МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ В JAVA

- Пакеты
  - java.lang
  - java.util.concurrent

ВВЕДЕНИЕ

ЧАСТЬ 1

ПОТОКИ

- Каждый поток имеет свой стек вызовов
- У потоков общая память!
- Java-поток != поток в ОС (в стандартной реализации они сопоставлены один к одному, но в общем случае – это не обязательно)
- **У JVM свой планировщик потоков, который не зависит от планировщика операционной системы под которой работает JVM**
- В Java есть два вида потоков: потоки-демоны (daemon threads) и пользовательские потоки (user threads). JVM завершает выполнение программы когда все пользовательские потоки завершат свое выполнение.

СОЗДАНИЕ ПОТОКОВ

- Класс Thread – поток
  - Позволяет создавать потоки и производить операции с ними
- Интерфейс Runnable – сущность, которая может быть запущена
  - public void run();

СОЗДАНИЕ ПОТОКА (RUNNABLE)

- Пример кода

```
// Создание потока
Thread t = new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello");
    }
});
// Запуск потока
t.start();
```

СОЗДАНИЕ ПОТОКА (THREAD)

- Не рекомендуется использовать
- Пример кода

```
// Создание потока
Thread t = new Thread() {
    public void run() {
        System.out.println("Hello");
    }
};
// Запуск потока
t.start();
```

ИНСТАНЦИРОВАНИЕ ПОТОКА

- Если вы расширили класс Thread:
  ```
  MyThread t = new MyThread();
  ```
- Если вы реализовывали Runnable:
  ```
  MyRunnable r = new MyRunnable();
  Thread t = new Thread(r);
  ```
- Один экземпляр Runnable можно передать нескольким объектам Thread:
  ```
  MyRunnable r = new MyRunnable();
  Thread foo = new Thread(r);
  Thread bar = new Thread(r);
  Thread bat = new Thread(r);
  ```

КОНСТРУКТОР THREAD

- Thread()
- Thread(String name)
- Thread(Runnable runnable)
- Thread(Runnable runnable, String name)
- …

ЗАПУСК ПОТОКА

- Нужно так:
  t.start();
- НЕ ТАК:
  t.run();

ПРИМЕР

```
public class Starter {
    public static void main(String[] args) {
        NameRunnable nr = new NameRunnable();
        Thread one = new Thread(nr);
        Thread two = new Thread(nr);
        Thread three = new Thread(nr);

        one.setName("Первый");
        two.setName("Второй");
        three.setName("Третий");

        one.start();
        two.start();
        three.start();
    }
}
class NameRunnable implements Runnable {
    public void run() {
        for (int x = 1; x <= 3; x++) {
            System.out.println("Запущен " + Thread.currentThread().getName() + ", x равен " + x);
        }
    }
}
```

СВОЙСТВА ПОТОКА

- Основные свойства
  - id – идентификатор потока
  - name – имя потока
  - priority – приоритет
  - daemon – поток-демон
- Свойства потока не могут изменяться после запуска

СОСТОЯНИЯ ПОТОКА

- Состояние потока возвращается методами
  - int getState()
  - boolean isAlive()

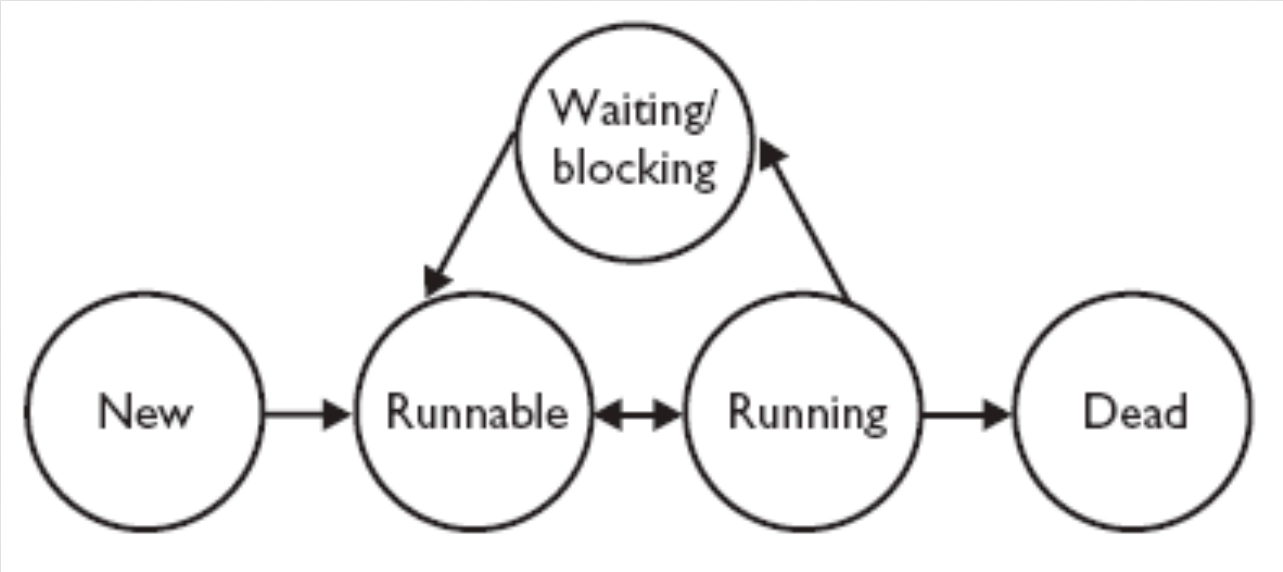
getState()	isAlive()
NEW	
RUNNABLE	+
BLOCKED	+
WAITING	+
TIMED_WAITING	+
TERMINATED	

