# Потоки

...

## многопоточное программирование

**дополнительное чтение:** http://sites.znu.edu.ua/javaprog/lect/1133.ukr.html

# Поток -
# единая последовательность выполнения команд

# Простейший поток

## поток основной программы

```
public class HelloWorld {

public static void main(String args[])
{System.out.println("Hello, world!"); }

}
```

---

поток запускается с main, выполняет строку кода и умирает

# Два простейших потока

file SayHelloThreads.java

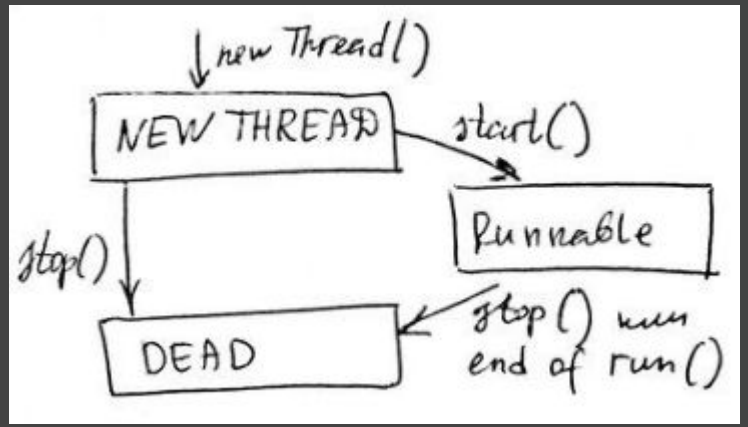
```
public class SayHelloThreads {
public static void main(String args[]) {
        new Greeter1().start(); //метод start() охотится на метод run()
        new Greeter2().start();
}
class Greeter1 extends Thread { public void run() {
                        System.out.println("Greetings from thread 1");}
class Greeter2 extends Thread { public void run() {
                        System.out.println("Greetings from thread 2");}
        }
}
```

# Жизнь и периоды жизни потока

1) Созданный, но еще не запущенный. Состояние **NEW THREAD**

2) Выполняемый. Состояние **RUNNABLE**.
   Метод, приводящий к нему - start()

3) Пассивный. Состояние **DEAD**. Метод, приводящий к нему - stop().
   Естественный переход - окончание работы метода run()



# Шаблон работы потока в ожидании stop()

file aThreadWaitsForStop.java

```
class aThreadWaitsForStop {
public static void main ()
{
        Thread aThread = new Thread();
        aThread.start();
...
        while(true) {...
        boolean stopAThread = FALSE;
        ...//значение меняется на TRUE в коде
        if (stopAThread) { aThread.stop(); }
        }}
}
```

Как только вызывается метод stop(), aThread вскоре “умирает”, причем “умирает” асинхронно, т.е. первый поток (поток main) не дожидается смерти потока aThread

