# МНОГОНИШКОВО ПРОГРАМИРАНЕ С JAVA

*27.11.2018*

- Въведение
- Управление на нишки
- Споделяне на ресурси
- Комуникация между нишки
- Преизползване на нишки
- Атомарни променливи
- Конкурентни колекции


MULTITHREADING
THEORY AND PRACTICE

# КАКВО Е НИШКА?

Отделен път на изпълнение в програма, който се изпълнява конкурентно.

# КОНКУРЕНТНО ИЗПЪЛНЕНИЕ

- Многозадачност
- Многонишковост

| | **Процеси** | **Нишки** |
|---|---|---|
| Стартиране | Бавно | Относително бързо |
| Изолация | Да | Не |
| Комуникация | Бавна | Бърза |

# ПОЛЗИ ОТ МНОГОНИШКОВОТО ИЗПЪЛНЕНИЕ

- Пълноценна употреба на наличните ресурси
- Подобрено потребителско усещане
- Подобрен дизайн

# ПРЕДИЗВИКАТЕЛСТВА

- Сложност на кода
- Източник на грешки

# УПРАВЛЕНИЕ НА НИШКИ

# НИШКИ В JAVA

- управление на нишки – `java.lang.Thread`
- синхронизация при достъп – ключовата дума `synchronized`
- комуникация между нишки – `Object.wait()` / `notify()`

# ДИСПЕЧЕР НА НИШКИ

- Java диспечер (green thread scheduler)
- Диспечер на операционната система

# ДЕФИНИРАНЕ НА НИШКА

- Всяка Java програма при стартирането си съдържа една нишка (main)
- За да създадем нова нишка в Java, наследяваме класа `java.lang.Thread`
- Инструкциите за изпълнение поставяме в метода `run()`, който не връща резултат

```
public class CustomThread extends Thread {
  public void run() {
    System.out.println("Hello asynchronous world!");
  }
}
```

# СТАРТИРАНЕ НА НИШКА

За да стартираме нишка, трябва да:

- инстанцираме класа, представляващ нишка
- извикаме метода `start()` (който вътрешно ще извика `run()`)

# СПИРАНЕ НА НИШКА

- Нишката прекратява изпълнението си автоматично след приключването на метода `run()`
- Нишката не може да бъде стартирана повторно!

## java.lang.Runnable

Нишка може да бъде дефинирана и чрез интерфейса java.lang.Runnable

```
class CustomRunnable implements Runnable {
  public void run() {
    System.out.println("Hello asynchronous world!");
  }
}
```

За да стартираме нишката, подаваме инстанция на конкретния клас на `java.lang.Thread`.

```
Thread customThread = new Thread(new CustomRunnable());
```

## Thread vs Runnable

При употреба на Runnable сме по-гъвкави:

- наследяване на друг клас
- можем да решим да изпълним имплементацията в:
  - друга нишка
  - чрез thread pool
  - в текущата нишка

# Thread API

Програмистът може да именува нишката чрез `setName()`. Имената не са уникални!

```
customThread.setName("Cool thread #1");
```

Също така, нишките могат да се групират логически чрез ThreadGroup. Групата може да се задава само чрез конструктора.

```
// Конструктор, който приема група и име
ThreadGroup coolThreads = new ThreadGroup("Cool thread group")
coolThread1 = new Thread(coolThreads, "Cool thread #1");
coolThread2 = new Thread(coolThreads, "Cool thread #2");
```

# Thread API

```
// „Спане“ – нишката „заспива“ и не получава процесорно време
// за определен интервал време
Thread.sleep(long miliseconds)

// Референция към текущата нишка
Thread.currentThread()

// Stack trace-ът на нишката
Thread.getStackTrace()
```

