

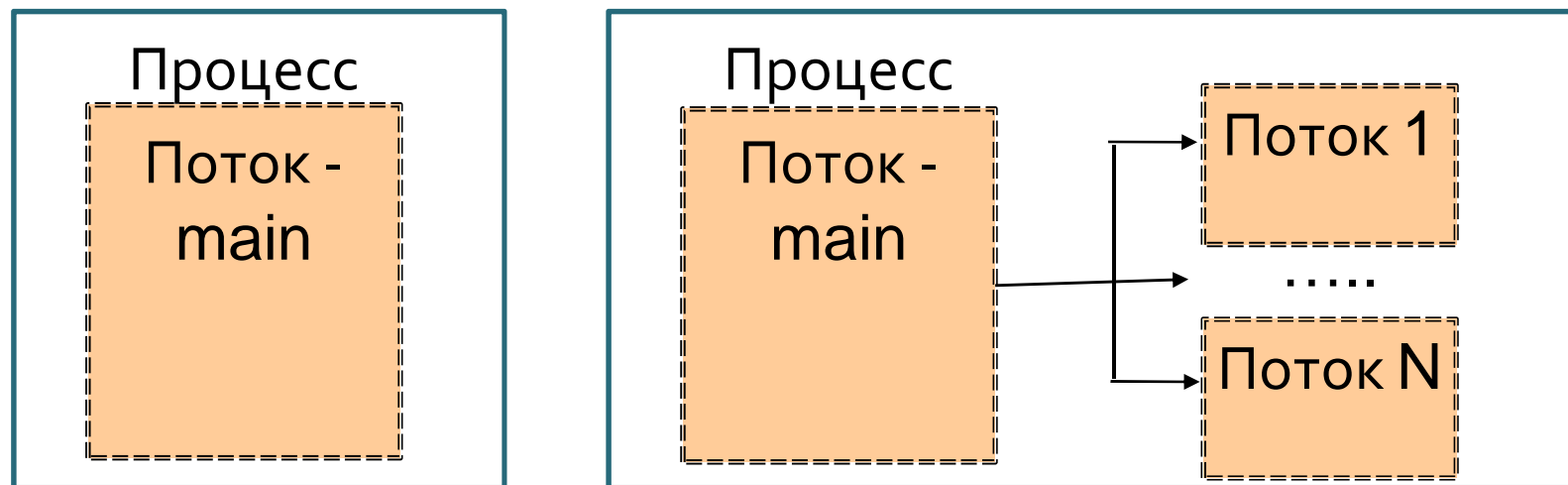


Потоки исполнения



Потоки исполнения

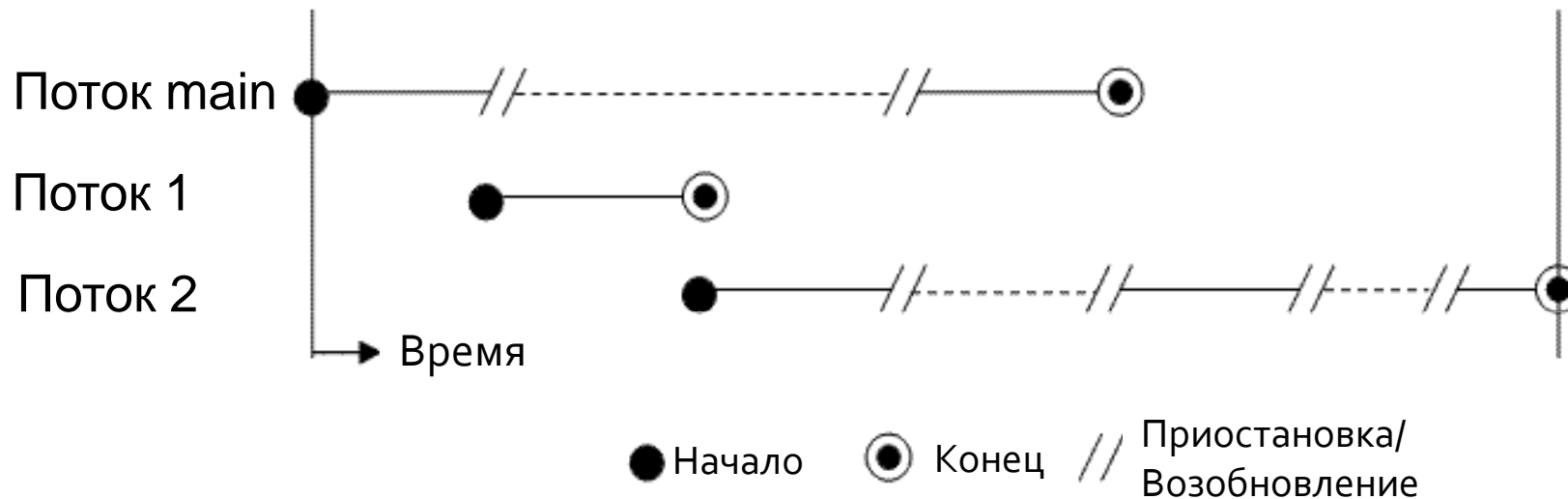
- ❑ **Потоки** (threads) – это задачи, на которые разбивается процесс и которые могут исполняться параллельно.
- **Процесс** – это программа, запущенная на исполнение.



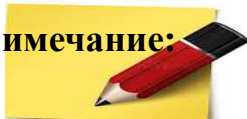
- ✓ Однопоточная программа имеет единственную точку входа (метод *main()*) и единственную точку выхода.
- ✓ Многопоточная программа имеет начальную точку входа (метод *main()*), а затем следует много точек входа и выхода, которые могут работать совместно с *main()*.

Потоки исполнения


Пример программы с тремя потоками, которая выполняется в однопроцессорной системе:



Примечание:



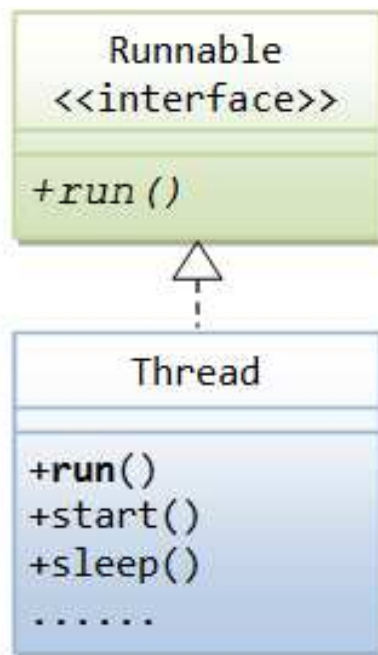
- Поток не является самостоятельной программой, потому что он не может работать сам по себе;
- Он работает в рамках процесса (программы);
- Он разделяет доступ к ресурсам процесса.



Создание и управление потоками исполнения

ПЕРВЫЙ СПОСОБ СОЗДАНИЯ ПОТОКА

- ❑ Расширить класс **Thread**
 - Описать подкласс класса **Thread**, в котором переопределить метод *run()*;
 - Создать экземпляр описанного подкласса;
 - Вызвать метод *start()* на созданном объекте.



Примечание:

1. Тело метода *run()* и есть тело потока.
2. Поток может быть запущен на исполнение только вызовом метода *start()* класса *Thread* или его подкласса.

Потоки исполнения

Пример 1:

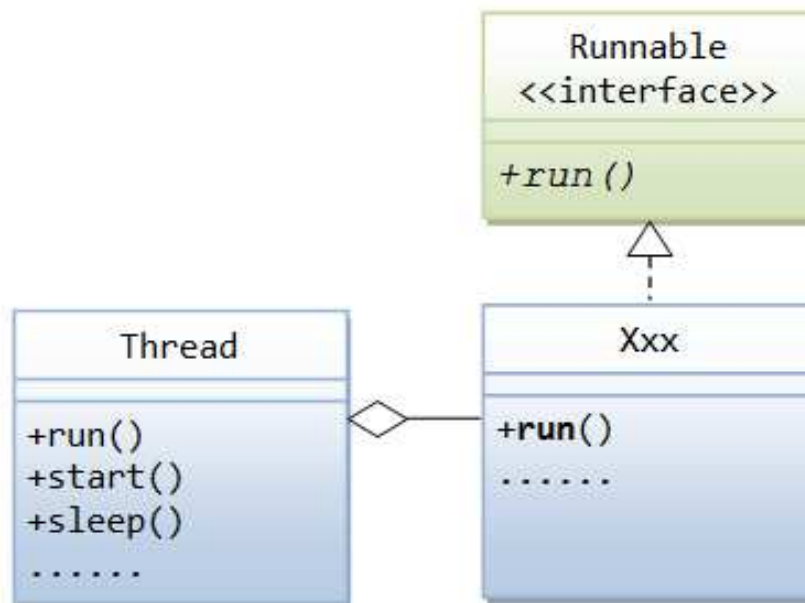
```
public class Talk extends Thread {  
    public void run() {  
        for (int i = 0; i < 5; i++) {  
            System.out.print(this.getName() + ": ");  
            System.out.println("Talking " + (i+1));  
        }  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        System.out.println("main start");  
        Talk talk = new Talk();  
        talk.start();  
        System.out.println("main ended!");  
    }  
}
```

Вывод в консоли:

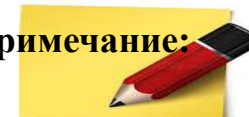
```
main start  
main ended!  
Thread-0: Talking 1  
Thread-0: Talking 2  
Thread-0: Talking 3  
Thread-0: Talking 4  
Thread-0: Talking 5
```

ВТОРОЙ СПОСОБ СОЗДАНИЯ ПОТОКА

- ❑ Реализовать интерфейс **Runnable**
 - Описать класс, реализующий интерфейс **Runnable**;
 - Создать объект класса **Thread**, которому передать ссылку на объект описанного класса;
 - Вызвать метод *start()* на объекте класса **Thread**.



Примечание:



Интерфейс **Runnable**
содержит только
один метод - *run()*

Потоки исполнения

Пример 2:

```
public class Walk implements Runnable {  
    public void run() {  
        String name = Thread.currentThread().getName();  
        for (int i = 0; i < 5; i++) {  
            System.out.println(name + ": Walking " + (i+1)) ;  
        }  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        System.out.println("main start");  
        Thread myThread = new Thread(new Walk());  
        myThread.start();  
        System.out.println("main ended!");  
    }  
}
```

Вывод в консоли:

```
main start  
main ended!  
Thread-0: Walking 1  
Thread-0: Walking 2  
Thread-0: Walking 3  
Thread-0: Walking 4  
Thread-0: Walking 5
```


Потоки исполнения



Какой способ использовать?

❑ Через наследование **Thread**

- Легче использовать в простых приложениях;
- Однако поток не может быть наследником никакого другого класса;
- Есть возможность переопределить поведение управлением потоком (различные методы класса Thread);

❑ Через реализацию **Runnable**

- Более гибкий, так поток может быть наследником любого класса;
- Отделено тело потока от управления потоком;
- Используется в высокоуровневом управлении потоками.

УПРАВЛЕНИЕ ПОТОКАМИ

В классе **Thread** описаны методы для управления выполнением потоков:

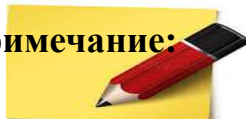
<i>Метод</i>	<i>Назначение</i>
isAlive()	Определение выполняется ли поток
join()	Ожидание для текущего потока завершения указанного
sleep()	Приостановка потока на заданный интервал времени
wait()	Блокировка/приостановка выполнения потока
notify()	Возобновление выполнения потока

Потоки исполнения

□ Статический метод *sleep()*

- заставляет текущий поток приостановить выполнение на указанный период;
- время обычно указывается в миллисекундах (первая форма, с одним параметром);
- время можно указать дополнительно к миллисекундам еще и в наносекундах (вторая форма, с двумя параметрами);
- бросает исключение типа **InterruptedException** если некоторый поток прерывает тот, что вызвал метод *sleep()*.