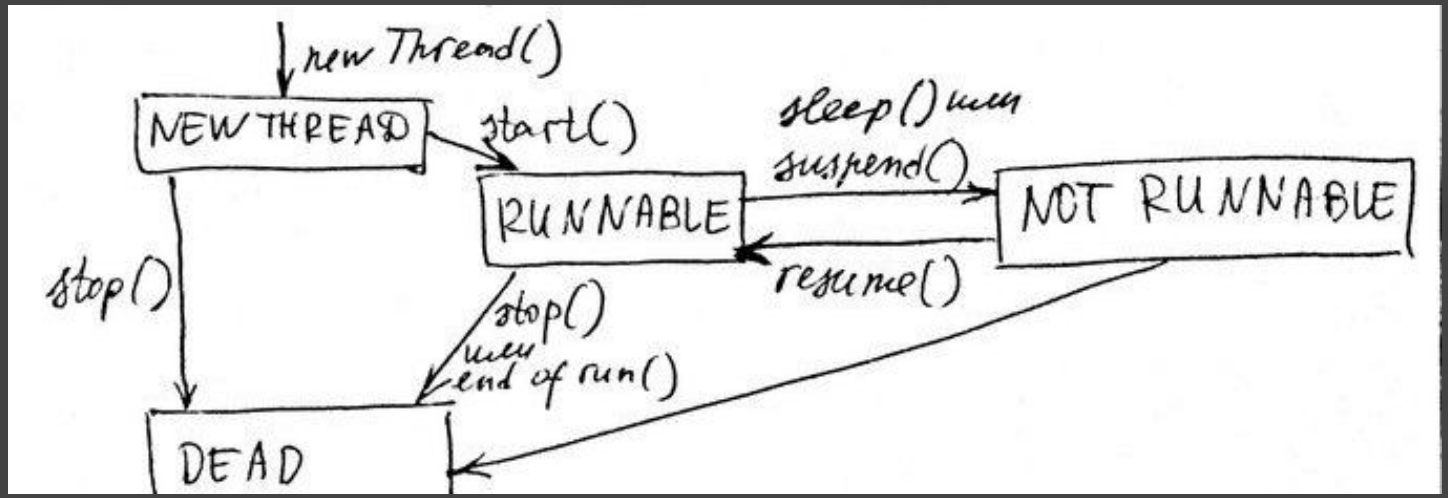
# Дополнительные периоды жизни потока

4) Приостановленный. Состояние **SUSPENDED**

Методы, приводящие к такому состоянию - suspend() и sleep()

Метод, возвращающий в состояние **RUNNABLE** - resume()

Метод, проверяющий, “жив ли” поток (**RUNNABLE**) - isAlive()



# isAlive()

Метод не различает, запущен ли поток или он мертвый или новый, а только показывает, пассивен ли поток или нет

( = новый незапущенный либо мертвый )

```
if (aThread.isAlive()) {
        aThread.suspend() ;
}
else {
        ...// действия, как если бы поток умер
}
```

# sleep(x)

Метод погружает поток в состояние SUSPENDED на x миллисекунд, посчле чего поток восстанавливает статус RUNNABLE

```
while (true) {...
        sleep(2000) ;
        ...
}
```

# Способы создания потока

## 1й вариант

Создание потока путем расширения класса Thread

```
public class DerivedThread extends Thread {

public void run() {...}

}
```

# Способы создания потока

## 2й вариант

Создание потока путем реализации интерфейса Runnable:

Шаг 1. Декларация класса. До тех пор, пока в теле этого класса не появится тело потока - он не будет являться Thread

```
public class NewRunnable implements
Runnable {
public void run() {...}
}
```

# Способы создания потока

## 2й вариант

Создание потока путем реализации интерфейса Runnable:

Шаг 2. Создаем экземпляр Thread и передаем его в NewRunnable (например, в методе main())

```
public static void main () {
...
NewRunnable runnable = new
                                NewRunnable();
        new Thread(runnable). start();
...
}
```

Принципиальное отличие варианта 1 от варианта 2 - невозможно узнать имя потока для варианта 2. Runnable-объект даже не знает, что он - часть потока.

В варианте 1 имя потока можно узнать, вызвав getName() класса Thread.

Однако, это отличие можно исправить искусственным путем:

В классе NewRunnable, имплементирующем Runnable, добавить в конструктор класса переменную String name, которая будет хранить имя потока. Тогда можно поступить так:

```
NewRunnable runnable2 = new NewRunnable (“поток 1”);
new Thread(runnable2). start();
```