# Приоритет на нишки

```
// Подсказка към диспечера на нишки, каква част от процесорнот
// време да получи дадена нишка. Скалата е от 1 до 10.
// Приоритетът по подразбиране е 5.
void setPriority(int prio)

// Текущата нишка се отказва от своето процесорно време в полз
// на друга, чийто приоритет е минимум колкото този на текущат
void yield()

// NB!
// Добре структурирано приложение не трябва да разчита на
// приоритетите на нишки или на yield за своята коректност
```

# Прекъсване на нишка

Една нишка може да съобщи на друга да спре изпълнението си чрез:

```
customThread.interrupt()
```

Какво действие ще се извърши в нишката, зависи изцяло от програмиста, възможно е сигналът да се игнорира

Нишката може да провери дали е била прекъсната чрез методите:

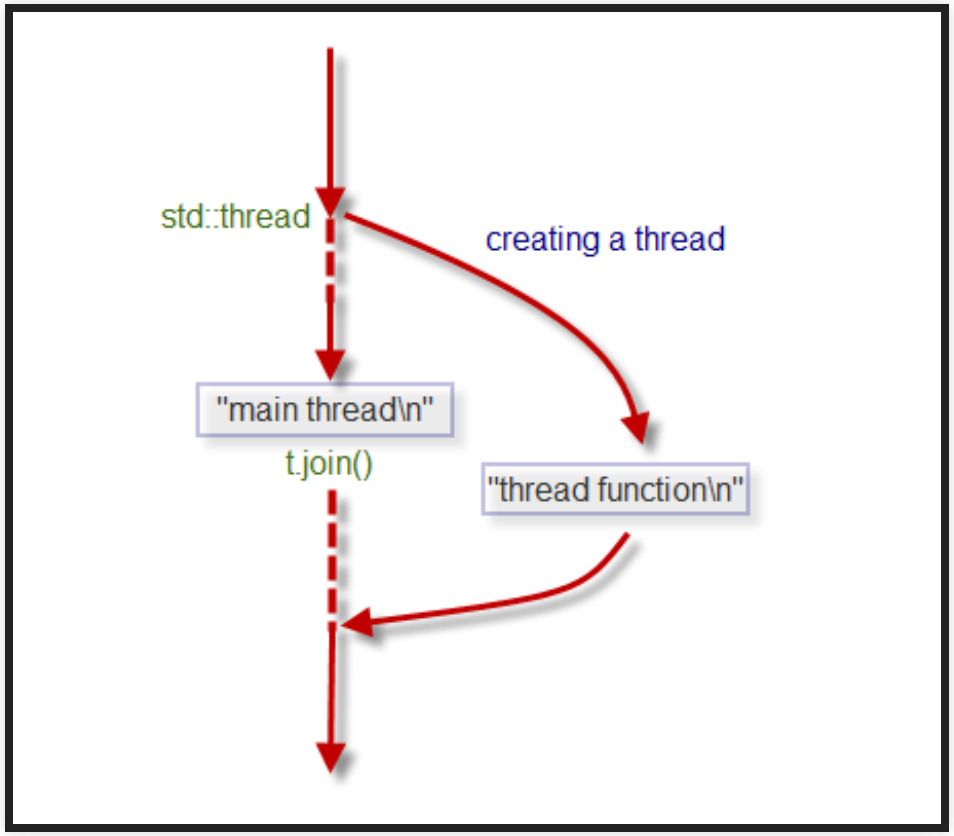
```
Thread.interrupted() // изчиства флага след прочитането (стати
isInterrupted() // не изчиства флага (не-статичен)
```

## java.lang.InterruptedException

Някои методи (`sleep / join`) хвърлят изключение `java.lang.InterruptedException`, когато нишката е получила сигнал за прекъсване, докато те се изпълняват

# Присъединяване към друга нишка

Дадена нишка може да паузира изпълнението си, докато друга нишка приключи, чрез метода join()