Примечание:



Это эффективное средство создания процессорного времени, доступного для других потоков приложения или других приложений, которые могут быть запущены на компьютере.

Потоки исполнения

Пример 3:

```
public class MyThread extends Thread {  
    public void run() {  
        for (int i = 0; i < 3; i++) {  
            System.out.println("Thread:" + getName() + " i=" + i);  
            try { Thread.sleep(1000);  
            } catch (InterruptedException e) { }  
        }  
    }  
}  
  
public class DemoSleep {  
    public static void main(String[] args) {  
        new MyThread().start();  
        new MyThread().start();  
    }  
}
```

Вывод в консоли:

```
Thread:Thread-0 i=0  
Thread:Thread-1 i=0  
Thread:Thread-0 i=1  
Thread:Thread-1 i=1  
Thread:Thread-0 i=2  
Thread:Thread-1 i=2
```

Потоки исполнения

❑ Метод экземпляра `join()`

- позволяет одному потоку "присоединиться в конец" другого потока (если поток ***B*** не может выполнять свою работу, пока поток ***A*** не завершит свою, тогда необходимо, чтобы поток ***B*** "примкнул" к потоку ***A***. Это означает, что поток ***B*** не будет работоспособным, пока не закончится поток ***A***);

Например,

```
Thread t = new Thread();  
t.start();  
t.join();
```

Это означает "присоединить текущий поток к концу потока ***t***, так чтобы ***t*** закончился прежде, чем текущий поток сможет работать снова".

Потоки исполнения

Пример 4:

```
public class MyThread1 extends Thread {  
    public void run() {  
        for (int i = 0; i < 4; i++) {  
            System.out.println("Thread:" + getName()  
                               + " i=" + i);  
  
            try {  
                Thread.sleep(500);  
            } catch (InterruptedException e) { }  
        }  
    }  
}  
// ....
```

Потоки исполнения

Продолжение примера 4:

```
public class DemoJoin {  
    public static void main(String[] args) {  
        System.out.println("main method start");  
        MyThread1 thr1 = new Thread();  
        thr1.start();  
        System.out.println("thread started");  
        try {  
            thr1.join();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println("main method end");  
    }  
}
```

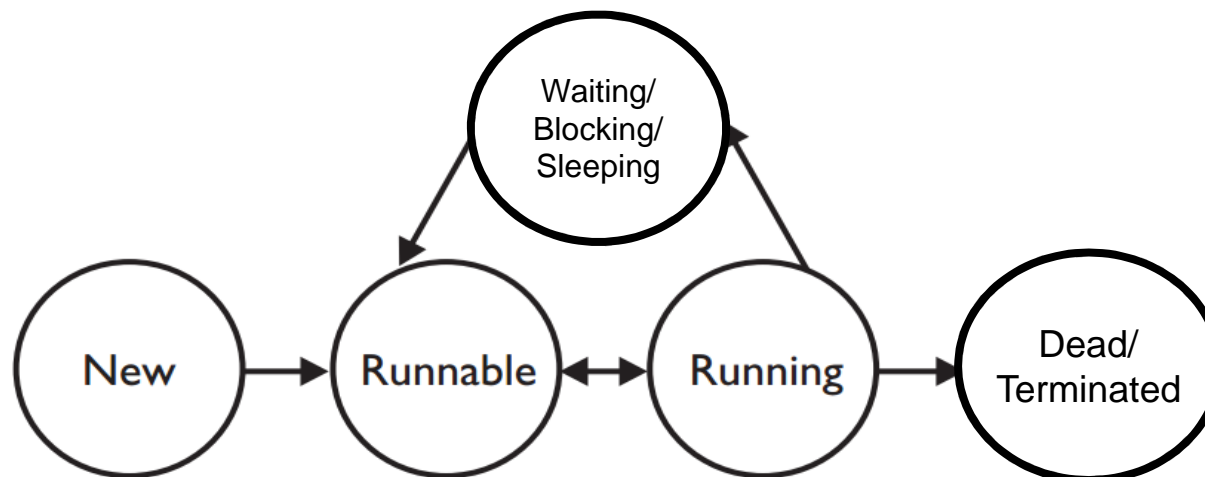
Вывод в консоли:

```
main method start  
thread started  
Thread:Thread-0 i=0  
Thread:Thread-0 i=1  
Thread:Thread-0 i=2  
Thread:Thread-0 i=3  
main method end
```

Потоки исполнения

СОСТОЯНИЯ ПОТОКА

- 1) Созданный (New)
- 2) Работоспособный (Runnable)
- 3) Работающий (Running)
- 4) Блокированный/ожидающий/спящий (Waiting/Blocking/Sleeping)
- 5) Остановленный (Dead/Terminated)



Потоки исполнения

- ❑ Созданный (новый) - это состояние потока после создания экземпляра **Thread**, но метод *start()* для потока не был вызван (поток не считается живым);
- ❑ Работоспособный (запущенный) - это состояние потока, когда он может выполняться, но планировщик потоков не выбрал его, чтобы выполнять (на потоке вызван метод *start()* и теперь поток считается живым);
- ❑ Работающий (выполняющийся) - это состояние потока когда планировщик потоков выбирает его из пула работоспособных, чтобы поток в данный момент стал выполняемым;
- ❑ Ожидающий/блокированный/спящий - это состояние потока, когда он не может быть выбранным для выполнения (поток все еще жив, но в настоящее время не имеет права работать, т.е. он может вернуться к состоянию работоспособный позже, если произойдет определенное событие);
- ❑ Остановленный (мертвый) - поток считается мертвым, когда его метод *run()* завершен (если вызвать метод *start()* на мертвом экземпляре **Thread**, то произойдет исключение времени выполнения).

Потоки исполнения



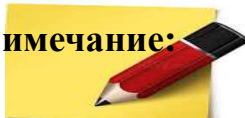
СОСТОЯНИЕ «ОЖИДАЮЩИЙ/БЛОКИРОВАННЫЙ/СПЯЩИЙ»

- ❑ Состояние «блокированный» - поток может быть заблокирован в ожидании ресурса (в этом случае событие, которое отправляет его обратно в состояние «работоспособный» является наличие ресурсов);
- ❑ Состояние «спящий» - для потока запущен *sleep()* на некоторый период времени (в этом случае, событие, которое отправляет его обратно в состояние «работоспособный» - это окончание его времени сна);
- ❑ Состояние «ожидаящий» - поток вызывает метод *wait()* (в этом случае, событие, которое отправляет его обратно в состояние «работоспособный» - это отправка уведомления о том, что он больше может не ждать, из в другого потока).

ПЛАНИРОВЩИК ПОТОКОВ

- ❑ Один из подходов планирования выполнения нескольких потоков – это **квантование времени** (каждому потоку выделяется достаточное количество времени, а затем он отправляется обратно в состояние работоспособный, чтобы дать шанс другому потоку);
- ❑ Большинство JVM используют **планировщик потоков** (упреждающее основанное на приоритетах планирование – позволяет один поток постоянной работы, пока не завершится его метод *run()*);
- ❑ Приоритет может варьироваться от 1 (Thread.MIN_PRIORITY) по 10 (Thread.MAX_PRIORITY). По умолчанию - 5 (Thread.NORM_PRIORITY).

Примечание:



В любой момент времени, исполняемый поток, как правило, будет иметь приоритет, который не ниже, чем приоритет любого из потоков в пуле работоспособных.

Потоки исполнения

Пример 5:

```
public class MyThread2 extends Thread {
    public void run() {
        for (int i = 0; i < 4; i++)
            System.out.println("Thread:" + getName() + " i=" + i);
    }
}

public class DemoPriority {
    public static void main(String[] args) {
        MyThread2 min_thr = new MyThread2();
        min_thr.setName("Thread Min");
        min_thr.setPriority(Thread.MIN_PRIORITY);
        MyThread2 max_thr = new MyThread2();
        max_thr.setName("Thread Max");
        max_thr.setPriority(Thread.MAX_PRIORITY);
        MyThread2 norm_thr = new MyThread2();
        norm_thr.setName("Thread Norm");
        norm_thr.setPriority(Thread.NORM_PRIORITY);
        min_thr.start();
        norm_thr.start();
        max_thr.start();
    }
}
```

Вывод в консоли:

```
Thread:Thread Max i=0
Thread:Thread Max i=1
Thread:Thread Max i=2
Thread:Thread Max i=3
Thread:Thread Norm i=0
Thread:Thread Norm i=1
Thread:Thread Norm i=2
Thread:Thread Norm i=3
Thread:Thread Min i=0
Thread:Thread Min i=1
Thread:Thread Min i=2
Thread:Thread Min i=3
```

ПРЕРЫВАНИЕ ПОТОКА

- ❑ **Прерывание** - это указание потоку, что он должен остановить выполняемую работу и сделать что-то другое (разработчик должен решить, как именно поток реагирует на прерывание, обычно это завершение);



Как поток поддерживает свое прерывание?

- ❑ Если поток часто вызывает методы, которые бросают **InterruptedException**, то он просто возвращается из этих методов и ловит это исключение;
- ❑ Если поток выполняется долгое время без вызова метода, который бросает **InterruptedException**, тогда он должен периодически вызывать метод **Thread.interrupted()**, который возвращает истину, если был получен запрос на прерывание.

Потоки исполнения

- ❑ Запрос на прерывание определяется статусом флага прерывания:
 - флаг прерывания устанавливается вызовом метода ***interrupt()*** для потока, который подлежит прерыванию.

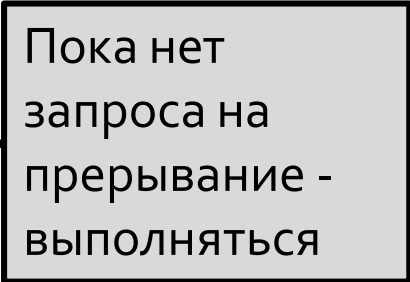


- ✓ Статический метод ***interrupted()*** после проверки запроса на прерывание очищает флаг прерывания;
- ✓ Метод экземпляра ***isInterrupted()*** только проверяет запрос на прерывание без очистки флага прерывания;
- ✓ Любой метод, который бросает ***InterruptedException***, очищает флаг прерывания.

Потоки исполнения

Пример 6:

```
public class MyThread3 extends Thread {  
    public void run() {  
        int i = 1;  
        while( !isInterrupted() ) {  
            System.out.println("Thread:" + getName()+ " i=" + i++);  
            try {  
                sleep(1000);  
            } catch (InterruptedException e) {  
                return;  
            }  
        }  
    }  
}  
// ....
```



Пока нет
запроса на
прерывание -
выполняться

Потоки исполнения

Продолжение примера 6:

```
public class DemoInterrupt {  
    public static void main(String[] args) {  
        MyThread3 th1 = new MyThread3();  
        th1.start();  
        try {  
            Thread.sleep(5000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        th1.interrupt();  
    }  
}
```

Уступить
процессорное
время
запущенному
потoku

Вывод в консоли:

```
Thread:Thread-0 i=1  
Thread:Thread-0 i=2  
Thread:Thread-0 i=3  
Thread:Thread-0 i=4  
Thread:Thread-0 i=5  
Thread:Thread-0 i=6
```


ФОНОВЫЕ ПОТОКИ (ПОТОКИ-ДЕМОНЫ)

❑ **Поток-демон** – это поток, который может выполняться на фоне основных потоков и используется только для их обслуживания (*например*, как таймер, который посылает сигналы другим потокам через определенные интервалы времени).



- Если в программе работоспособными остаются только фоновые потоки, то программа завершает работу;
- Определить поток как фоновый поток можно до его запуска на исполнение вызовом метода *setDaemon(boolean isDaemon)*;
- Метод *isDaemon()* позволяет определить, является ли текущий поток фоновым или нет.