# метод run()

Главный метод класса Thread

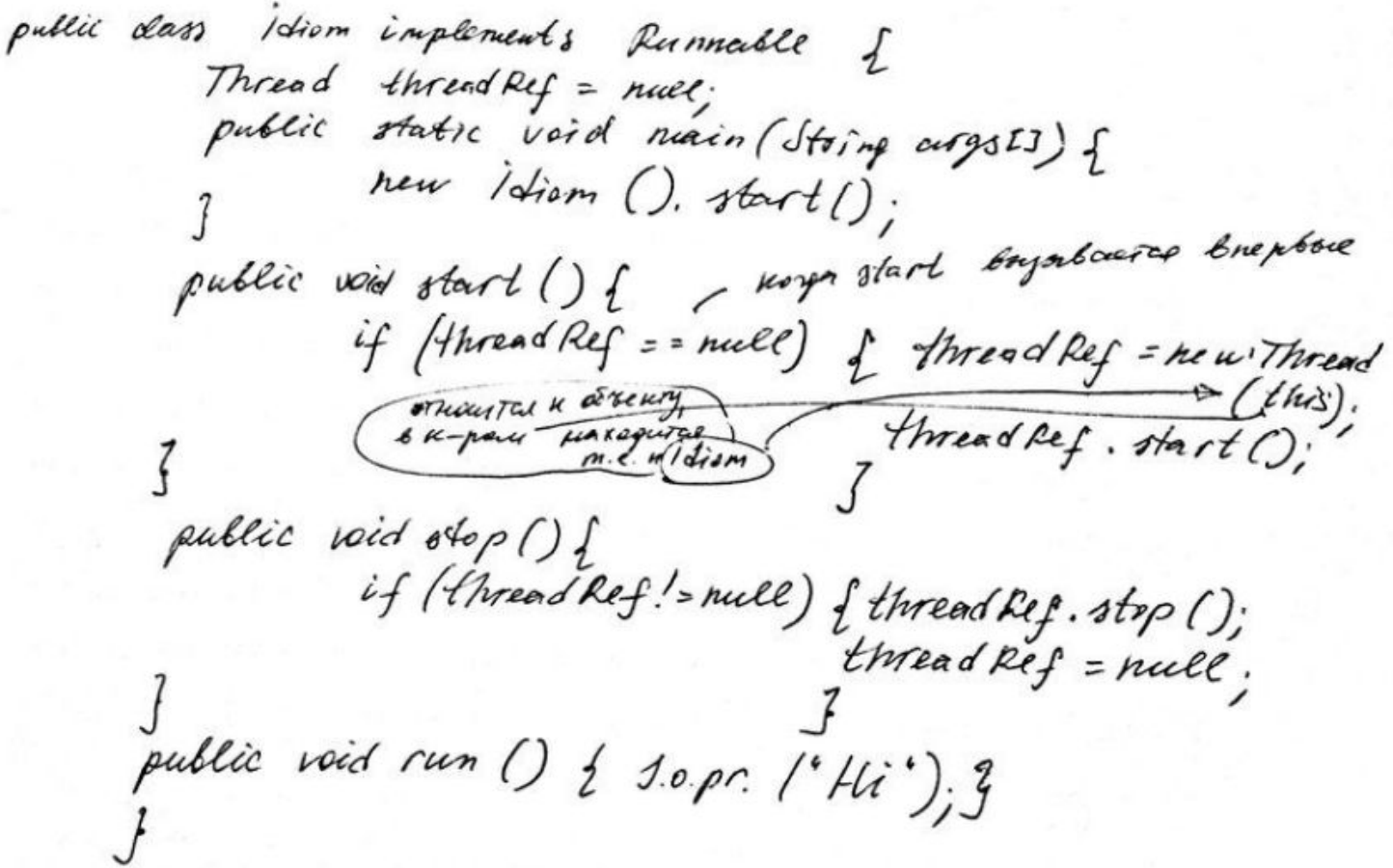
Единственное место, реализующее логику работы потока.

Типичный вид

```
public void run () {
        ... while (true) { ... }
}
```

Забегая вперед: метод run() *не должен быть синхронизирован* в Thread.

# Стандартная идиома потока.
# Пример - класс Idiom, реализующий интерфейс Runnable



# Стандартная идиома потока.
# Пример - класс Idiom, реализующий интерфейс Runnable

Idiom реализует интерфейс Runnable (“запускаемое нечто”)

В момент new Thread(this) объект this уже есть, и конструктор потока сможет сработать.

Метод start() вызывается для объекта threadRe (объекта класса Thread), значит вызовется метод run() класса Idiom.

Метод stop() проверяет, есть ли поток (точнее, выполняется ли поток threadRef), и если да - вызывается метод stop() класса Thread (а не класса Idiom).

# Блокирование ввода/вывода

Поскольку программа часто взаимодействует с вводом/выводом, возможно её зависание на время ожидания ввода от пользователя. Чтобы не “завешивать” всю многопоточную систему, систему обработки ввода/вывода реализуют отдельным потоком.


Ex: Охранное действие:

```
synchronized void waitForCond() { while (checkCondition() == false)
                                    { wait(); }}
void someMethod { waitForCond(); // продолжается с кода, для которого
                                 // условие должно быть истинным
}
```

Метод notifyAll() пробуждает все дожидающиеся потоки в объекте

```
public synchronized void wakeMe () { notifyAll(); }
```

# Охранное действие (предотвращение “завешивания” всех потоков)

Пишется код для потока, так, как если бы он выполнялся несколькими потоками.