# Thread.join()

```
// Извикващата нишка блокира, докато нишката, на която е извик
// join приключи
void join()

// Ако нишката приключи или зададеното време изтече, извикваща
// нишка ще продължи изпълнението си.
void join(long millis)

// Можем да проверим дали дадена нишка не е приключила изпълне
boolean isAlive()
```

## Daemon нишки

Според режима на работа, нишките в Java могат да бъдат два вида:

- Стандартни (non-daemon) нишки
- Демон (daemon) нишки

## Стандартни нишки

- изпълняват задачи, които са свързани с основната идея на програмата
- всяка JVM работи, докато има поне една работеща стандартна нишка

## Daemon

- изпълняват задачи, които не са жизненоважни за програмата
- JVM ще прекрати работата на нишките от този тип, ако няма поне една работеща стандартна нишка

Нишките наследяват режима на работа от тази, която ги е създала.

```
// Може да сменим режима на нишка чрез:
void setDaemon(boolean flag)
```

## Състояние на нишка

- Нишката може да бъде в различно състояние в даден момент от изпълнението си.
- Методът `getState()` ни дава възможност да проверим моментното състояние на нишка.

## Thread.State

enum съдържащ всички възможни състояния:

NEW

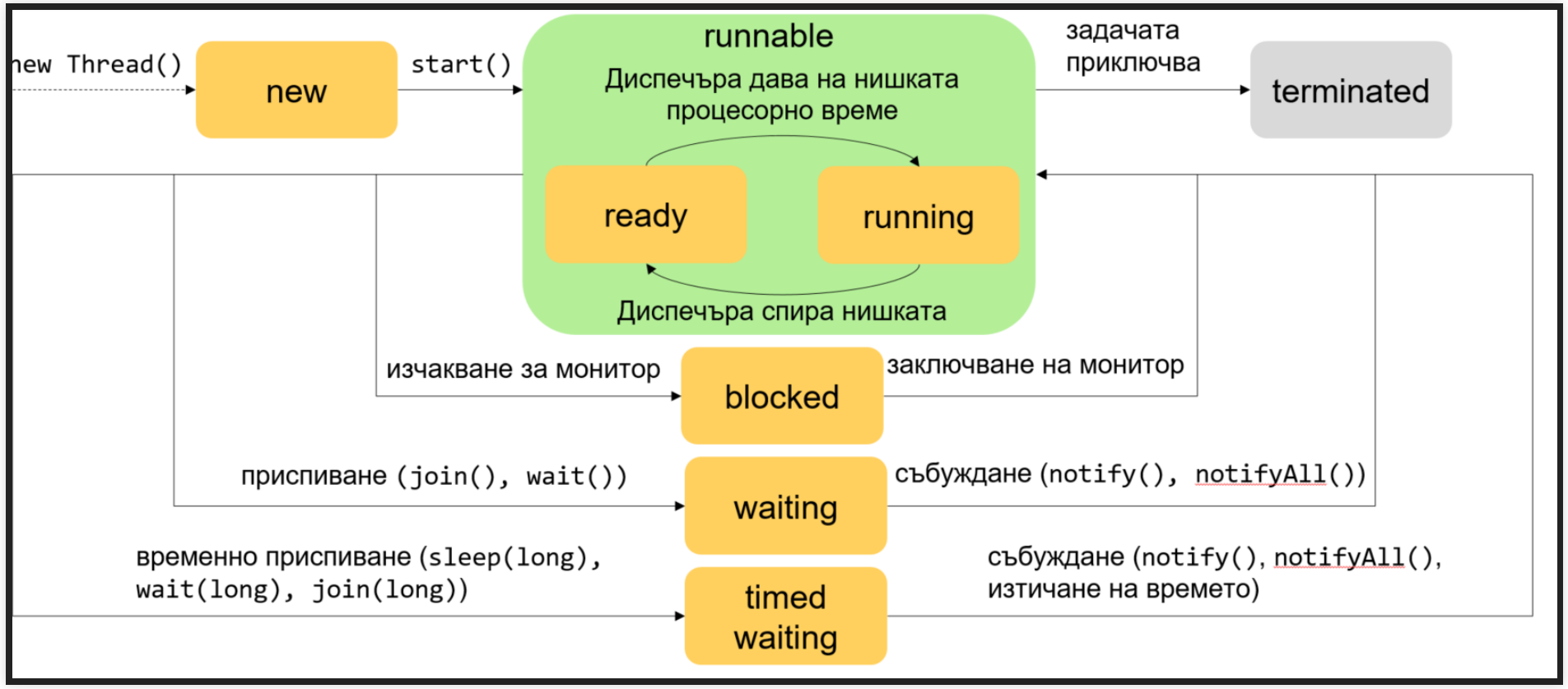
RUNNABLE

BLOCKED

WAITING

TIMED_WAITING

TERMINATED


new Thread()
new
start()
runnable
Диспечъра дава на нишката процесорно време
ready
running
Диспечъра спира нишката
задачата приключва
terminated
изчакване за монитор
blocked
заключване на монитор
приспиване (join(), wait())
waiting
събуждане (notify(), notifyAll())
временно приспиване (sleep(long), wait(long), join(long))
timed waiting
събуждане (notify(), notifyAll(), изтичане на времето)

# СПОДЕЛЯНЕ НА РЕСУРСИ

## Синхронизирана секция

Когато две или повече нишки достъпват конкурентно даден ресурс, който може да бъде променян, е необходима синхронизация.

В Java това се постига чрез ключовата дума `synchronized`. Секцията се състои от:

- монитор – логическа „ключалка“
- блок код, който ще се изпълни ексклузивно от една нишка за даден монитор.

```
public void depositMoney(BankAccount acc, double amount) {
  // Критична секция – една единствена нишка за дадена сметк