Потоки исполнения

Пример 7, только рабочие (пользовательские) потоки

```
public class MyThread4 extends Thread {  
    public void run() {  
        for (int i = 0; i < 6; i++) {  
            System.out.println(getName() + ", i=" + i);  
            try { Thread.sleep(1000);  
            } catch (InterruptedException e) { }  
        }  
    }  
}  
  
public class Demo {  
    public static void main(String[] args) {  
        new MyThread4().start();  
        try { Thread.sleep(2000);  
        } catch (InterruptedException e) { }  
        System.out.println("method main() finished");  
    }  
}
```

Вывод в консоли:

```
Thread-0, i=0  
Thread-0, i=1  
method main() finished  
Thread-0, i=2  
Thread-0, i=3  
Thread-0, i=4  
Thread-0, i=5
```

Потоки исполнения


Пример 8, рабочие (пользовательские) и фоновые потоки

```
public class MyThread5 extends Thread {  
    public void run() {  
        for (int i = 0; i < 6; i++) {  
            System.out.println(getName() + ", i=" + i);  
            try { Thread.sleep(1000);  
            } catch (InterruptedException e) { }  
        }  
    }  
}  
  
public class DemoDaemon {  
    public static void main(String[] args) {  
        MyThread5 tt = new MyThread5();  
        tt.setDaemon(true);  
        tt.start();  
        try { Thread.sleep(2000);  
        } catch (InterruptedException e) { }  
        System.out.println("method main() finished");  
    }  
}
```

Вывод в консоли:

```
Thread-0, i=0  
Thread-0, i=1  
method main() finished
```

← Установка потока как фонового



Использование общих ресурсов

ОБЩЕНИЕ МЕЖДУ ПОТОКАМИ

- ❑ Потоки общаются при совместном доступе к ресурсам (переменным);
- ❑ Эта форма общения производит два вида возможных ошибок:
 - столкновения (вмешательство) потоков;
 - согласованность памяти.
- ❑ Необходимый инструмент для предотвращения этих ошибок - это **синхронизация**.
 - Однако, синхронизация приводит к конкуренции потоков (когда два или более потока пытаются получить доступ к одному ресурсу одновременно) и заставляет исполнительную систему Java выполнять один или несколько потоков более медленно или даже приостановить их исполнение.

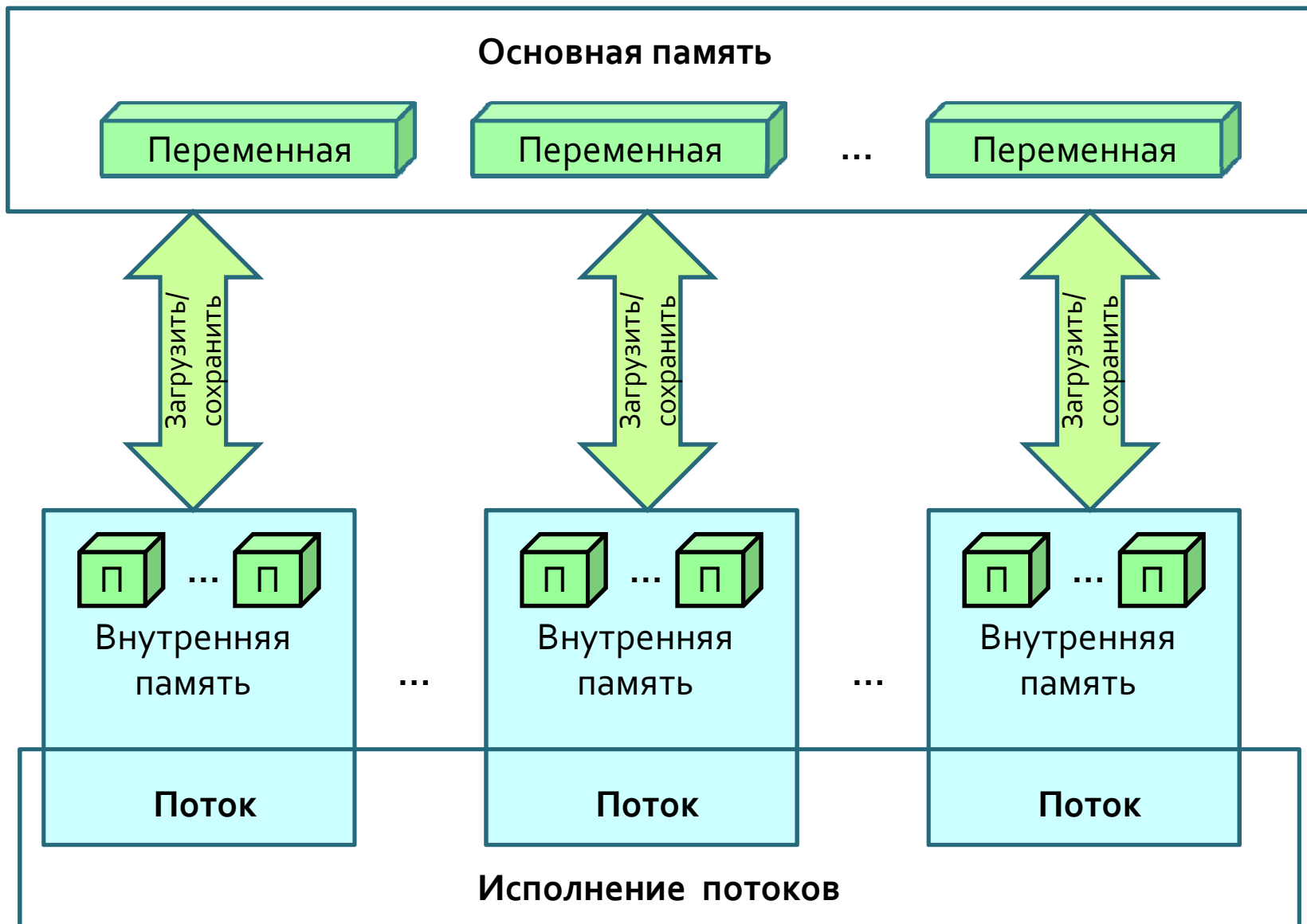
Потоки исполнения



□ Использование общих ресурсов:

- любая переменная перед использованием потоком всегда считывается из основной памяти и любая переменная записываемая потоком, всегда записывается в основную память;
- в целях повышения эффективности работы JRE сохраняет локальную копию переменной в каждом потоке, который ссылается на нее (эти «внутрипоточные» копии переменных помогают избежать проверки основной памяти каждый раз, когда требуется доступ к значению переменной);
- нет никаких гарантий относительно того, когда JVM считывает/записывает данные внутренней памяти из/в основную память (т.е. изменения, проводимые с «внутрипоточной» копией, не обязательно сразу же видны другим потокам);

Потоки исполнения



Потоки исполнения



❑ Ключевое слово **volatile**

➤ *гарантирует видимость изменений в переменной через все потоки:*

- ✓ переменная всегда считывается из основной памяти, и никогда не сохраняется в память потока, а значит, всегда доступна любому потоку;
- ✓ при запросах на чтение и запись от нескольких потоков, системой гарантируется выполнение вначале запросов на запись;
- ✓ гарантируется атомарность операций чтения/записи (для всех прочих операций, как ++, должна делаться синхронизация);
- ✓ потоки не блокируются в ожидании освобождения ресурса.

Потоки исполнения

Пример 9:

```
public class VolatileTest {  
    private static volatile int varVolat = 0;  
    private static int varNonVolat = 0;  
    public static void main(String[] args) {  
        ChangeListener thread1 = new ChangeListener();  
        ChangeMaker thread2 = new ChangeMaker();  
        thread1.start();  
        thread2.start();  
        try { Thread.sleep(500);  
        } catch (InterruptedException e) { }  
        thread1.interrupt();  
        thread2.interrupt();  
    }  
    // ...  
}
```

Потоки исполнения

Продолжение примера 9:

```
static class ChangeMaker extends Thread {  
    @Override  
    public void run() {  
        int local_value = 0;  
        while ( !isInterrupted() ) {  
            varVolat = varNonVolat = ++local_value;  
        }  
    }  
}  
// ...
```

Потоки исполнения

Продолжение примера 9:

```
static class ChangeListener extends Thread {  
    @Override  
    public void run() {  
        while ( !isInterrupted() ) {  
            if ( varVolat != varNonVolat ) {  
                System.out.println("Error: " + varVolat +  
                                   " != " + varNonVolat);  
            }  
        }  
    }  
}
```

Вывод в консоли:

```
Error:99! = 148  
Error:117887! = 117900  
Error:119837! = 119843  
Error:121639! = 121645  
Error:123476! = 123482  
Error:125276! = 125284
```

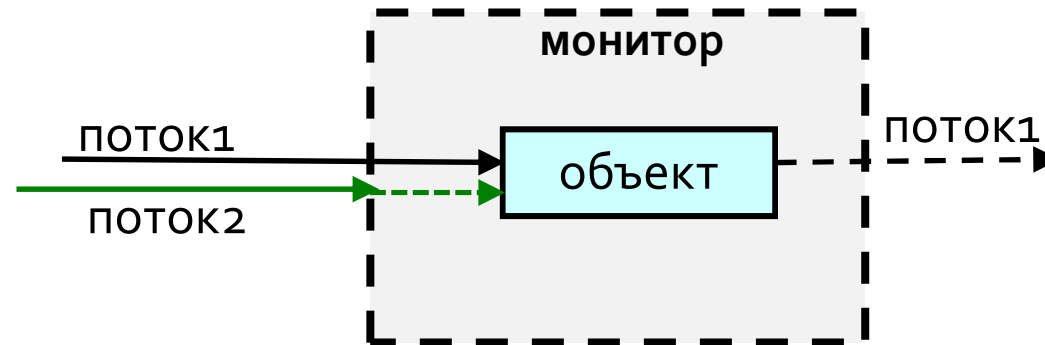
ВЫВОД

- ❑ Использование переменных **volatile** снижает риск *ошибок согласованности памяти*, потому что любая запись в переменную **volatile** устанавливает отношение "происходит до" с последующим чтением этой же переменной;
- ❑ Все изменения в переменной **volatile** всегда видимы для других потоков;
- ❑ Когда поток читает переменную **volatile**, он видит не только последние изменения в **volatile**, но и побочные эффекты кода, который произвел изменения.

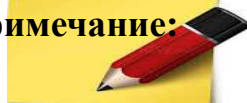
ВНЕШНЯЯ СИНХРОНИЗАЦИЯ ПОТОКОВ

- ❑ **Синхронизация** – это процесс упорядочивания (разделения) доступа к ресурсам между несколькими потоками;
- ❑ В синхронизации используется понятие **монитора** – это объект, который применяется для блокировки доступа к общему ресурсу;
- ❑ Когда поток получает блокировку, то говорят, что он «вошел в монитор»;
- ❑ Все другие потоки, пытающиеся ввести монитор на этом же объекте, будут приостановлены (говорят, что они «ожидают монитор»).

Потоки исполнения



Примечание:



- ❑ В языке Java каждый объект имеет:
 - неявный монитор (объект блокировки);
 - связанное с ним неявное условие (правило использования монитора).

СПОСОБЫ БЛОКИРОВКИ

- ❑ Использование синхронизированных методов;
- ❑ Использование синхронизированных блоков;
- ❑ Использование API высокого уровня из пакета **java.util.concurrent.locks**

I) Синхронизированные методы

- ❑ В описание метода добавляется ключевое слово **synchronized**.

<доступ> **synchronized** <описание метода >

Потоки исполнения



- ❑ Когда JVM выполняет `synchronized`-метод, то выполняющий поток определяет, что в `method_info` установлен флаг **ACC_SYNCHRONIZED**
 - Тогда поток автоматически ставит блокировку на объект, вызывает метод и снимает блокировку при выходе из метода;
 - Если метод бросается исключение, то поток автоматически снимает блокировку.
- ❑ Когда один поток выполняет `synchronized`-метод для объекта, все остальные потоки, которые вызывают `synchronized`-методы для того же объекта блокируются (приостанавливают выполнение) пока первый поток не оставит объект;
- ❑ Когда `synchronized`-метод завершается, то он автоматически устанавливает отношение "происходит до" с любым последующим вызовом `synchronized`-метода на том же объекте (что гарантирует – изменения в состоянии объекта видны всем потокам).