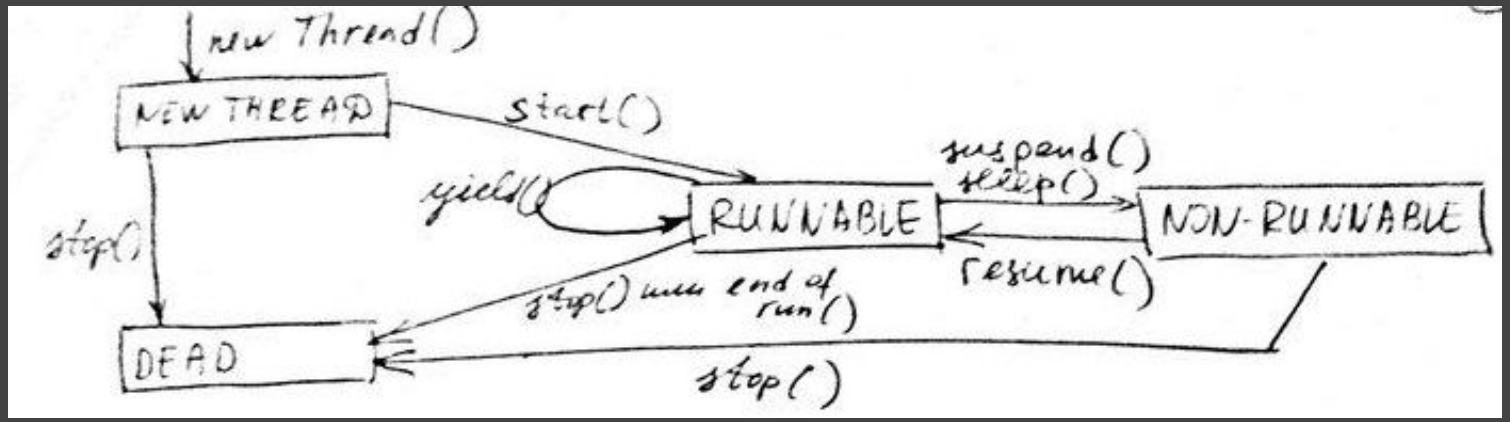
В соответствующих местах ставятся “охранники”: они проверяют логическое условие, и если оно равно FALSE, выполнение входит в цикл while и запускается метод wait(). Пока приложение выполняется, в состоянии wait() может оказаться несколько потоков, задержанных “охранниками”.

Чтобы их отключить, нужно, чтобы условие стало равно TRUE, тогда управление перейдет на метод waitForCond() (т.к. именно этот метод содержит в себе wait()). Где-то в коде есть метод, в котором после waitForCond() есть другие действия, которые начнут выполнятся, когда поток освободят “охранники”.

# Полная диаграмма состояний потока

Метод yield() ставится в стратегических местах кода.

Его цель - прервать поток выполнения и проверить, нет ли другого потока X с таким же приоритетом (RUNNABLE) и дожидающимся выполнения. Если такой поток X находится, то текущий освобождается и начинает выполняться X.
Важно: yield() должен выполняться периодически, чтобы разблокировать очередь ожидания,



# Синхронизация

# Плюсы использования потока

Потоки разумно использовать тогда, когда они повышают производительность или упрощают задачу программирования.

# Минусы использования потока

Все потоки программы разделяют одно пространство кода и пространство данных => потоки могут одновременно изменять одни и те же данные.

```
j=5 //выполняется за 1 такт работы процессора. Все нормально.

Vector i = new Vector();
i=(Vector)j; //структура Vector - более сложная, и требуется больше тактов
```

Период времени, в течение которого выполняется присвоение (время выполнения присвоения), называется **окном уязвимости потока**

В таком случае, есть возможность того, что поток в середине присвоения будет “вытолкнут” из очереди, а когда попытается доделать присвоение, данные будут уже изменены.