  // acc може изпълнява кода в синхронизираната секция
  synchronized (acc) {
    acc.deposit(amount);
  }

  // Не-критична секция - много нишки могат да бъдат тук
  System.out.println("Deposit completed");
}
```

# Синхронизирана секция

- Всеки обект има вътрешен имплицитен монитор (ключалка, lock) т.е. може да се ползва за монитор
- Само една нишка в даден момент за даден монитор може да изпълнява кода (mutex)
- `synchronized` също така задава правила на видимост на данни между нишките

Мониторът се управлява имплицитно от JVM:

- при влизане, ако е свободен, се маркира за „зает“ от съответната нишка
- при влизане, ако не е свободен - нишката блокира
- при излизане lock-ът се освобождава и, ако има блокирани нишки, те могат да се опитат да вземат ключалката

Правило! - всеки достъп на даден ресурс, който може да бъде променян от друга нишка, трябва винаги да става в синхронизирана секция по един и същ монитор!

## Синхронизиран метод

Много често искаме да поставим цялото тяло на даден метод в критична секция. С цел по-четим код, Java ни предлага по-сбит вариант.

```
public void doSomeWork() {
  synchronized (this) {
    // Критична секция - само една нишка може да
    // изпълнява кода за конкретната инстанция 'this'
  }
}
```

## може да се напише съкратено до:

```
public synchronized void doSomeWork() {
  // Критична секция - само една нишка може да
  // изпълнява кода за конкретната инстанция 'this'
  }
}
```

## Синхронизирана секция или метод?

Методът се препоръчва само ако е достатъчно кратък (и бърз за изпълнение) и наистина има нужда цялото тяло да бъде „охранявано"

Синхронизираната секция е за предпочитане, когато:

- нуждаещото се от синхронизация парче код е малка част от метод
- искаме да ползваме монитор, различен от `this`

## Рекурсивност

Lock-ът е рекурсивен (reentrant): нишката, която го „притежава“, може да извиква други критични секции по същия монитор.

```
class Demo {
  public void method1() {
    synchronized (this) {
      // изпълняващата нишка вече притежава lock-а
      // следователно може да извика method2()
      method2();
    }
  }

  public synchronized void method2() {
  }
}
```

