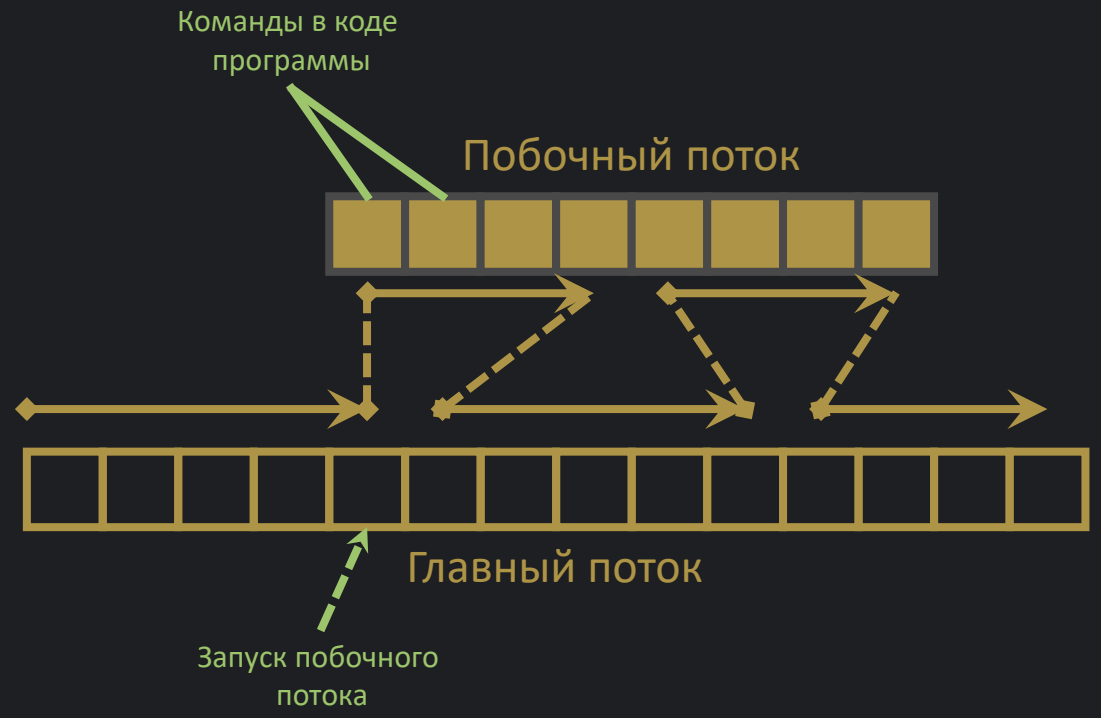
Java: Threads

Что такое поток?

Поток (ветвь исполнения) в Java — это отдельная последовательность выполнения команд в коде. Каждый из них работает независимо от других.

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



Что такое поток?

Каждый поток в Java представлен объектом класса Thread.

Существует несколько способов создания потока:
- Наследование от класса Thread и переопределение метода run().
- Реализация интерфейса Runnable и передача его экземпляра в конструктор Thread.

И класс Thread, и интерфейс Runnable относятся к библиотеке java.lang.

Методы Thread.

long getId()	- получение идентификатора потока
String getName()	- получение имени потока
int getPriority()	- получение приоритета потока
void run()	- запуск потока, если поток был создан с использованием интерфейса Runnable
void start()	- запуск потока
void sleep()	- останавливает поток на указанное количество миллисекунд

Методы Runnable.

void run()	- запуск потока

Создание потока через Thread

Код

```java
public class ThreadExtension1 {
    public static void main(String[] args) {

        try {
            MyThread myThread = new MyThread();
            myThread.start();
            System.out.println("Main thread: " + myThread.getName() + " started");
            Thread.sleep( millis: 1000);
            System.out.println("Main thread: finished");
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

    }
}

class MyThread extends Thread {  2 usages
    @Override
    public void run() {
        try {
            sleep( millis: 500);
            System.out.println("....Work hard!");
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Результат

```
Main thread: Thread-0 started
....Work hard!
Main thread: finished
```

Создание потока через Thread

Код

```
public class ThreadExtension2 {
    public static void main(String[] args) {

        try {
            MyThread2 myThread = new MyThread2();
            myThread.start();

            for (int i = 0; i < 10; i++) {
                System.out.println(". " + i);
                Thread.sleep( millis: 100);
            }
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