СОСТОЯНИЯ ПОТОКА

Поток может находиться в одном из пяти состояний:

- **Новый** (new). После создания экземпляра потока, он находится в состоянии Новый до тех пор, пока не вызван метод start(). В этом состоянии поток не считается живым.
- **Работоспособный** (runnable). Поток переходит в состояние Работоспособный, когда вызывается метод start(). Поток может перейти в это состояние также из состояния Работающий или из состояния Блокирован.
- **Работающий** (running). Поток переходит из состояния Работоспособный в состояние Работающий, когда Планировщик потоков выбирает его из runnable pool как работающий в данный момент.
- **Ожидающий** (waiting)/**Заблокированный** (blocked)/**Спящий** (sleeping). Эти состояния характеризуют поток как не готовый к работе.
- **Мёртвый (**dead**)**.

СОСТОЯНИЯ ПОТОКА



ВЗАИМОДЕЙСТВИЕ ПОТОКОВ

- Создание потока
- Запуск потока (start)
- Ожидание окончания потока (join)
- Прерывание потока (interrupt)

- Засыпание потока (sleep)
- Переключение потоков (yield)

ОТПРАВКА В СОН

- Thread.sleep(millis, nanos)
- TimeUnit.SECONDS.sleep(secs)
- Перечисление TimeUnit
  - SECONDS
  - MILLISECONDS
  - MICROSECONDS
  - NANOSECONDS
  - DAYS
  - HOURS
  - MINUTES
- Все методы кидают InterruptedExcepton

ПРИОРИТЕТЫ И YIELD

- Приоритеты:
  Thread.MIN_PRIORITY (1)
  Thread.NORM_PRIORITY (5)
  Thread.MAX_PRIORITY (10)

- ```
  FooRunnable r = new FooRunnable();
  Thread t = new Thread(r);
  t.setPriority(8);
  t.start();
  ```

- Метод yield() можно использовать для того чтобы предложить планировщику выполнить другой поток, который ожидает своей очереди.
  ```

БЛОКИРОВКА ДРУГОГО ПОТОКА

- Методы sleep и yield – статические! И действуют на текущий поток!
- Приостановить работу другого потока нельзя.

ОЖИДАНИЕ ОКОНЧАНИЯ ПОТОКА

- Методы класса Thread
  - join() –**текущий поток** ожидает до завершения
  - join(long millis) – ожидать до завершения или истечения millis миллисекунд
  - join(long millis, long nanos) – ожидать до завершения или истечения millis миллисекунд и nanos наносекунд
- Все методы ожидания кидают InterruptedExcepton

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

ПРЕРЫВАНИЕ ПОТОКА

- Методы класса Thread
  - interrupt() – установить флаг прерывания
  - isInterrupted() – проверить флаг прерывания
  - interrupted() – проверить и сбросить флаг прерывания
- Методы, которые ожидают в процессе выполнения должны бросать InterruptedException

ОБРАБОТКА ДАННЫХ В ЦИКЛЕ

```
class Worker implements Runnable {
    public void run() {
        try {
            while (!Thread.interrupted()) {
                // Полезные действия
            }
        } catch (InterruptedException e) {
        }
        // Исполнение потока прервано
        // Поток заканчивает работу
    }
}
```

СИНХРОНИЗАЦИЯ КОДА

ЧАСТЬ - 2

БЛОКИРОВКИ В JAVA

- Любой объект может служить блокировкой
- Снятие блокировки производится автоматически
- Синтаксис

