     private String name;
5
6     public HelloDeadLock(String name) {
7         this.name = name;
8     }
9
10    public void setAnotherThread(Thread anotherThread) {
11        this.anotherThread = anotherThread;
12    }
13
14    @Override
15    public void run() { ... }
16
17    public static void main(String[] args) {
18        HelloDeadLock first = new HelloDeadLock("First");
19        HelloDeadLock second = new HelloDeadLock("Second");
20
21        first.setAnotherThread(second);
22        second.setAnotherThread(first);
23        first.setDaemon(true);
24        first.start();
25        second.start();
26    }
27 }
28 First waiting.
29 Second waiting.
```

Как избежать DeadLock?

1. **Первое правило «взаимных блокировок»:** избегать взаимных блокировок.
2. **Второе правило «взаимных блокировок»:** используйте интерфейс **Lock** из пакета **java.util.concurrent.locks**, который позволяет занять монитор, связанный с экземпляром данного класса методом **tryLock()** (возвращает **true**, если удалось занять монитор).

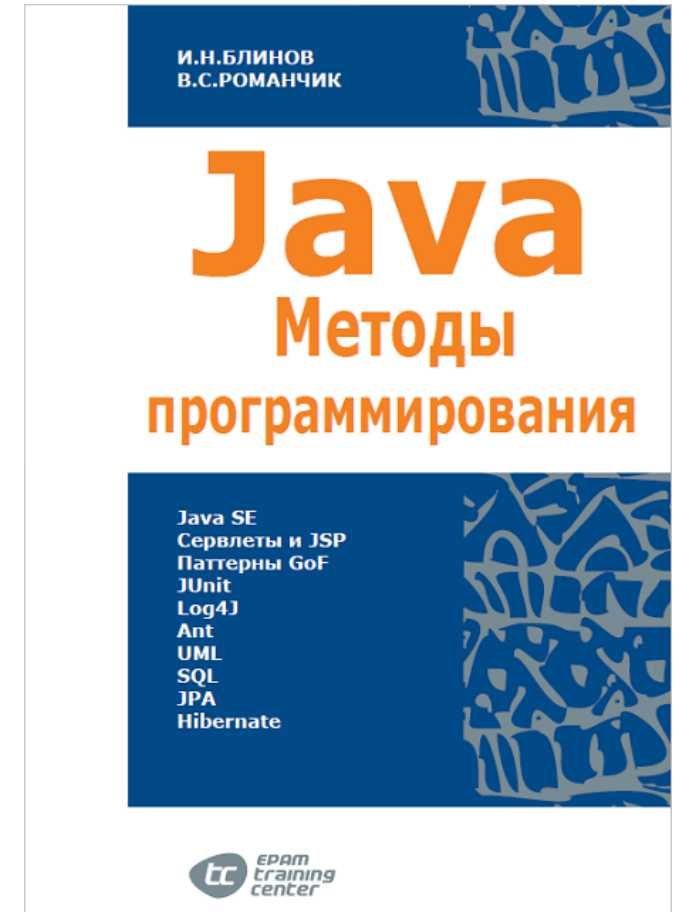
!!! Необходимо использовать **блок try-finally** и метод **unlock()**, т.к. классы из пакета **java.util.concurrent.locks** автоматически не освобождают монитор, и если в процессе выполнения вашей задачи возникло какое-то исключение, то монитор зависнет в заблокированном состоянии.

Подробнее рекомендую изучить самостоятельно:

- Глава 11. Поток выполнения, страница 290.
- Либо в любой другой книге.

p.s. особо рекомендуется к изучению
для успешного получения зачета =^.^=

Не будьте (V)_Oo_(V) (V)_oO_(V) (V)_00_(V)



Увидимся на следующей лекции!