    }
}

class MyThread2 extends Thread {  2 usages
    @Override
    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                System.out.println("... " + i);
                sleep( millis: 100);
            }
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Результат

```
... 0
. 0
. 1
... 1
. 2
... 2
. 3
... 3
. 4
... 4
... 5
. 5
... 6
. 6
. 7
... 7
. 8
... 8
. 9
... 9
```

Создание потока через Runnable

Код

```java
public class RunnableImplementation {
    public static void main(String[] args) {
        try {
            MyRunnable myRunnable = new MyRunnable();
            Thread thread = new Thread(myRunnable);
            thread.start();

            for (int i = 0; i < 10; i++) {
                System.out.println(". " + i);
                Thread.sleep( millis: 100);
            }
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}

class MyRunnable implements Runnable {  2 usages
    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                System.out.println("... " + i);
                sleep( millis: 100);
            }
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Результат

```
... 0
. 0
. 1
... 1
. 2
... 2
. 3
... 3
. 4
... 4
... 5
. 5
... 6
. 6
. 7
... 7
. 8
... 8
. 9
... 9
```

Несколько потоков. Реализация MyThread

Код

```
class MyThread implements Runnable {  15 usages  Nik *
    //Ссылка на объект потока
    public Thread thread;

    MyThread(String name) {  1 usage  Nik
        //инициализация потока
        thread = new Thread( task: this, name);
    }

    //Фабричный метод, который создает поток и сразу его запускает
    public static MyThread createAndStart(String name) {  6 usages  Nik
        MyThread myThread = new MyThread(name);
        myThread.thread.start();
        return myThread;
    }

    @Override  Nik
    public void run() {
        try {
            System.out.println("Thread " + thread.getName() + " started");
            for (int i = 0; i < 10; i++) {
                System.out.println("- Thread " + thread.getName() + " -> " + i);
                sleep( millis: 100);
            }
        } catch (InterruptedException ex) {
            System.out.println("Thread " + thread.getName() + " interrupted");
        }
    }
}
```

Thread(Runnable task, String name)

- конструктор создает поток на основе объекта task и присваивает ему имя name.

Несколько потоков. Реализация основной программы

Код

```java
public class ManyThreads {
    public static void main(String[] args) {
        System.out.println("MainThread started");

        MyThread myThread1 = MyThread.createAndStart( name: "Child 1");
        MyThread myThread2 = MyThread.createAndStart( name: "Child 2");
        MyThread myThread3 = MyThread.createAndStart( name: "Child 3");

        try {
            for (int i = 0; i < 10; i++) {
                System.out.println("MainThread -> " + i);
                sleep( millis: 100);
            }
        } catch (InterruptedException ex) {
            System.out.println("MainThread interrupted");
        }
        System.out.println("MainThread finished");
    }
}
```

```
MainThread started
MainThread -> 0
Thread Child 2 started
Thread Child 1 started
Thread Child 3 started
- Thread Child 1 -> 0
- Thread Child 3 -> 0
- Thread Child 2 -> 0
MainThread -> 1
- Thread Child 2 -> 1
- Thread Child 3 -> 1
- Thread Child 1 -> 1
MainThread -> 2
- Thread Child 3 -> 2
- Thread Child 2 -> 2
- Thread Child 1 -> 2
MainThread -> 3
- Thread Child 2 -> 3
- Thread Child 3 -> 3
- Thread Child 1 -> 3
MainThread -> 4
- Thread Child 1 -> 4
- Thread Child 3 -> 4
- Thread Child 2 -> 4
MainThread -> 5
- Thread Child 3 -> 5
- Thread Child 1 -> 5
- Thread Child 2 -> 5
MainThread -> 6
- Thread Child 2 -> 6
- Thread Child 3 -> 6
- Thread Child 1 -> 6
MainThread -> 7
- Thread Child 3 -> 7
- Thread Child 1 -> 7
- Thread Child 2 -> 7
MainThread -> 8
- Thread Child 2 -> 8
- Thread Child 3 -> 8
- Thread Child 1 -> 8
MainThread -> 9
- Thread Child 2 -> 9
- Thread Child 1 -> 9
- Thread Child 3 -> 9
MainThread finished
```

Несколько потоков. Объединение потоков

Код