```
synchronized (o) { // Получение блокировки
 …
} // Снятие блокировки
```

МЕТОДЫ ЭКЗЕМПЛЯРА

- Метод экземпляра может быть объявлен синхронизованным

```
public synchronized int getValue() { … }
```

- Эквивалентно (почти)

```
public int getValue() {
    synchronized (this) { … }
}
```

МЕТОДЫ КЛАССА

- Метод класса может быть объявлен синхронизованным

```
Class Example {
    public static synchronized int getValue() { … }
```

- Эквивалентно

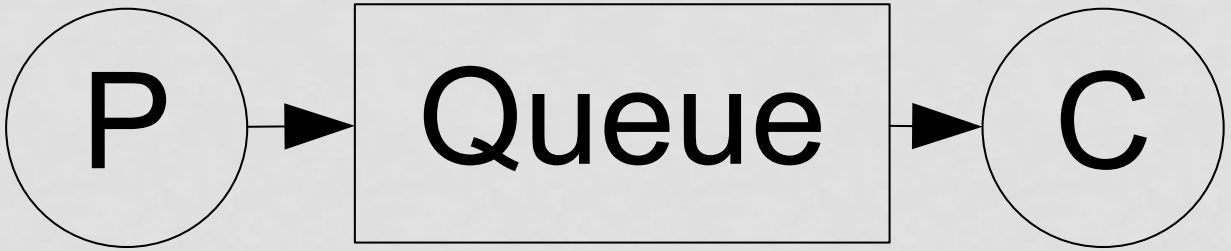
```
public int getValue() {
    synchronized (Example.class) { … }
}
```

УДЕРЖИВАНИЕ БЛОКИРОВОК

- При вызове yield, sleep, join поток удерживает все свои блокировки!

ПРИМЕР ПРОИЗВОДИТЕЛЬ-ПОТРЕБИТЕЛЬ

- Один поток производит данные, второй их потребляет
- Несколько потоков производят данные и несколько их потребляют
- Данные могут храниться в очереди (не)ограниченного объема



ИНТЕРФЕЙС ОЧЕРЕДИ

- Хранит один элемент

```
class Queue {
    private Object data;
    public void set(Object data) { … }
    public Object get() { … }
}
```

ПРОИЗВОДИТЕЛЬ

- Установка значения

```
public void set(Object data) {
        while (true) { // Активное ожидание
            synchronized (this) {
                if (this.data == null) {
                    this.data = data;
                    break;
                }
            }
        }
    }
```

ПОТРЕБИТЕЛЬ

- Получение значения

```
public Object get() {
        while (true) { // Активное ожидание
            synchronized (this) {
                if (data != null) {
                    Object d = data;  data = null;
                    return d;
                }
            }
        }
    }
```

МОНИТОРЫ И УСЛОВИЯ

ЧАСТЬ 3

ВЗАИМОДЕЙСТВИЕ ПОТОКОВ

- В классе Object есть три метода
  wait(),
  notify(),
  notifyAll(),
  которые позволяют потоку сообщать информацию о своем состоянии другим, заинтересованным в этой информации, потокам.

МОНИТОРЫ

- Любой объект может быть монитором
- Передача событий
  - wait(time?) – ожидание условия
  - notify() – извещение одного из ждущих потоков
  - notifyAll() – извещение всех ждущих потоков
- Нужно владеть блокировкой
  - IllegalMonitorStateException

МОНИТОРЫ

- При ожидании монитора (wait) блокировка с него снимается
- При извещении поток не получает управления пока не может получить блокировку обратно
- notify() и notifyAll() не снимают блокировку!

ПРОИЗВОДИТЕЛЬ (1')

- Установка значения

```
public synchronized void set(Object data)
    throws InterruptedException
{
    if (this.data != null) {
        wait(); // Пассивное ожидание
    }
    this.data = data;
    notify();
}
```

ПОТРЕБИТЕЛЬ (1')

- Получение значения

```
public synchronized Object get()
    throws InterruptedException {
    if (data == null) {
        wait(); // Пассивное ожидание
    }
    Object d = data;
    data = null;
    notify();
    return d;
}
```

ПРОИЗВОДИТЕЛЬ (2)

- Установка значения