Потоки исполнения

Пример 10, без синхронизации

```
class Synchro {
    private File f = new File("d:\\data.txt");
    public Synchro() {
        try {
            f.delete();
            f.createNewFile();
        } catch (IOException e) { e.printStackTrace(); }
    }
    public void writing(String str, int i) {
        try {
            RandomAccessFile raf = new RandomAccessFile(f, "rw");
            raf.seek(raf.length());
            System.out.print(str);
            raf.writeBytes(str);
            Thread.sleep((long) (Math.random() * 15));
            System.out.println("->" + i);
            raf.writeBytes("->" + i + " ");
        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Потоки исполнения

```
public class SynchroThreads {  
    public static void main(String[] args) {  
        Synchro s = new Synchro();  
        T t1 = new T("First", s);  
        T t2 = new T("Second", s);  
        t1.start();  
        t2.start();  
    }  
}  
  
class T extends Thread {  
    private String str;  
    private Synchro s;  
    public T(String str, Synchro s) {  
        this.str = str;  
        this.s = s;  
    }  
    public void run() {  
        for (int i = 0; i < 5; i++)  
            s.writing(str, i);  
    }  
}
```

Вывод в консоли:

```
SecondFirst->0  
Second->0  
First->1  
Second->1  
First->2  
First->3  
First->2  
Second->4  
->3  
Second->4
```

Содержимое
файла

```
Second->0 t->0 nd->1 t->1 nd->2 t->2 First->3 First->4 nd->3 Second->4
```

Потоки исполнения

Добавим синхронизацию:

```
class Synchro {  
    private File f = new File("d:\\data.txt");  
    public Synchro() {  
        try {  
            f.delete();  
            f.createNewFile();  
        } catch (IOException e) { e.printStackTrace(); }  
    }  
    public synchronized void writing(String str, int i) {  
        try {  
            RandomAccessFile raf = new RandomAccessFile(f, "rw");  
            raf.seek(raf.length());  
            System.out.print(str);  
            raf.writeBytes(str);  
            Thread.sleep((long) (Math.random() * 15));  
            System.out.println("->" + i);  
            raf.writeBytes("->" + i + " ");  
        } catch (IOException | InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Вывод в консоли:

```
First->0  
First->1  
First->2  
First->3  
First->4  
Second->0  
Second->1  
Second->2  
Second->3  
Second->4
```

Потоки исполнения

II) Синхронизированные блоки

- ❑ Это блок кода отмеченный ключевым словом **synchronized**;
- ❑ Синхронизированный блок всегда синхронизируется на каком-то объекте;

```
synchronized (object) {  
    // операторы, подлежащие синхронизации  
}
```

- ❑ Все синхронизированные блоки, синхронизированные на одном и том же объекте, могут иметь только один выполняемый поток внутри него одновременно;
- ❑ Все остальные потоки, пытающиеся войти в синхронизированный блок, блокируются, пока поток внутри синхронизированного блока не выйдет из него.

Потоки исполнения

Пример 11, без синхронизации

```
public class TwoThread {  
    public static void main(String args[]) {  
        final StringBuffer s = new StringBuffer();  
        new Thread() {  
            public void run() {  
                int i = 0;  
                while (i++ < 3) {  
                    s.append("A");  
                    try { sleep(15); }  
                    catch (InterruptedException e) {  
                        e.printStackTrace();  
                    }  
                }  
                System.out.println(s);  
            }  
        }.start();  
    }  
}
```

Первый поток

Общая строка

Код,
изменяющий
строку

Потоки исполнения

Продолжение примера 11:

Второй поток

Код,
изменяющий
строку

```
new Thread() {  
    public void run() {  
        int j = 0;  
        while (j++ < 3) {  
            s.append("B");  
            try { sleep(10);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            System.out.println(s);  
        }  
    }  
}.start();  
}  
}
```

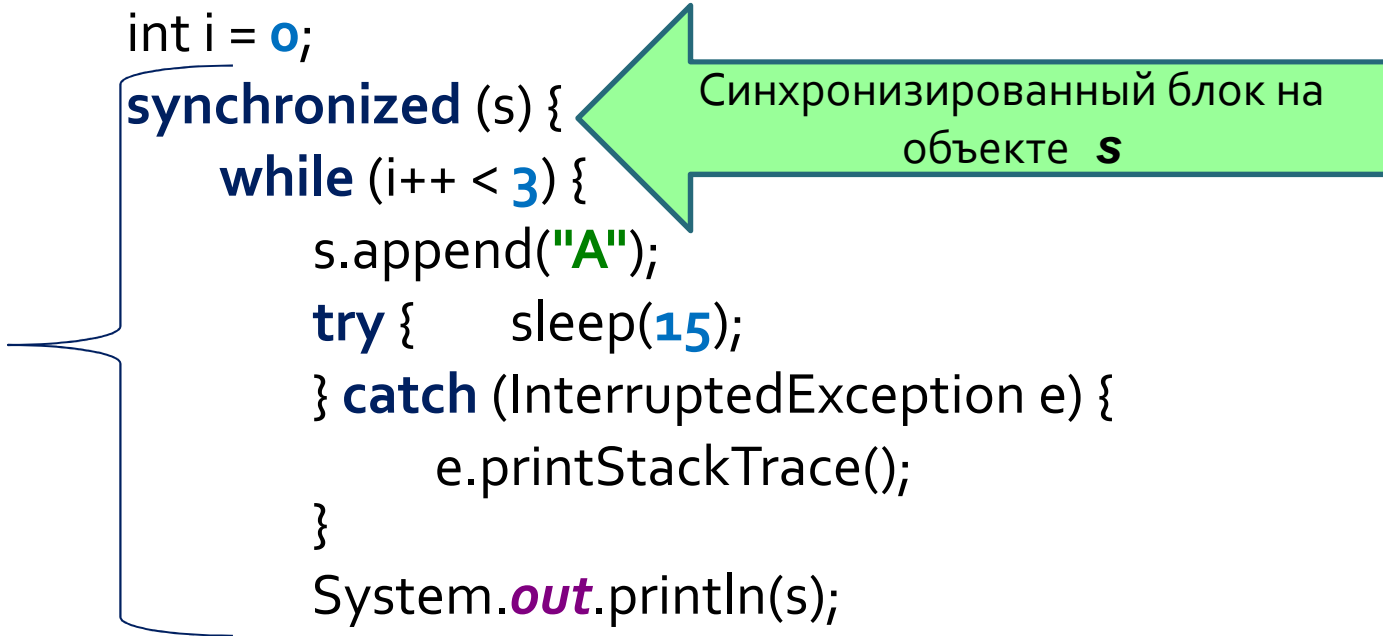
Вывод в консоли:

```
BA  
BAB  
BABBB  
BABBA  
BABBA  
BABBA
```

Потоки исполнения

Добавим синхронизацию на блоках

```
public class TwoThread {  
    public static void main(String args[]) {  
        final StringBuffer s = new StringBuffer();  
        new Thread() {  
            public void run() {  
                int i = 0;  
                synchronized (s) {  
                    while (i++ < 3) {  
                        s.append("A");  
                        try { sleep(15); }  
                        catch (InterruptedException e) {  
                            e.printStackTrace();  
                        }  
                    }  
                    System.out.println(s);  
                }  
            }  
        }.start();  
    }  
}
```



The diagram illustrates the synchronization of the `run()` method in the `TwoThread` class. A large green arrow points from the text "Синхронизированный блок на объекте **s**" to the `synchronized (s)` block in the code. A blue bracket on the left side of the code groups the `run()` method and its `synchronized` block, indicating that this specific thread's execution is synchronized on the `s` object.

Потоки исполнения

// ...

```
new Thread() {
```

```
    public void run() {
```

```
        int j = 0;
```

```
        synchronized (s) {
```

```
            while (j++ < 3) {
```

```
                s.append("B");
```

```
                try {    sleep(10);
```

```
            } catch (InterruptedException e) {
```

```
                e.printStackTrace();
```

```
            }
```

```
            System.out.println(s);
```

```
        } } } } .start();
```

```
    }
```

```
}
```

Синхронизированный блок на
объекте **s**

Вывод в консоли:

A

AA

AAA

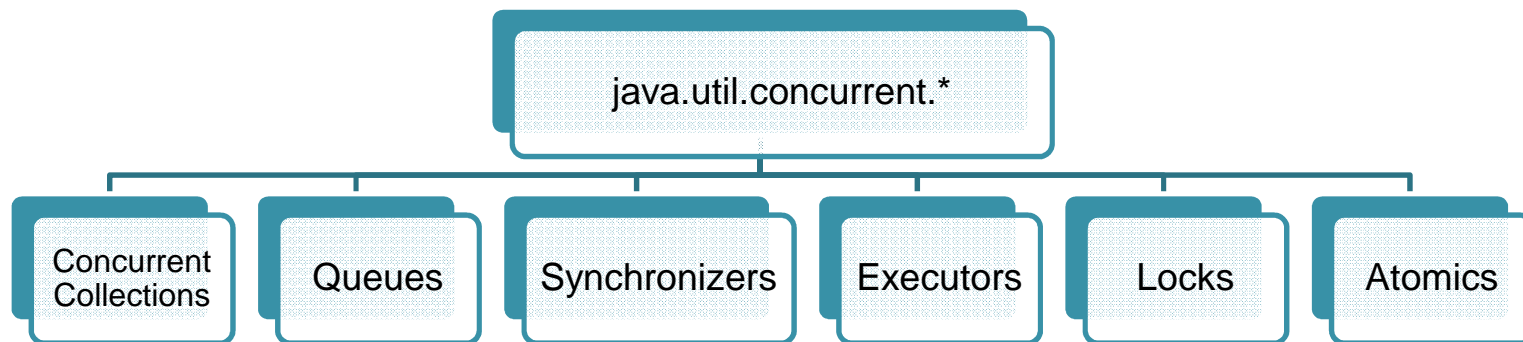
AAAB

AAABBB

AAABBBB

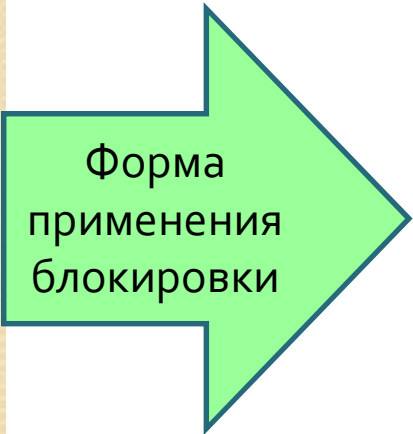
III) Пакет `java.util.concurrent`

- ❑ Начиная с версии 5.0, платформа Java имеет высокоуровневый параллельный API в пакете **`java.util.concurrent`**:
 - Этот пакет включает несколько стандартных расширяемых фреймворков, а также классы, которые обеспечивают полезную функциональность и утомительные/трудные реализации.



Потоки исполнения

- ❑ Ключевое слово **synchronized** не поддерживает справедливое поведение:
 - поток может быть заблокирован в ожидании "открытия ресурса" в течение неопределенного периода времени и нет никакого способа, чтобы управлять этим ожиданием.
- ❑ Интерфейс **Lock** обеспечивает возможность различного приобретения блокировки для управления ожиданием;
- ❑ Класс **ReentrantLock** - это конкретная реализация интерфейса **Lock**.