```java
public class ManyThreadsJoin {  new *
    public static void main(String[] args) {  new *
        System.out.println("MainThread started");

        MyThread myThread1 = MyThread.createAndStart( name: "Child 1");
        MyThread myThread2 = MyThread.createAndStart( name: "Child 2");
        MyThread myThread3 = MyThread.createAndStart( name: "Child 3");

        try {
            for (int i = 0; i < 5; i++) {
                System.out.println("MainThread -> " + i);
                sleep( millis: 100);
            }
        } catch (InterruptedException ex) {
            System.out.println("MainThread interrupted");
        }

        try {
            myThread1.thread.join();//Ждать завершения указанного потока
            System.out.println(myThread1.thread.getName() + " joined");
            myThread2.thread.join();//Ждать завершения указанного потока
            System.out.println(myThread2.thread.getName() + " joined");
            myThread3.thread.join();//Ждать завершения указанного потока
            System.out.println(myThread3.thread.getName() + " joined");
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

        System.out.println("MainThread finished");
    }
}
```

void join() - заставляет текущий поток дождаться завершения другого потока, у которого вызван этот метод, прежде чем продолжить выполнение программы.

```
MainThread started
Thread Child 1 started
MainThread -> 0
Thread Child 3 started
Thread Child 2 started
- Thread Child 3 -> 0
- Thread Child 1 -> 0
- Thread Child 2 -> 0
- Thread Child 2 -> 1
- Thread Child 1 -> 1
- Thread Child 3 -> 1
MainThread -> 1
- Thread Child 2 -> 2
MainThread -> 2
- Thread Child 3 -> 2
- Thread Child 1 -> 2
- Thread Child 2 -> 3
MainThread -> 3
- Thread Child 3 -> 3
- Thread Child 1 -> 3
- Thread Child 2 -> 4
- Thread Child 1 -> 4
MainThread -> 4
- Thread Child 3 -> 4
- Thread Child 2 -> 5
- Thread Child 3 -> 5
- Thread Child 1 -> 5
- Thread Child 3 -> 6
- Thread Child 1 -> 6
- Thread Child 2 -> 6
- Thread Child 1 -> 7
- Thread Child 3 -> 7
- Thread Child 2 -> 7
- Thread Child 1 -> 8
- Thread Child 3 -> 8
- Thread Child 2 -> 8
- Thread Child 2 -> 9
- Thread Child 1 -> 9
- Thread Child 3 -> 9
Child 1 joined
Child 2 joined
Child 3 joined
MainThread finished
```

Несколько потоков. Приоритет

Каждый поток имеет приоритет, который влияет на то, как JVM распределяет время процессора между потоками. Однако приоритет не гарантирует, что высокоприоритетные потоки будут всегда выполняться раньше низкоприоритетных.

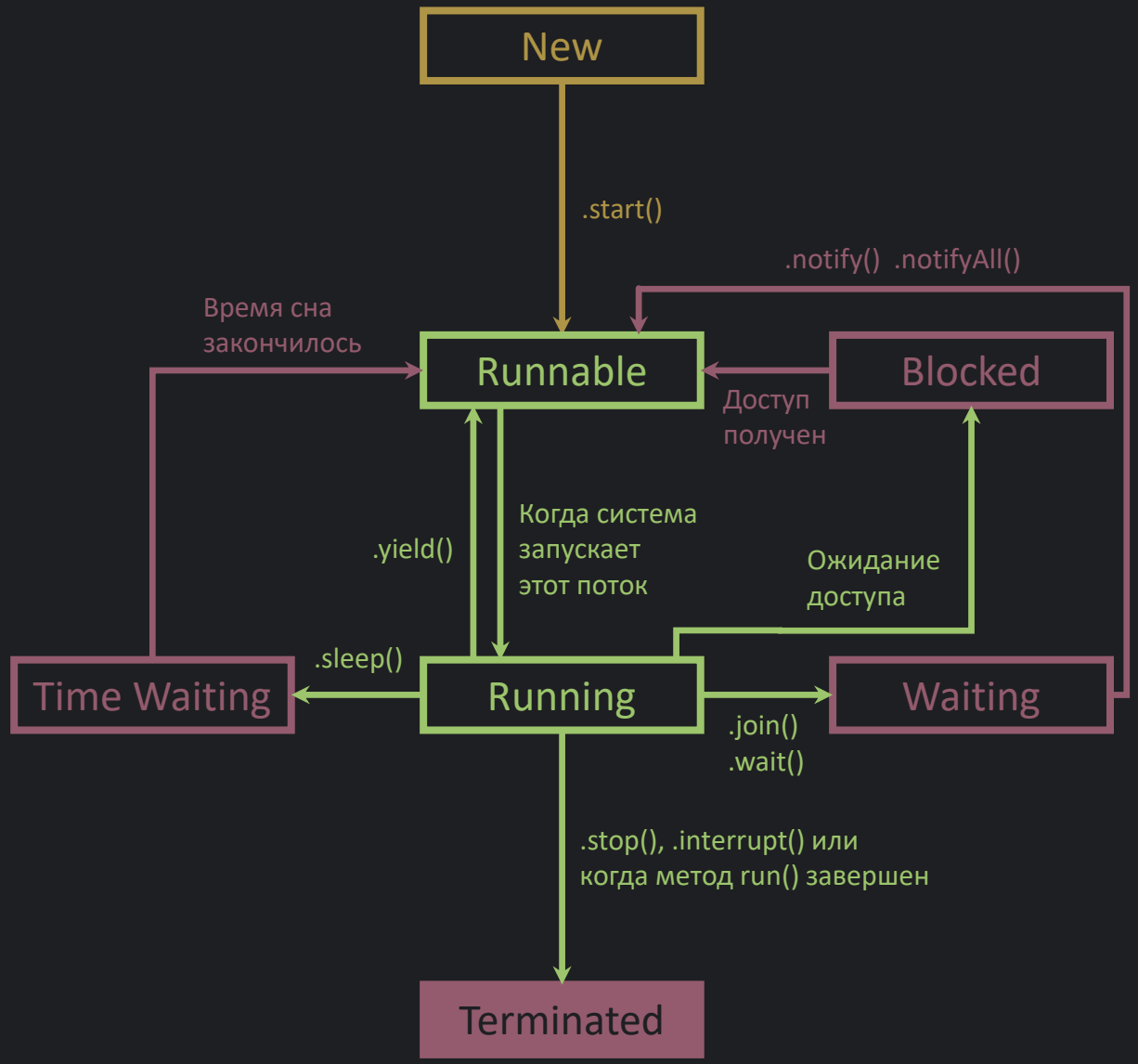
Приоритеты потоков задаются числами от 1 до 10:
MIN_PRIORITY (1): Минимальный приоритет.
NORM_PRIORITY (5): Нормальный приоритет (по умолчанию).
MAX_PRIORITY (10): Максимальный приоритет.

void setPriority(int newPriority)
- устанавливает приоритет потока (только до запуска потока