```
public synchronized void set(Object data)
    throws InterruptedException
{
    while (this.data != null) {
        wait(); // Пассивное ожидание
    }
    this.data = data;
    notify();
}
```

ПОТРЕБИТЕЛЬ (2)

- Получение значения

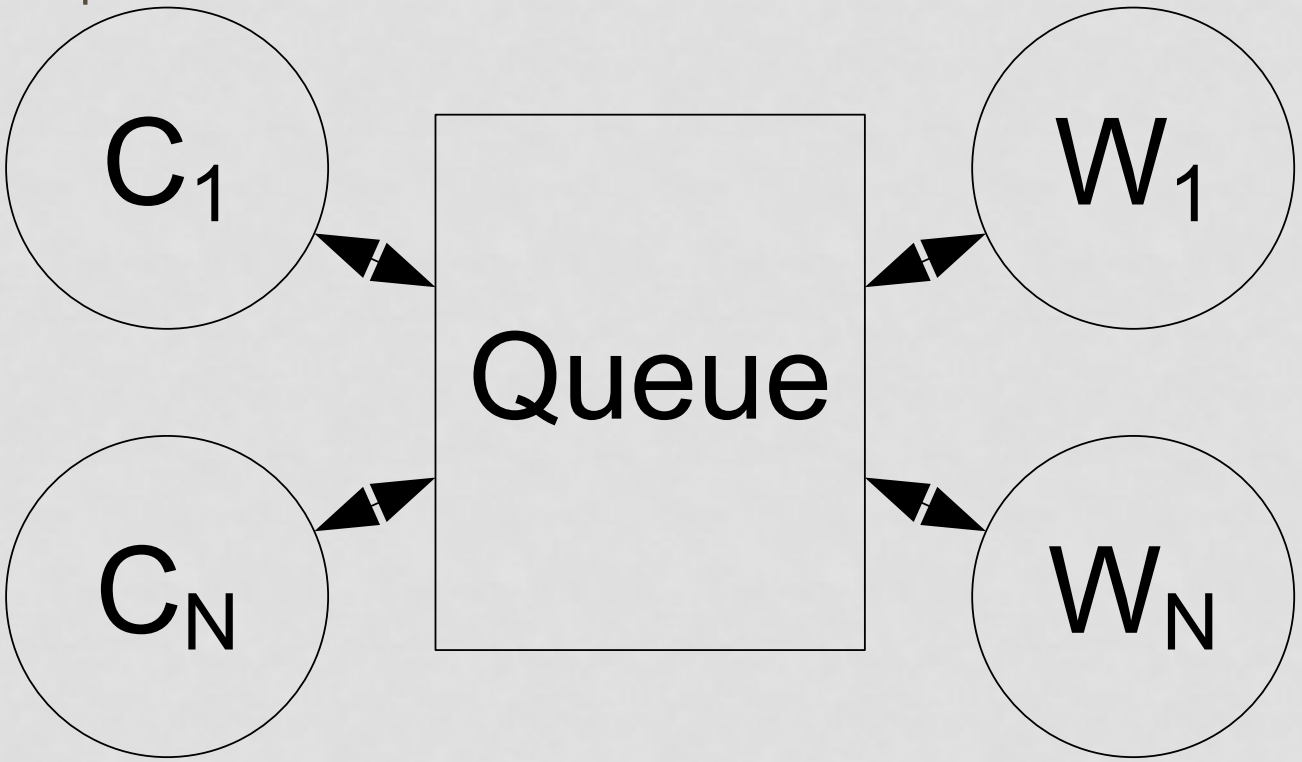
```
public synchronized Object get()
    throws InterruptedException {
    while (data == null) {
        wait(); // Пассивное ожидание
    }
    Object d = data;
    data = null;
    notify();
    return d;
}
```

ВНЕЗАПНЫЕ ПРОБУЖДЕНИЯ

- wait() может завершиться без notify()
  - Проверить наступление события
  - Ожидать всегда в цикле
- Идиома
  - while (*дождался*) wait();

ЗАДАНИЯ-РАБОТНИКИ

- Поток-клиент ждет выполнения задания потоком-сервером



ЗАДАНИЯ-РАБОТНИКИ

- Решение с помощью монитора
  - Задание

```
queue.add(task);
queue.notify();
task.wait();
```

  - Работник

```
while (queue.isEmpty()) queue.wait();
Task t = queue.get();
// Обработка задания
t.notify();
```

NOTIFYALL()