Форма
применения
блокировки

```
object.lock();                // войти в монитор
try {
    // фрагмент кода
} finally {
    object.unlock();          //выйти из монитора
}
object – объект класса ReentrantLock
```

Потоки исполнения

Методы интерфейса *Lock*

- *lock()* – если блокировка не доступна, то текущий поток будет приостановлен до тех пор, пока блокировка будет снята;
- *lockInterruptible()* – если блокировка не доступна, то текущий поток будет приостановлен, но дается возможность прервать ожидающий поток (для ожидающего блокировки потока вызывается метод *interrupt()* и ожидание будет прервано, а значит поток попыток доступа к защищенному ресурсу не делает, а продолжают любые другие действия);
- *tryLock()* – если блокировка не доступна на момент вызова, то возвращается **false** (в этом случае можно отказаться от ожидания и выполнять какие-либо другие действия);
- *tryLock(long timeout, TimeUnit timeUnit)* – если блокировку нельзя получить после истечения **timeout**, то возвращается значение **false**, обозначающее невозможность получения блокировки);
- *unlock()* – снять блокировку.

Потоки исполнения

Пример 12:

```
class MyCounter {  
    private long count = 0;
```

```
    Lock lock = new ReentrantLock();
```

```
    public void increment() {
```

```
        lock.lock();
```

```
        try {    count++;
```

```
        } finally {
```

```
            lock.unlock();
```

```
        }
```

```
    }
```

```
    public long getValue() {
```

```
        return count;
```

```
    }
```

```
}
```

```
// ...
```

Класс как счетчик

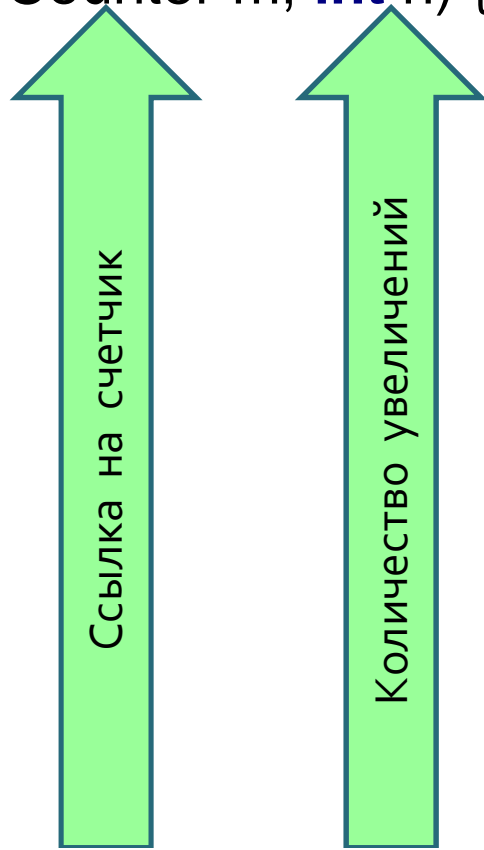
Метод, увеличивающий значение счетчика

Получить значение счетчика

Потоки исполнения

Продолжение примера 12:

```
class MyCounterThread extends Thread {  
    MyCounter meter;  
    int number;  
    public MyCounterThread(MyCounter m, int n) {  
        this.meter = m;  
        this.number = n;  
    }  
    public void run() {  
        for(int i=0;i<number;i++)  
            meter.increment();  
    }  
}  
// ...
```



Ссылка на счетчик

Количество увеличений

Потоки исполнения

Продолжение примера 12:

```
public class DemoCounter {  
    public static void main(String[] args) {  
        MyCounter meter = new MyCounter();  
        MyCounterThread[] tgs = new MyCounterThread[100];  
        for (int i = 0; i < 100; i++)  
            tgs[i] = new MyCounterThread(meter, 1_000_000);  
        for (MyCounterThread t : tgs)  
            t.start();  
        try {  
            for (MyCounterThread t : tgs) {  
                t.join();  
            }  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
        System.out.println(meter.getValue());  
    }  
}
```

Создали счетчик

Создали массив
ссылок на потоки

Создали потоки

Запустили
потоки

Примкнули поток
main к
завершению всех
запущенных
потоков

Получили рез-т

Вывод в консоли:
100000000

АТОМАРНЫЕ ПЕРЕМЕННЫЕ

- ❑ *Атомарные переменные* – это переменные атомарных классов;
- ❑ *Атомарные классы* гарантируют, что определённые операции будут выполняться потокобезопасно (например, операции инкремента и декремента, обновления и добавления значения);
- ❑ Атомарные классы описаны в пакете ***java.util.concurrent.atomic*** (включает такие классы как ***AtomicInteger***, ***AtomicBoolean***, ***AtomicLong***, ***AtomicIntegerArray*** и так далее);

Примечание:



Атомарные переменные необходимы при сложных (составных) операциях чтения/записи (например, инкремент/декремент) без внешней синхронизации - на переменных *volatile* они не выполняются атомарно.

Потоки исполнения

- Например, следующая операция на переменной **volatile** является небезопасной при использовании несколькими потоками:

myVolatileVar++;

Это все равно что:

```
int temp = 0;  
synchronized (myVolatileVar) {  
    temp = myVolatileVar;  
}  
temp++;  
synchronized (myVolatileVar) {  
    myVolatileVar = temp;  
}
```

Операция
инкремента не
атомарна

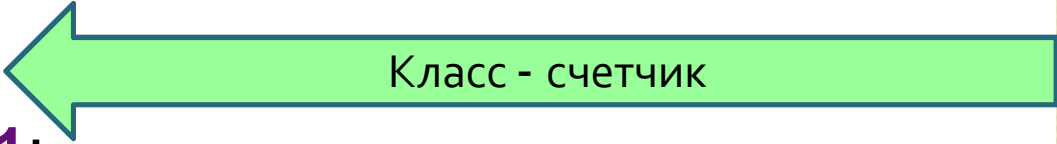
Чтение
синхронизировано
(атомарно)

Запись
синхронизирована
(атомарно)

Потоки исполнения

Пример 13, использования переменных: обычная, атомарная и **volatile**:

```
class MyCounter {  
    public int count1;  
    public volatile int count2;  
    public AtomicInteger count3 =  
        new AtomicInteger(0);  
}  
// ...
```

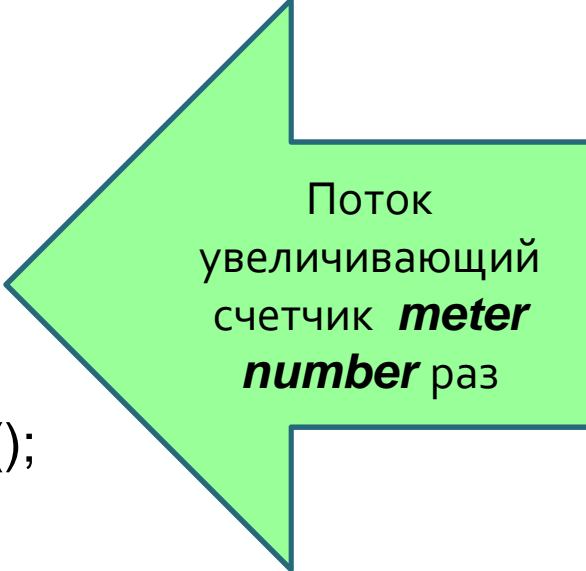


Класс - счетчик

Потоки исполнения

Продолжение примера 13:

```
class MyCountThread extends Thread {  
    MyCounter meter;  
    int number;  
    public MyCountThread(MyCounter m, int n) {  
        this.meter = m;  
        this.number = n;  
    }  
    public void run() {  
        for(int i=0; i<number; i++) {  
            this.meter.count1++;  
            this.meter.count2++;  
            this.meter.count3.getAndIncrement();  
        }  
    }  
}  
// ...
```



Поток
увеличивающий
счетчик ***meter***
number раз

Потоки исполнения

Продолжение примера 13:

```
public static void main(String[] args) {  
    MyCounter meter = new MyCounter();  
    MyCountThread[] tgs = new MyCountThread[100];  
    for (int i = 0 ; i < 100; i++)  
        tgs[i] = new MyCountThread(meter, 1_000_000);  
    for (MyCountThread thread : tgs)  
        thread.start();  
    try {  
        for (MyCountThread thread : tgs)  
            thread.join();  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
    System.out.println("int: " + meter.count1 + "\nvolatile: "  
        + meter.count2 + "\nAtomic: " + meter.count3);  
}
```

Вывод в консоли:

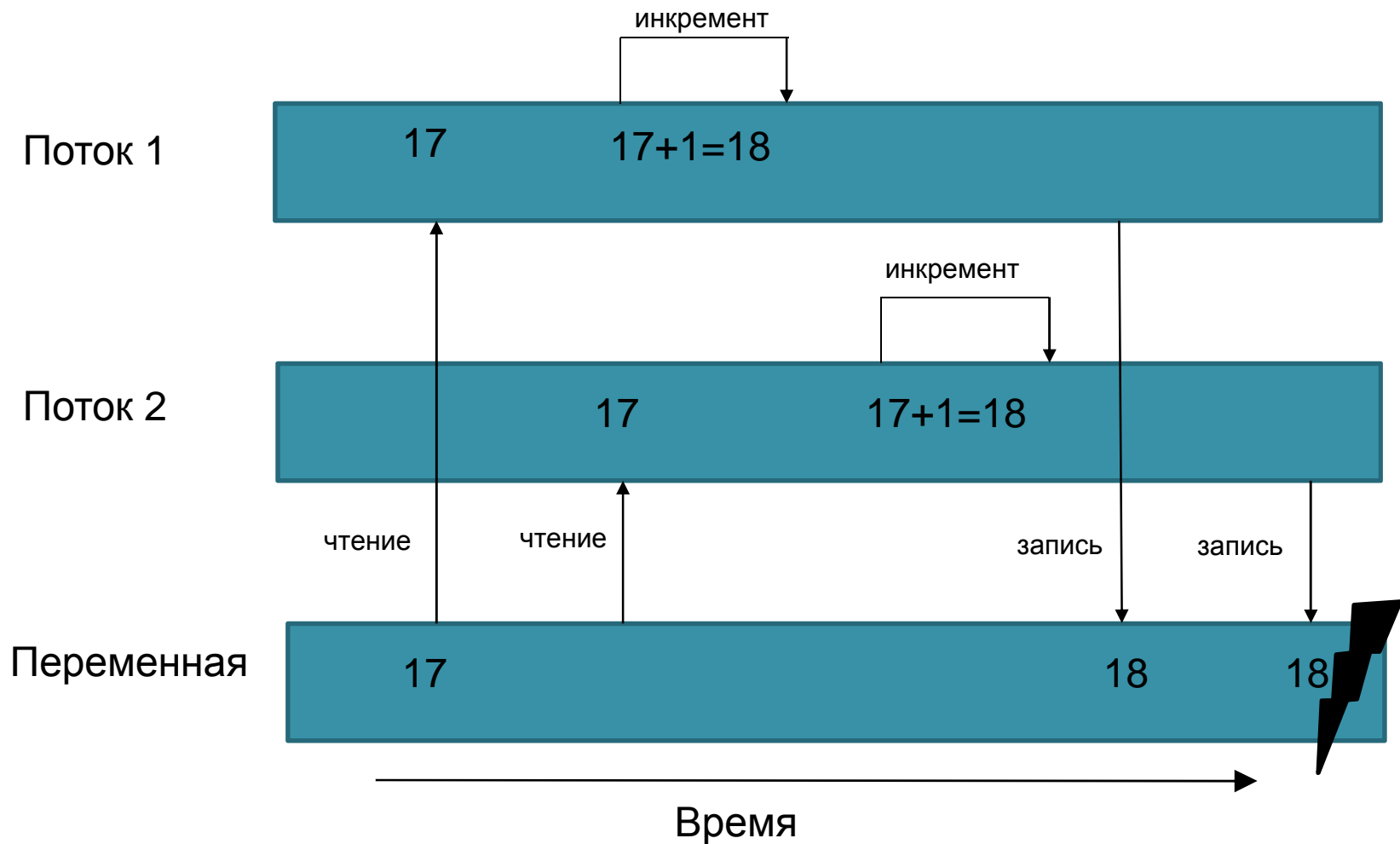
int: 49793826

volatile: 99998132

Atomic: 100000000

Потоки исполнения

- ❑ **volatile** переменная обновляется неявно, т.е. значение читается, изменяется, а затем присваивается как новое.