```
public static void main(String[] args) {  new *
    Thread lowPriorityThread = new Thread(() -> {
        System.out.println("Низкоприоритетный поток начал работу.");
    });

    Thread highPriorityThread = new Thread(() -> {
        System.out.println("Высокоприоритетный поток начал работу.");
    });

    lowPriorityThread.setPriority(Thread.MIN_PRIORITY); // Приоритет 1
    highPriorityThread.setPriority(Thread.MAX_PRIORITY); // Приоритет 10

    lowPriorityThread.start();
    highPriorityThread.start();
}
```

```
Высокоприоритетный поток начал работу.
Низкоприоритетный поток начал работу.
```

Этапы жизни потока



NEW - Поток создан, но ещё не запущен.
RUNNABLE - В этом состоянии поток считается готовым к выполнению и может быть выбран планировщиком потоков для выполнения на процессоре. Поток может фактически выполняться или ожидать своей очереди на выполнение.
BLOCKED - Поток переходит в это состояние, если он пытается войти в синхронизированный блок или метод, доступ к которому в данный момент удерживается другим потоком.
WAITING - Поток находится в состоянии ожидания без указания времени, пока другой поток не разбудит его.
TIMED_WAITING - Похожее на состояние WAITING, но с указанным временем ожидания. Поток переходит в это состояние при вызове методов sleep() или join() с указанием таймаута.
TERMINATED - Поток переходит в это состояние, когда метод run() завершает своё выполнение либо из-за нормального завершения, либо из-за неперехваченного исключения.

Этапы жизни потока

State getState() - возвращает текущий статус потока

boolean isAlive() - возвращает true, если поток уже запущен (start) и еще не прерван (terminate)