## Използване на монитор, различен от `this`

В определени случаи, може да изберем да синхронизираме достъпа в дадена инстанция на обект чрез „отдадена“ член променлива.

```
private final Object dedicatedMonitor = new Object();
```

- енкапсулираме монитора, но използваме допълнителна памет
- даден клас има нужда от различни монитори за охраняване на различно състояние

# Използване на няколко монитора

Една нишка може да „притежава“ много на брой монитори, стига те да са свободни:

```
// NB! Държането на няколко ключалки е лоша практика и
// при възможност трябва да се избягва !!!
public void multipleLocks() {
  synchronized (lock1) {
    // нишката притежава lock1
    synchronized (lock2) {
      // нишката притежава lock1 & lock2
      synchronized (lock3) {
        // нишката притежава lock1, lock2 & lock3
      }
      // нишката притежава lock1 & lock2
    }
    // нишката притежава lock1
  }
}
```

## Синхронизация между инстанции на клас

- Подходящо е да ползваме ключалка, обща за класа
- … разбира се, най-удобно е да ползваме статична променлива за монитор.

Всяка инстанция на клас има статична референция към обекта на класа, към който принадлежи. Можем да я достъпим чрез:

```
BankAccount.class // статично
this.getClass() // чрез ‘this’
```

# Статични синхронизирани методи

Java отново ни предоставя съкратен вариант:

```
static void incrementOpCount() {
  synchronized (BankAccount.class) {
    opCount++;
  }
}
```

може да се напише съкратено като:

```
static synchronized void incrementOpCount() {
  opCount++;
}
```

## Thread-safe обекти

Някои обекти са thread-safe по подразбиране:

- Локални обекти
- Stateless обекти
- Immutable обекти
- Обекти, които са ефективно final (read-only)

## volatile променливи

- JVM гарантира, че простите операции върху примитиви с размер до 32 бита ще бъдат третирани като неделими.
- Това изключва променливите от тип `long` и `double`

Четенето и писането на 64 бита може да бъде разделено на 2 операции, което може да доведе до неконсистентност:

```
1. Изпълнява се първата 32 битова операция
2. Диспечерът на нишките сменя контекста на процесора
3. По същото време, друга нишка достъпва 64-битовата
   променлива и работи с частично променени данни
```

Дефинирането на променлива като volatile ни гарантира:

- Атомарност – простите операции върху 64-битови примитиви са неделими
- Видимост – стойността на променливата се чете и съхранява директно в паметта.

NB! Ключовата дума volatile осигурява атомарност само и единствено на четене и записване на примитиви.

```
private volatile int counter = 0;

public void incrementCount() {
  // Не е thread-safe, това са 3 операции.
  // Трябва ни критична секция
  counter++;
}
```

# ПРОБЛЕМИ ПРИ КОНКУРЕНТЕН ДОСТЪП ДО РЕСУРСИ

## Race condition

- Коректността на приложението зависи от реда на изпълнение на нишките
- Проблемът може да стои незабелязан дълго време
- Обикновено е трудно откриваема грешка

## Бързодействие

- Синхронизацията е относително бавна операция
- Ако критичната ни секция е ненужно голяма, може да забави изпълнението, понеже нишките се изпълняват серийно.

- Препоръчително е да поставим в критична секция единствено кода, който достъпва променливи, които могат да бъдат конкурентно модифицирани.
- NB! Винаги правете кода си първо коректен, а после бързодействащ!