Межпоточное взаимодействие

Потоки исполнения

- ❑ Потокам часто приходится координировать свои действия;
- ❑ Наиболее распространенная идея координации – это *осмотрительный (осторожный) блок*;
- ❑ Такой блок начинается с проверки условия продолжения своей работы:
 - если результат **true** – поток выполняется;
 - если результат **false** – поток уступает выполнение другому потоку, пока не получит уведомления, что возможно он может возобновить свою работу;
- ❑ В классе **Object** есть методы, с помощью которых можно организовать осмотрительный блок:
 - *wait()*
 - *notify()*
 - *notifyAll()*

МЕТОДЫ КЛАССА Object

❑ *final void wait() throws InterruptedException*

- уступить монитор другим потокам до тех пор, пока не вернут управление;

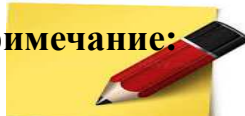
❑ *final void notify()*

- послать уведомление о пробуждении любого потока, который вызывал метод *wait()* на том же объекте;

❑ *final void notifyAll()*

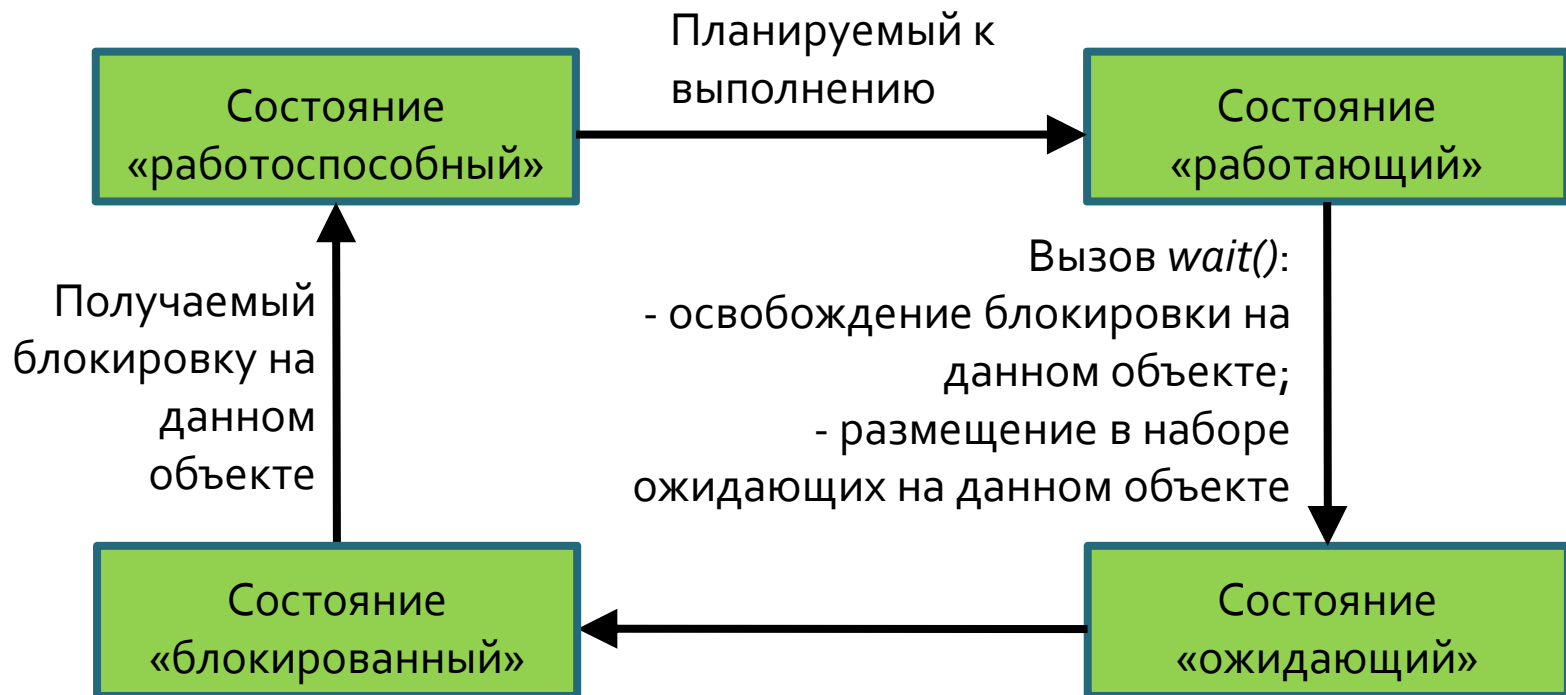
- послать уведомление о пробуждении всех потоков, которые вызывали метод *wait()* на том же объекте.

Примечание:



Эти методы вызываются только из синхронизированных методов и блоков!

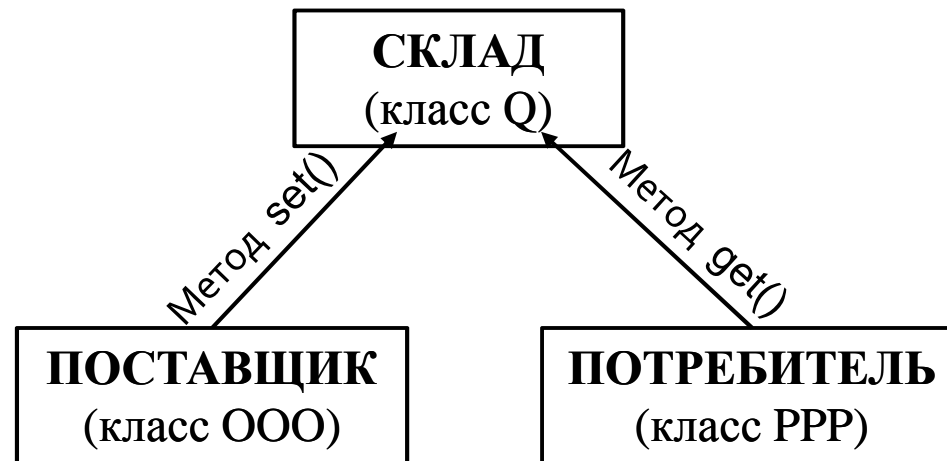
Механизм межпоточного взаимодействия (wait/notify)



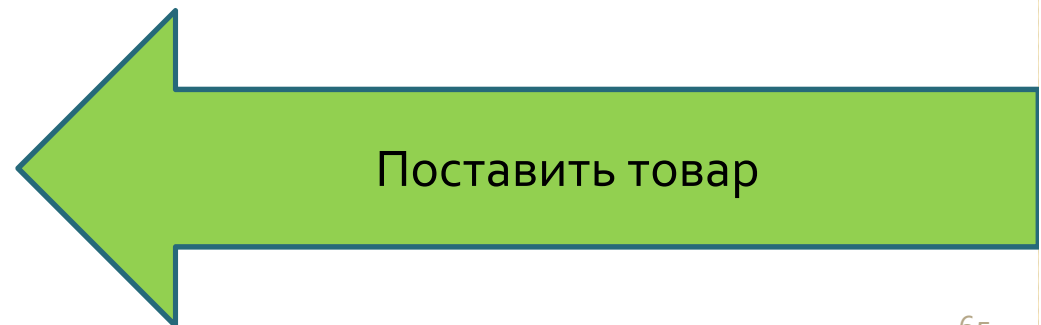
Вызов *notify()* или *notifyAll()* исполняющимся потоком:
- когда приходит уведомление о возможности возобновления, ожидающие потоки пытаются получить блокировку на данном объекте и при отказе переходят в набор блокированных

Потоки исполнения

Пример 14:



```
class OOO implements Runnable {
    Q stock;
    OOO(Q stock) {
        this.stock = stock;
        new Thread(this, "Поставщик").start();
    }
    public void run() {
        int number = 0;
        while (true)
            stock.set(++number);
    }
}
```



Потоки исполнения

Продолжение примера 14:

```
class PPP implements Runnable {
```

```
    Q stock;
```

```
    PPP(Q stock) {
```

```
        this.stock = stock;
```

```
        new Thread(this, "Потребитель").start();
```

```
    }
```

```
    public void run() {
```

```
        while (true)
```

```
            stock.get();
```

```
    }
```

```
}
```

```
class Q {
```

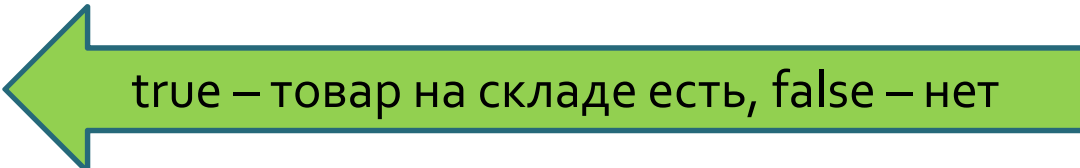
```
    int amount;
```

```
    boolean value = false;
```

```
// ...
```



Получить товар



true – товар на складе есть, false – нет

Потоки исполнения

Продолжение примера 14:

```
synchronized int get() {  
    while ( !value)  
        try {    wait();  
        } catch (InterruptedException e) { System.out.println(e);    }  
    System.out.println("Получено: " + amount);  
    value = false;    notify();  
    return amount;  
}  
  
synchronized void set(int n) {  
    while (value)  
        try {    wait();  
        } catch (InterruptedException e) { System.out.println(e);    }  
    amount = n;  
    System.out.println("Отправлено: " + amount);  
    value = true;    notify();  
}  
}
```

Ждать пока на складе не появится товар

Забрав товар, информировать поставщика

Ждать пока на складе не появится место

Поставив товар, информировать потребителя

Потоки исполнения

Продолжение примера 14:

```
public class Main {  
    public static void main(String[] args) {  
        Q obj = new Q();  
        new OOO(obj);  
        new PPP(obj);  
        System.out.println("Для остановки нажмите Ctrl+C");  
    }  
}
```

Вывод в консоли:

```
Для остановки нажмите Ctrl+C  
Отправлено: 1  
Получено: 1  
Отправлено: 2  
Получено: 2  
Отправлено: 3  
Получено: 3  
Отправлено: 4  
Получено: 4  
Отправлено: 5  
Получено: 5
```

Потоки исполнения

- ❑ Пример 15, межпоточного взаимодействия с использованием высокоуровневого API управления потоками:
 - Есть класс **Bank** предоставляющий возможность:
 - ✓ передачи средств с одного счета на другой внутри этого банка (метод *trans(...)*);
 - ✓ проверки, что общий баланс средств не изменяется при передаче средств (метод *getBalance()*);
 - Есть класс-поток **Transfer**, который отвечает за работу с одним счетом банка.