```java
class MyThread extends Thread {  2 usages  new *
    private boolean timeToSleep = false;  2 usages

    public void setTimeToSleep(boolean timeToSleep) {  1 usage  new *
        this.timeToSleep = timeToSleep;
    }

    @Override  new *
    public void run() {
        System.out.println("Мы находимся в методе 'run'.");
        while (!timeToSleep) {
            ;
        }
        try {
            sleep( millis: 500);// Переход потока в TIMED_WAITING
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

```java
public class ThreadLifeCycle {  new *
    public static void main(String[] args) {  new *
        try {
            MyThread thread = new MyThread();

            System.out.println(
                    "Состояние потока после создания: " +
                            thread.getState());  // NEW
            System.out.println("Живой? " + thread.isAlive());
            thread.start();
            System.out.println(
                    "Состояние потока после вызова start(): " +
                            thread.getState());  // RUNNABLE
            System.out.println("Живой? " + thread.isAlive());

            thread.setTimeToSleep(true);
            //Остановим основной поток, чтобы дочерний успел заснуть
            Thread.sleep( millis: 300);
            System.out.println(
                    "Состояние потока после вызова sleep(): " +
                            thread.getState());  // TIMED_WAITING
            System.out.println("Живой? " + thread.isAlive());

            //Остановим основной поток до завершения дочернего
            Thread.sleep( millis: 600);

            System.out.println(
                    "Состояние потока после завершения: " +
                            thread.getState());  // TERMINATED
            System.out.println("Живой? " + thread.isAlive());
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Этапы жизни потока

```java
public class ThreadLifeCycle {  new *
    public static void main(String[] args) {  new *
        try {
            MyThread thread = new MyThread();

            System.out.println(
                    "Состояние потока после создания: " +
                            thread.getState());  // NEW
            System.out.println("Живой? " + thread.isAlive());
            thread.start();
            System.out.println(
                    "Состояние потока после вызова start(): " +
                            thread.getState());  // RUNNABLE
            System.out.println("Живой? " + thread.isAlive());

            thread.setTimeToSleep(true);
            //Остановим основной поток, чтобы дочерний успел заснуть
            Thread.sleep( millis: 300);
            System.out.println(
                    "Состояние потока после вызова sleep(): " +
                            thread.getState());  // TIMED_WAITING
            System.out.println("Живой? " + thread.isAlive());

            //Остановим основной поток до завершения дочернего
            Thread.sleep( millis: 600);

            System.out.println(
                    "Состояние потока после завершения: " +
                            thread.getState());  // TERMINATED
            System.out.println("Живой? " + thread.isAlive());
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

```
Состояние потока после создания: NEW
Живой? false
Состояние потока после вызова start(): RUNNABLE
Мы находимся в методе 'run'.
Живой? true
Состояние потока после вызова sleep(): TIMED_WAITING
Живой? true
Состояние потока после завершения: TERMINATED
Живой? false
```

Гонка состояний

Гонка состояний (Race Contidion) - явление, когда потоки делят между собой некоторый ресурс и код написан таким образом, что не предусматривает корректную работу в таком случае.

```
private static int value = 0;  2 usages

public static void main(String[] args) {  new *
    Runnable task = () -> {
        for (int i = 0; i < 10000; i++) {
            int oldValue = value;
            int newValue = ++value;
            if (oldValue + 1 != newValue) {
                throw new IllegalStateException(
                        oldValue + " + 1 = " + newValue);
            }
        }
    };
    new Thread(task).start();
    new Thread(task).start();
    new Thread(task).start();
}
```

```
Exception in thread "Thread-1" Exception in thread "Thread-2" java.lang.IllegalStateException: 3964 + 1 = 3968
    at org.example.Problems.RaceCondition.lambda$main$0(RaceCondition.java:13) <1 internal line>
java.lang.IllegalStateException Create breakpoint : 3864 + 1 = 3918
    at org.example.Problems.RaceCondition.lambda$main$0(RaceCondition.java:13) <1 internal line>
```

Синхронизация

Из чего формируется синхронизация:

Монитор (intrinsic lock, оно же monitor lock) – механизм, который существует у каждого объекта Java и который управляет блокировкой этого объекта.