# Synchronized

Для защиты от проблемы уязвимости нужен способ получения исключительного управления методом или блоком кода, на период времени, достаточный для завершения операции, уязвимой к неконтролируемым изменениям.

synchronized {блок кода}

гарантирует “закрытие” данных для обеспечения непрерывного эксклюзивного доступа к ним.

public synchronized void someMethod(){...}

String someData = “qwertyuiopasdfghjklzxcvbnm”
synchronized (someData) {...//действия по обработке someData}

# Детализация блокировок

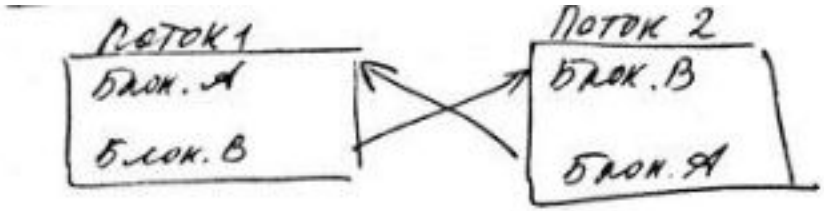
Крупнозернистая - небольшое количество больших блоков синхронизированного кода
уменьшаем количество блоков = уменьшаем время обработки блоков

Минус: данные и методы проводят больше времени, будучи недоступными другим потокам

Мелкозернистая - синхронизации подвергаются только те строки, которые обязательно нуждаются в блокировании

**Общая рекомендация:** вначале использовать крупнозернистую блокировку, оценить производительность, и если не устраивает - то переделать под мелкозернистую блокировку.

# Взаимная блокировка




```
Object a, b;
synchronized (a) { ...
                    synchronized (b) {...}...}

synchronized (b) { ...
                    synchronized (a) {...}...}
```

**Общая рекомендация:** поток не обращается к методам, могущим вызвать этот поток, без проверки возможной рекурсии

# Данные, специфичные для потока

Иногда нужно, чтобы где-то хранились данные, специфичные для конкретного потока. Сложность в том, чтобы оградить эти данные от посягательств других потоков.

Static переменные класса не подходят, т.к. они принадлежат классу, а не конкретному экземпляру.

Можно использовать объект класса Hashtable, который хранит данные в виде пары “ключ-значение”. Тогда ключом будет экземпляр класса Thread, а соответствующим значением - специфичные для потока данные.

# Шаблон класса, *хранящего* данные, специфичные для потока

file ThreadSpecificData.java

```
class ThreadSpecificData {
String name; float data;
ThreadSpecificData (String s, float d)
        {
                name = n; data=d;
        }
float getData() {
                return data;
                }
}
```

# Шаблон класса, *использующего* специфичные для потока данные

```
class ThreadSpecificDataUser extends Thread {
    static Hashtable table;
    static ThreadSpecificData data;
    ThreadSpecificDataUser (String s, float d, Hashtable t) {
        table = t;
        synchronized (table) {
            ThreadSpecificData data = new ThreadSpecificData(String s, float d);
            table.put(data);
                }}
    public void run() { ...
        synchronized (table) {
            Object object = table.get(getName());
            if (object != null) {
                data = (ThreadSpecificData) object;
                float f = data.getData();
            } } }
```

# Шаблон главного класса, *вызывающего* потоки со специфичными данными

```
class MainClass {
        Hashtable thrSpecDataTable;
MainClass () {
        thrSpecDataTable = new Hashtable();
                        }
someFunction() {
        new ThreadSpecificDataUser (‘First thread’, 1.2, thrSpecDataTable).start();
        new ThreadSpecificDataUser (‘Second thread’, 5.8, thrSpecDataTable).start();
                        }
}
```

# Приоритеты потоков

У каждого потока - свой приоритет.

MIN_PRIORITY, MAX_PRIORITY, NORM_PRIORITY

Очередность выполнения потоков - вытесняющая. Как только появляется поток с приоритетом выше, чем у выполняющихся, все потоки с низким приоритетом выталкиваются  из области команд процессора.

**Используйте** в коде потока вызовы yield() и sleep() для балансировки нагрузки с разных потоков.

## Общие рекомендации:

1) цикл событий организовать в отдельном потоке, т.к. обязательным требованием к приложению является требование надежности интерфейса пользователя. События интерфейса пользователя всегда должны быть “под рукой”.
2) Вспомогательные потоки создаются в ответ на специфические запросы: доступ к узлу интернет может породить отдельный поток для связи
3) Разумно учесть системные обращения, которые могут вызвать блокирования, и вместо них создать специальный поток для того, чтобы “переждать блокировки”

| кандидат в поток | свойства | примеры |
|---|---|---|
| да | содержит операции, естественно следующие друг за другом.<br>Более высокая производительность. | GUI, коммуникационные программы |
| возможно | операции нужно анализировать перед тем как разделить | операции над матрицами |
| нет | потоки все усложнят | большие программы |