Потоки исполнения

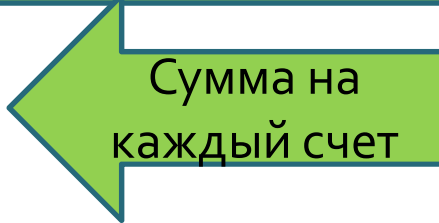
```
import java.util.concurrent.locks.*;
public class Main {
    public static final int count = 100;

    public static final double balance = 1000.0;

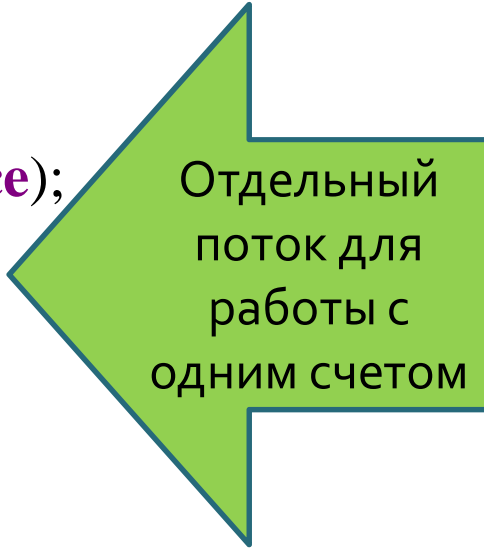
    public static void main(String[] args) {
        Bank b = new Bank(count, balance);
        for (int i=0; i<count; i++) {
            Transfer tr = new Transfer(b, i, balance);
            Thread t = new Thread(tr);
            t.start();
        }
    }
}
// ...
```



Количество счетов



Сумма на
каждый счет



Отдельный
поток для
работы с
одним счетом

Потоки исполнения

Продолжение примера 15:

```
class Bank {  
    private double kol[];  
    private Lock bl;  
    private Condition con;  
    Bank(int n, double bal) {  
        kol = new double[n];  
        for (int i=0; i< kol.length; i++)  
            kol[i] = bal;  
        bl = new ReentrantLock();  
        con = bl.newCondition();  
    }  
    // ...  
}
```



Массив счетов



Объект для создания блокировки



Условие использования блокировки

Потоки исполнения

Продолжение примера 15:

```
public void trans(int from, int to, double sum) throws InterruptedException{  
    bl.lock();  
    try {  
        while (kol[from] < sum)  
            con.await();  
        System.out.print(Thread.currentThread());  
        kol[from] -= sum;  
        System.out.print(sum + " from " + from + " to" + to);  
        kol[to] += sum;  
        System.out.println(" Общий счет = " + getBalance());  
        con.signalAll();  
    } finally {  
        bl.unlock();  
    }  
}  
// ...
```

Установка блокировки (монитор)

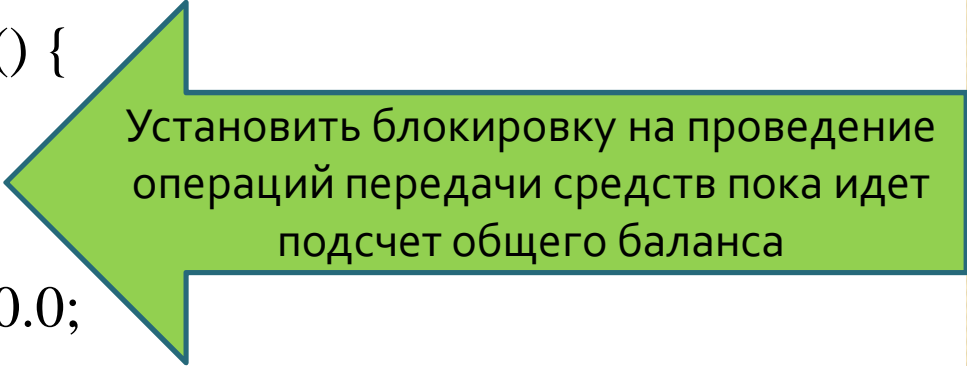
Если на счете не хватает средств, то снять монитор и подождать

Уведомить все потоки

Потоки исполнения

Продолжение примера 15:

```
public double getBalance() {  
    bl.lock();  
    try {  
        double summa = 0.0;  
        for (double aa : kol)  
            summa += aa;  
        return summa;  
    } finally {  
        bl.unlock();  
    }  
}  
// ...
```



Установить блокировку на проведение операций передачи средств пока идет подсчет общего баланса

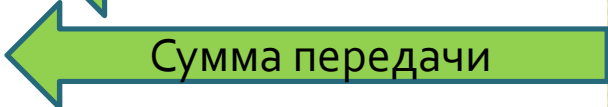
Потоки исполнения

Продолжение примера 15:

```
class Transfer implements Runnable {  
    private Bank bb;    private int fromCh;    private double maxSum;  
    Transfer(Bank b, int ch, double sum) {  
        bb = b;    fromCh = ch;    maxSum = sum;  
    }  
    public void run() {  
        try {  
            while (true) {  
                int toCh = (int)(Math.random() * bb.length);  
                double ss = Math.random() * maxSum;  
                bb.trans(fromCh, toCh, ss);  
                Thread.sleep((int)(Math.random() * 10));  
            }  
        } catch (InterruptedException e) { }  
    }  
}
```



Номер счета



Сумма передачи

Потоки исполнения

Вывод в консоли:

```
Thread[Thread-0,5,main]631.5678341951805 from 0 to76  Общий счет = 100000.0
Thread[Thread-4,5,main]667.7495298034709 from 4 to93  Общий счет = 100000.0
Thread[Thread-6,5,main]368.5470445389627 from 6 to60  Общий счет = 100000.0
Thread[Thread-8,5,main]822.0696809826526 from 8 to76  Общий счет = 100000.0
Thread[Thread-10,5,main]613.2017543471802 from 10 to87  Общий счет = 100000.0
Thread[Thread-12,5,main]256.2414126252944 from 12 to62  Общий счет = 100000.0
Thread[Thread-2,5,main]361.05886076801187 from 2 to33  Общий счет = 100000.0
Thread[Thread-14,5,main]266.136098485431 from 14 to3  Общий счет = 100000.0
Thread[Thread-16,5,main]761.3484165280794 from 16 to63  Общий счет = 100000.0
Thread[Thread-18,5,main]463.5929520514993 from 18 to15  Общий счет = 100000.0
Thread[Thread-20,5,main]467.65272999493925 from 20 to30  Общий счет = 100000.0
Thread[Thread-22,5,main]484.50765025036713 from 22 to86  Общий счет = 100000.0
Thread[Thread-24,5,main]495.18479102135205 from 24 to16  Общий счет = 100000.0
.....
```

Потоки исполнения

Использование класса `ReentrantReadWriteLock`

- ❑ Если в некотором процессе присутствуют в большем количестве потоки «читающие» данные, чем «записывающие», то имеет смысл предоставить совместный доступ «читающим» потокам:
 - Создать объект класса **`ReentrantReadWriteLock`**;
 - Получить из него объекты блокировки чтения и записи (методы *`readLock()`* и *`writeLock()`* соответственно);
 - Использовать блокировку чтения при чтении данных, а блокировку записи – при модификации данных.

Потоки исполнения

Например, изменения в примере 15:

```
class Bank {
```

```
.....
```

```
private ReentrantReadWriteLock rwl;
```

```
private Lock readLock;
```

```
private Lock writeLock;
```

```
.....
```

```
rwl = new ReentrantReadWriteLock();
```

```
readLock = rwl.readLock();
```

```
writeLock = rwl.writeLock();
```

```
.....
```

```
}
```

Изменить поля
класса для
организации
блокировки

Изменить
операторы
конструктора

Потоки исполнения

```
public double getBalance() {  
    readLock.lock();  
    try {  
        .....  
    } finally {  
        readLock.unlock();  
    }  
}
```

Изменения в методе
подсчета общего
баланса на
блокировку чтения

```
public void trans(int from, int to, double sum) throws InterruptedException {  
    writeLock.lock();  
    try {  
        .....  
    } finally {  
        writeLock.unlock();  
    }  
}
```

Изменения в методе
перевода средств на
блокировку записи



ВЫСОКОУРОВНЕВЫЙ ИНТЕРФЕЙС УПРАВЛЕНИЯ ПОТОКАМИ



Потоки исполнения

Потокобезопасные наборы данных

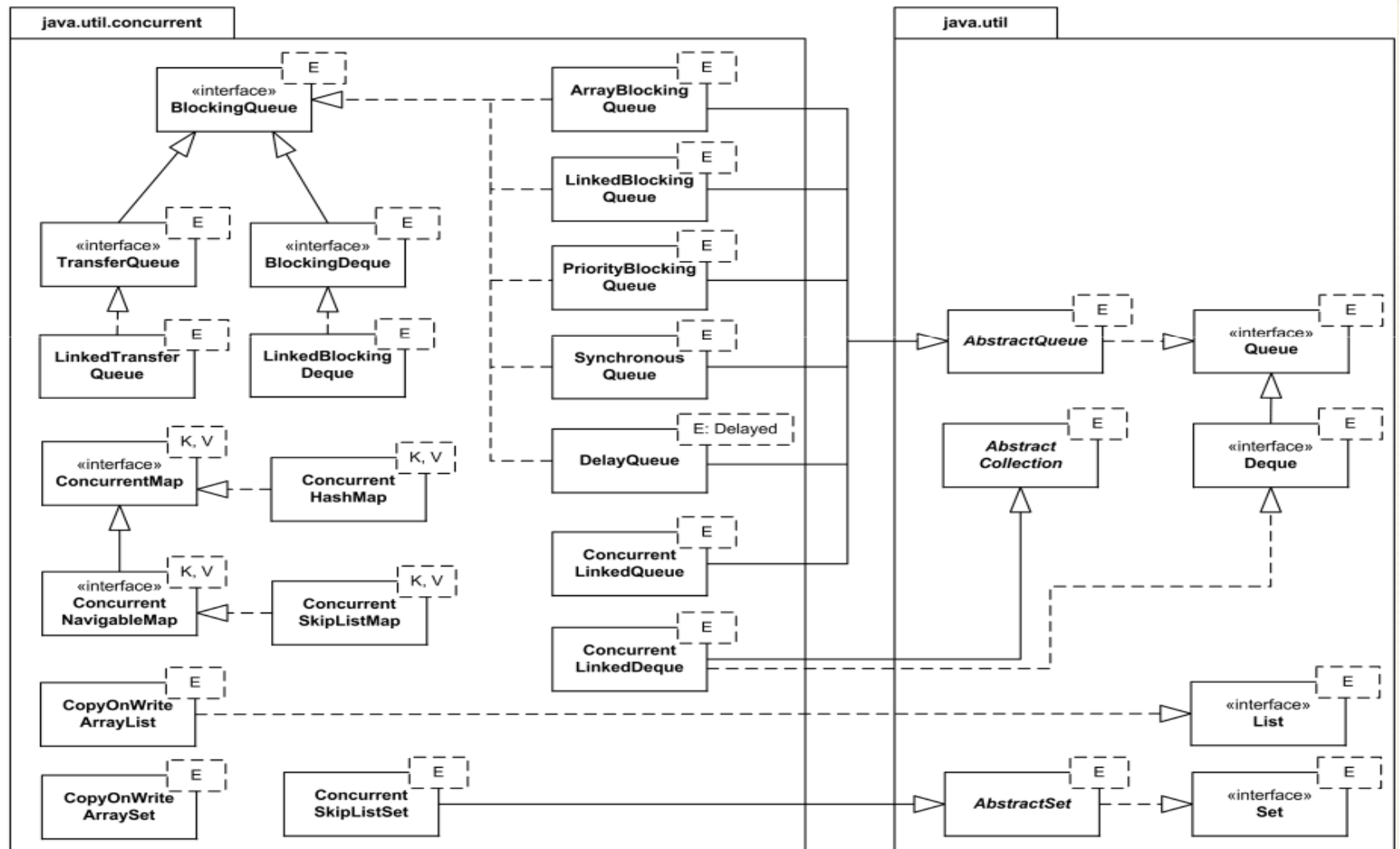
- ❑ Синхронизированные коллекции

Способы управления потоками

- ❑ Механизм асинхронного исполнения
- ❑ Механизм управления заданиями, основанный на пуле потоков
- ❑ Механизм синхронизаторов общего назначения

Потоки исполнения

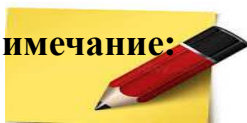
- ❑ Синхронизированные коллекции – это потокобезопасные реализации нескольких коллекций:



Потоки исполнения

- **BlockingQueue** определяет структуру данных FIFO, которая блокирует потоки каждый раз, когда вы попытаетесь добавить к полной очереди или извлечь из пустой очереди;
- **ConcurrentMap** - это подинтерфейс **java.util.Map**, который определяет атомарные операции удаления, замены или добавления пары ключ-значение (это помогает избежать синхронизации). Стандартная реализация - **ConcurrentHashMap** - аналог **HashMap**;
- **ConcurrentNavigableMap** - это подинтерфейс **ConcurrentMap**, который поддерживает приближенные сопоставления. Стандартная реализация - **ConcurrentSkipListMap** - аналог **TreeMap**.