Оператор synchronized – задает границы синхронизированного кода (критическая секция). Данный оператор использует объект блокировки – по сути любой объект Java.

Блок синхронизации (критическая секция) – область кода, доступ к которой выполняется через объект блокировки и которая доступна только тому процессу, который захватил объект блокировки.

```
Object lock = new Object();

Runnable task = () -> {
    synchronized (lock) {
        for (int i = 0; i < 10; i++) {
            try {
                System.out.println(
                        Thread.currentThread().getName() + "-" + i);
                sleep( millis: 100);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
};
```

Варианты синхронизации

1. Синхронизация блока кода. Доступ к коду происходит через объект блокировки

```
Runnable task = () -> {
    synchronized (lock) {
        System.out.println("Hello World");
    }
};
```

2. Синхронизация метода. Доступ к методу происходит через блокировку инстанции, которой метод принадлежит.

```
class Task implements Runnable {  1 usage   Nik *
    @Override  new *
    public void run() {
        synchMethod();
    }
    synchronized void synchMethod() {  1 usage  new *
        System.out.println("Hello World");
    }
}
```

```
public static void main(String[] args) throws InterruptedException {

    Runnable task = new Task();
    Thread thread0 = new Thread(task);
    Thread thread1 = new Thread(task);
    thread0.start();
    thread1.start();
}
```

3. Синхронизация статического метода. Доступ к методу происходит через блокировку объекта класса. Например, Integer.class.

Wait() и Notify()

Оператор wait()
- может быть вызван только внутри синхронизированной секции
- останавливает выполнение текущего потока и ставит его в очередь ожидания (WAIT-SET) монитора объекта, по которому выполняется синхронизация.

Оператор notify()
- может быть вызван только внутри синхронизированной секции
- оповещает поток, находящийся в очереди ожидания (WAIT-SET) монитора объекта, по которому выполняется синхронизация, о возможности продолжить работу.

Оператор notifyAll() – оповещает все потоки в очереди ожидания монитора.

notify()
- **Пробуждает только один случайный поток** из очереди ожидающих (какой именно — зависит от JVM, порядок не гарантируется).
- **Эффективнее**, если нужно разбудить строго один поток (например, в пуле рабочих потоков).
- **Риск "зависания"**, если выбранный поток не сможет выполнить условие и не разбудит следующий.

notifyAll()
- **Пробуждает все потоки**, ожидающие на этом мониторе.
- **Потоки переконкурируют** за доступ к монитору (выигравший продолжит работу, остальные снова ждут).
- **Надежнее**, когда условие пробуждения сложное или его могут обработать несколько потоков.
- **Менее эффективен**, так как будит лишние потоки.

Wait() и Notify()

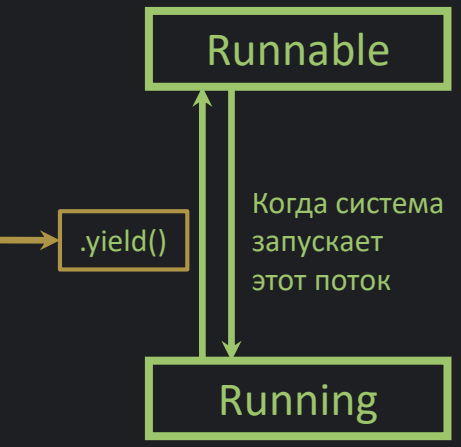
```java
public static void main(String[] args) {  NikOnNote
    Object lock = new Object();
    Runnable task = ()->{
        synchronized (lock) {
            try{
                System.out.println("ChildThread get lock");
                lock.wait();
                System.out.println("ChildTread finished waiting");
            }catch (InterruptedException e){
                System.out.println("Thread interrupted");
            }
        }
    };

    Thread thread = new Thread(task);
    thread.start();
    try{
        Thread.sleep( millis: 200);
    }catch (InterruptedException e){
        System.out.println("Thread interrupted");
    }

    synchronized (lock) {
        System.out.println("MainThread get lock");
        lock.notify();
        System.out.println("MainThread notify child");
    }
}
```

```
ChildThread get lock
MainThread get lock
MainThread notify child
ChildTread finished waiting
```

Yield() – загадочный метод

Если верить схеме жизненного цикла потока – метод yield() способен остановить поток.