```
public class Reader extends Thread {
    Calculator c;
    public Reader(Calculator calc) { c = calc;  }

    public void run() {
        synchronized (c) {
            try {
                System.out.println("Вычисление...");
                c.wait();
            } catch (InterruptedException e) {           }
            System.out.println("Total равно: " + c.total);
        }
    }
    public static void main(String[] args) {
        Calculator calculator = new Calculator();
        new Reader(calculator).start();  new Reader(calculator).start(); new Reader(calculator).start();
        calculator.start();
    }
}
class Calculator extends Thread {
    int total;
    public void run() {
        synchronized (this) {
            for (int i = 0; i < 100; i++) { total += i; }
            notifyAll();
        }
    }
}
```

JAVA MEMORY MODEL

...FOR DUMMIES

ОСНОВНЫЕ СВОЙСТВА

- Атомарность
- Видимость
- Упорядоченность

АТОМАРНОСТЬ

- Атомарная операция выполняется как единое целое
- Операции над всеми типами кроме long и double являются атомарными (это почти правда ☺)

ПРИМЕР

```
int a = 0;
long b = 0;

// T1
a = 1;
b = -1;
```

- Возможные значения a
  - 0
  - 1
- Возможные значения b
  - 0
  - -1
  - 0xffffffff00000000
  - 0x00000000ffffffff
  - …

ВИДИМОСТЬ

- Изменения произведенные потоком 1 видимы потоком 2
- Видимость гарантируется в следующих случаях
  - После изменений поток 1 освободил блокировку, которую захватил поток 2
  - После изменения поток 1 создал поток 2
  - Поток 2 дождался окончания потока 1
- При неправильной синхронизации изменения могут быть видимы в произвольном порядке

ПРИМЕР

```
int a = 0;
int b = 0;

// T1
a = 1;
b = 2;
```

- Возможные значения пары a, b
  - 0, 0
  - 1, 0
  - 1, 2
  - 0, 2

УПОРЯДОЧЕННОСТЬ

- Программы выполняются как если бы они были написаны последовательно
- С точки зрения других потоков выполнение программы может производиться в произвольном порядке

ПРИМЕР

```
int a = 0;
a = 1;
a = 2;
```

- Возможные последовательности значений a
  - 0, 0
  - 0, 1
  - 0, 2
  - 1, 2
  - 2, 0
  - 2, 1
  - …

VOLATILE-ПЕРЕМЕННЫЕ

- Операции с volatile-переменными всегда атомарны
- При чтение значения volatile-переменной оно всегда читается из общей памяти
- При записи значения volatile-переменной оно всегда записывается в общую память
- Если volatile-ссылка изменилась, то данные доступные по ней могли не измениться

ПРО SINGLETON

```
// Однопоточная версия
class Foo {
    private static Helper helper = null;
    public static Helper getHelper() {
        if (helper == null)
            helper = new Helper();
        return helper;
    }

    // и остальные члены класса…
}
```

ПРО SINGLETON

```
// Правильная, но "дорогая" по времени выполнения многопоточная версия
class Foo {
    private static Helper helper = null;
    public synchronized Helper getHelper() {
        if (helper == null)
            helper = new Helper();
        return helper;
    }

    // и остальные члены класса…
}
```

ПРО SINGLETON

```
// Неработающая многопоточная версия
// Шаблон "Double-Checked Locking"
class Foo {
    private static Helper helper = null;
    public static Helper getHelper() {
        if (helper == null) {
            synchronized(Foo.class) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }
    // и остальные члены класса…
}
```

ПРО SINGLETON

```
// Работает с новой семантикой volatile
// Не работает в Java 1.4 и более ранних версиях из-за семантики volatile
class Foo {
    private volatile Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null)
                    helper = new Helper();
            }
        }
        return helper;
    }

    // и остальные члены класса…
}
```

ПРО SINGLETON

```
public class Singleton {
   private Singleton() {}

   private static class SingletonHolder {
      public static final Singleton instance = new Singleton();
   }

   public static Singleton getInstance()  {
      return SingletonHolder.instance;
   }
}
```

ПРО SINGLETON

```
public class Singleton {
    private static final Singleton instance =
                    new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return instance;
    }
}
```

ПРО SINGLETON

```
public class Singleton {
  private static final Singleton instance;

  static {
    try {
      instance = new Singleton();
    } catch (IOException e) {
      throw new RuntimeException("Darn, an error occurred!", e);
    }
  }

  public static Singleton getInstance() {
    return instance;
  }

  private Singleton() {
    // ...
  }
}
```