Примечание:

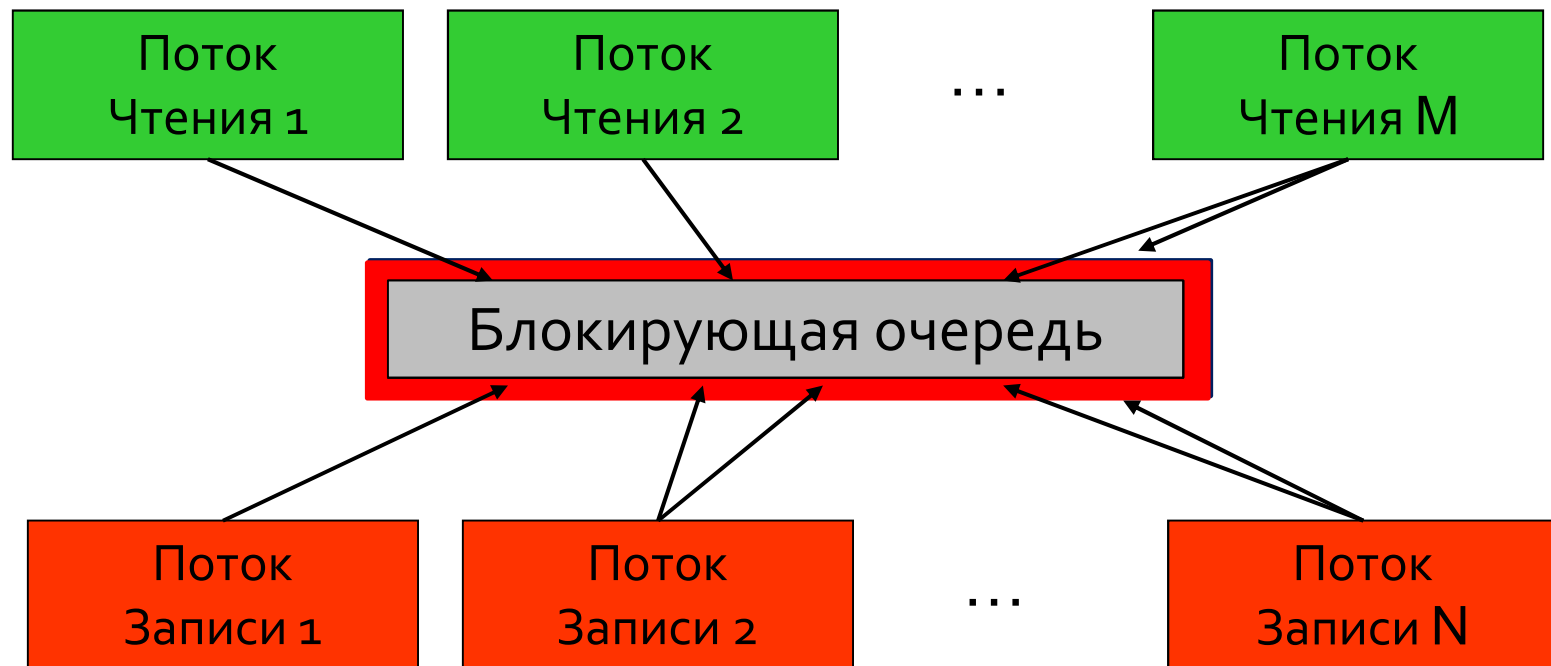


Эти коллекции помогают избежать ошибок согласованности памяти, определяя отношения "происходит до" между операциями, которые добавляют объект в коллекцию с последующими операциями, которые имеют доступ или удаляют этот объект.

Потоки исполнения

I) Блокирующие очереди

- Инструмент для координации действий нескольких потоков.
(Например, для задач когда одни потоки создают данные, а другие используют)



Потоки исполнения

Создание блокирующей очереди

- Класс *ArrayBlockingQueue* (обязательно указать размер)
- Класс *LinkedBlockingQueue* (неограниченный размер)
- Класс *PriorityBlockingQueue* (неограниченный размер)
- Класс *DelayQueue* (неограниченный размер)

Для последнего типа особенности:

- ✓ выбираются только те элементы, чья задержка истекла;
- ✓ располагаются элементы по убыванию срока окончания задержки;
- ✓ если ни у одного элемента в очереди не истекло время задержки, очередь блокируется для выборки.

Потоки исполнения

Пример 16:

```
public static void main(String[] args) {  
    Scanner sc = new Scanner(System.in);  
    System.out.print("Enter directiry -> ");  
    String dir = sc.next();  
    System.out.print("Enter keyWord -> ");  
    String word = sc.next();  
  
    BlockingQueue <File> que = new ArrayBlockingQueue <File> (10);  
    FileRunTask running = new FileRunTask(que, new File(dir));  
    new Thread(running).start();  
    for (int i=0; i<50; i++)  
        new Thread(new SearchTask(que, word)).start();  
}
```

Потоки исполнения

Продолжение примера 16:

```
public class FileRunTask implements Runnable {  
    private BlockingQueue <File> que;  
    private File startDir;  
    public static File EXIT = new File("");  
  
    public FileRunTask(BlockingQueue <File> que, File startDir) {  
        this.que = que;  
        this.startDir = startDir;  
    }  
    public void run() {  
        try {  
            runDir(startDir);  
            que.put(EXIT);  
        } catch (InterruptedException e) { }  
    }  
}
```

Потоки исполнения

Продолжение примера 16:

```
public void runDir(File dir) throws InterruptedException {  
    File[] files = dir.listFiles();  
    for (File ff: files)  
        if (ff.isDirectory())  
            runDir(ff);  
        else  
            que.put(ff);  
    }  
}
```

Потоки исполнения

Продолжение примера 16:

```
public class SearchTask implements Runnable {  
    private BlockingQueue <File> que;  
    private String word;  
    SearchTask(BlockingQueue <File> que, String word) {  
        this.que = que;  
        this.word = word;  
    }  
    public void search(File ff) throws IOException {  
        Scanner sc = new Scanner(new FileInputStream(ff));  
        while (sc.hasNextLine()) {  
            String str = sc.nextLine();  
            if (str.contains(word))  
                System.out.println(ff.getPath() + " -> " + str);  
        }  
        sc.close();  
    }  
}
```


Потоки исполнения

Продолжение примера 16:

```
public void run() {  
    try {  
        while (true) {  
            File ff = que.take();  
            if (ff == FileRunTask.EXIT) {  
                que.put(ff);  
                break;  
            }  
            else search(ff);  
        }  
    } catch (IOException e) { e.printStackTrace();  
    } catch (InterruptedException e) { }  
}
```

Потоки исполнения

Вывод в консоли:

Enter directiry -> d:/Projects/Tasm/text

Enter keyWord -> add

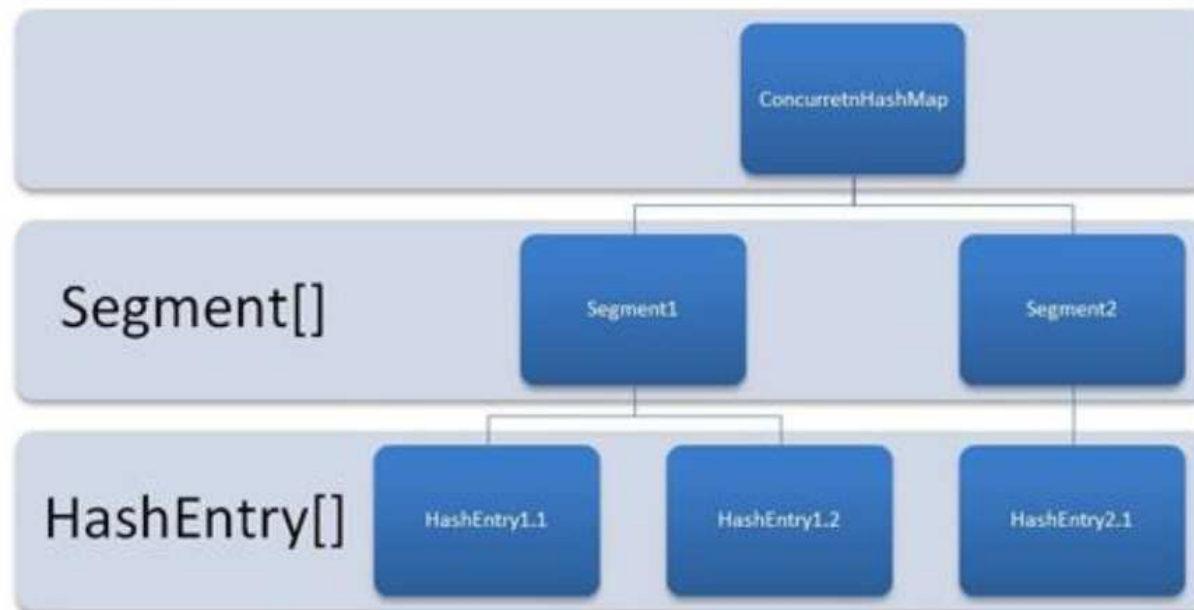
d:\Projects\Tasm\text\exitproc.asm ->	add	ax,bx
d:\Projects\Tasm\text\exitproc.asm ->	add	ax,200h
d:\Projects\Tasm\text\exitproc.asm ->	add	ax,1
d:\Projects\Tasm\text\L5_V6.ASM ->	add	ax, min1
d:\Projects\Tasm\text\L6_V7.asm ->	add	ax, C
d:\Projects\Tasm\text\L6_V7.asm ->	add	ax, bx
d:\Projects\Tasm\text\L6_V7.asm ->	add	dx, '0'
d:\Projects\Tasm\text\F2.asm ->	add	si, 4
d:\Projects\Tasm\text\Oleg.asm ->	add	rez, ax
d:\Projects\Tasm\text\Preob.asm ->	add	ax, bx
d:\Projects\Tasm\text\Preob.asm ->	add	dx, '0'
d:\Projects\Tasm\text\pr_seg.asm ->	add	ax, 2
d:\Projects\Tasm\text\pr_seg.asm ->	add	bx, ax
d:\Projects\Tasm\text\pr_seg.asm ->	add	dx, 2
d:\Projects\Tasm\text\rgr_14.asm ->	add	mx1, ax
d:\Projects\Tasm\text\rgr_14.asm ->	add	cx, len
d:\Projects\Tasm\text\rgr_14.asm ->	add	dx, len

.....

Потоки исполнения

II) Синхронизированные классы

- ❑ Класс *ConcurrentHashMap* позволяет:
 - избежать блокирования всей структуры данных;
 - минимизировать конфликты, допуская одновременный доступ к разным частям структуры данных.



Потоки исполнения

- ❑ Конструктор **ConcurrentHashMap** принимает три параметра:
 - *initialCapacity* – начальная емкость;
 - *loadFactor* – коэффициент загрузки;
 - *concurrencyLevel* – количество сегментов (по умолчанию 16).
- ❑ Количество сегментов будет выбрано как ближайшая степень двойки, большая чем *concurrencyLevel*. Ёмкость каждого сегмента, соответственно, будет определяться как отношение округлённого до ближайшей большей степени двойки значения ёмкости карты *initialCapacity*, к полученному количеству сегментов.
- ❑ Между хэш-кодами ключей и соответствующими им сегментами устанавливается зависимость на основе применения к старшим разрядам хэш-кода битовой маски.
- ❑ Операции чтения не требуют блокировок и выполняются параллельно (т.к. элементы объявлены как ***volatile***).
- ❑ Операции записи также могут выполняться без блокировок, если происходят в разных сегментах.

Потоки исполнения

- ❑ Сравнение производительности некоторой программы, которая использовала классы *ConcurrentHashMap* и *Hashtable*

Количество потоков	<i>ConcurrentHashMap</i>	<i>Hashtable</i>
1	1.0	1.03
2	2.59	32.4
4	5.58	78.23
8	13.21	163.48
16	27.58	341.21
32	57.27	778.41