… но это не так



На самом деле метод yield() лишь передаёт некоторую рекомендацию планировщику потоков Java, что данному потоку можно дать меньше времени исполнения. Но что будет на самом деле, услышит ли планировщик рекомендацию и что вообще он будет делать — зависит от реализации JVM и операционной системы. А может и ещё от каких-то других факторов.

DeadLock

Deadlock (взаимная блокировка) возникает, когда два или более потоков бесконечно ждут друг друга, удерживая ресурсы, которые нужны другому.

```
public class DeadLock {  new *
    public static void main(String[] args) {  new *
        final Object lock1 = new Object();
        final Object lock2 = new Object();

        // Поток 1: захватывает lock1, затем пытается взять lock2
        Thread thread1 = new Thread(() -> {
            synchronized (lock1) {
                System.out.println("Поток 1: удерживает lock1");
                try {
                    Thread.sleep( millis: 100); // Имитация работы
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Поток 1: ждёт lock2...");
                synchronized (lock2) {
                    System.out.println("Поток 1: удерживает lock1 и lock2");
                }
            }
        });
        // Поток 2: захватывает lock2, затем пытается взять lock1
        Thread thread2 = new Thread(() -> {
            synchronized (lock2) {
                System.out.println("Поток 2: удерживает lock2");
                try {
                    Thread.sleep( millis: 100);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Поток 2: ждёт lock1...");
                synchronized (lock1) {
                    System.out.println("Поток 2: удерживает lock2 и lock1");
                }
            }
        });

        thread1.start();
        thread2.start();
    }
}
```

```
Поток 1: удерживает lock1
Поток 2: удерживает lock2
Поток 2: ждёт lock1...
Поток 1: ждёт lock2...
```

LiveLock

Livelock — это ситуация, когда потоки не блокируются, но и не могут продвинуться дальше, постоянно "вежливо" уступая друг другу ресурсы. В отличие от deadlock, потоки активны, но их работа бесполезна.

Как избежать Livelock?

- Разные стратегии ожидания: добавьте случайные задержки (Thread.sleep(randomTime)), чтобы потоки не синхронизировали свои действия.
- Ограничение числа попыток.
- Приоритизация потоков.

Starvation

Starvation возникает, когда один или несколько потоков не могут получить доступ к общим ресурсам, потому что другие потоки (обычно с более высоким приоритетом) постоянно их монополизируют.

Как избежать Starvation?

- Использовать fair (честные) блокировки (это из java.util.concurrent).
- Уменьшить время удержания блокировки: минимизировать код внутри synchronized
- Добавить принудительные паузы
- Балансировать приоритеты потоков

Volatile

```
public static boolean flag = false;  2 usages

public static void main(String[] args) throws InterruptedException {
    Runnable whileFlagFalse = () -> {
        while(!flag) {
        }
        System.out.println("Flag is now TRUE");
    };

    new Thread(whileFlagFalse).start();
    Thread.sleep( millis: 1000);
    flag = true;
}
```

Вопрос к знатокам:

Почему код слева будет работать бесконечно?

Демоны

Поток-демон - это фоновый поток, который автоматически завершается, когда все обычные (не-демоны) потоки завершили свою работу.

Метод setDaemon(true) – помечает поток, как демона. Использование этого метода возможно только ДО запуска потока.

Когда использовать демоны?

- Фоновые сервисы. Например, сборка мусора (GC), автосохранение, мониторинг.
- Вспомогательные задачи: очистка временных файлов, отправка логов.
- Потоки, которые можно безопасно прервать

Ограничения демонов.

- Поток-демон может завершиться на любой операции, поэтому он не гарантирует исполнение заложенных алгоритмов. Например, запись в БД или сохранение файла может прерваться.
- Не имеют гарантии выполнения finally.

Задание

1. Реализовать паттерн «Producer-consumer».

Общий ресурс – массив элементов.
Producer – отдельный поток, который добавляет элементы в массив.
Consumer – отдельный поток, который считывает элементы из массива и выводит в консоль.

Нельзя допускать переполнение массива или чтение элементов при пустом массиве.

2. Внедрить логирование в разработанную программу.