## Съставни операции

Ако комбинираме няколко thread-safe операции в една по-сложна (обща), нямаме никаква гаранция, че те ще се изпълнят атомарно

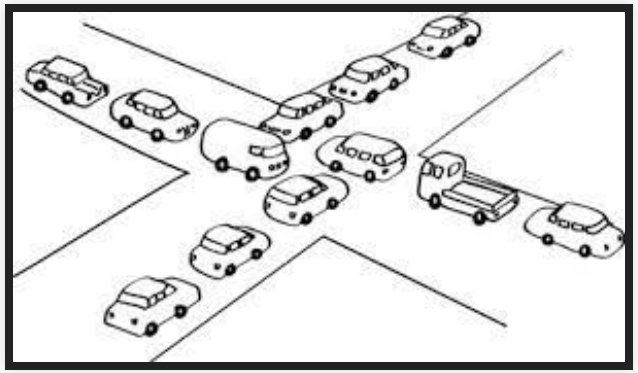
```
public synchronized void withdraw(double amount) {
  this.balance -= amount;
}

public synchronized double getBalance() {
  return balance;
}

// Бъг! - този метод също трябва да е синхронизиран!
public void verifyAndWithdraw(double amount) {
  if (getBalance() >= amount) {
    withdraw(amount);
  }
}
```

## Мъртва хватка (Deadlock)

Получава се, когато две или повече нишки се блокират една друга, всяка от тях притежаваща ключалка, от която друга нишка има нужда, но чакайки за ключалка, която някоя от другите нишки притежава.



- Нишките не могат да бъдат прекратявани отвън.
- Ключалките не могат да бъдат отнемани насилствено.
- Единственият изход от мъртва хватка е рестартиране на JVM.

## Предотвратяване

- мониторите да се вземат винаги в същия ред
- използване на един общ монитор

За съжаление, на практика това често е невъзможно. В такива случаи:

- избягваме притежаване на повече монитори
- подреждаме мониторите спрямо някакъв признак

# Комуникация между нишки

## Busy wait

Неефективен (и възможно некоректен) начин за комуникация е т.нар. busy wait: в цикъл изчакваме събитието да се случи

```
// Всеки месец се пуска нова нишка, която тегли сумата на ме
// от сметката на кредитополучателя
public void withdrawCreditPayment(double monthFee) {
  while (this.balance < monthFee) {
    // Стоим в цикъла, докато няма достатъчно пари да погася
    Thread.sleep(1000);
  }
  balance -= monthFee;
}
```

# Изчакване чрез wait()

Нишка може да заяви, че иска да изчака, докато дадено събитие се случи в друга нишка, чрез метода wait() на java.lang.Object

```
public synchronized void withdrawCreditPayment(double monthF
  while (this.balance < monthFee) {
    try {
      // Изчакваме и освобождаваме монитора this
      this.wait();
    } catch (InterruptedException e) {
      // хвърля грешка, ако нишката бъде прекъсната
    }
  }
  balance -= monthFee;
}
```

## java.lang.Object.wait()

- извикването на `wait()` винаги става в критична секция по обекта, който ползваме за монитор
- в противен случай се хвърля `java.lang.IllegalMonitorStateException`
- `wait()` освобождава текущия монитор и нишката преминава в статус “WAITING”

Възможно е да извикаме `wait()` с аргумент за време – по този начин нишката ще се събуди, ако бъде известена или времето изтече.

Винаги след събуждане проверявайте в цикъл дали събитието, за което чакате, в действителност е настъпило: шаблон, известен като “guarded wait”.

# Известяване чрез notify()

Нишка може да извести чакащи нишки, че дадено събитие се е случило, и те могат да продължат своето изпълнение. Това става чрез метода `notify()` на `java.lang.Object`

```
// При депозиране на пари по сметка, уведомяваме чакащите ни
// че са постъпили средства по сметката
public synchronized void deposit(double amount) {
  this.notify();
  this.balance += amount;
}
```

## java.lang.Object().notify()

- Събужда една нишка, чакаща за съответния монитор
- Извикването на `notify()` винаги става в критична секция по обекта, който ползваме за монитор
- не освобождава монитора

Имаме два варианта:

- notify() – събужда една нишка
- notifyAll() – събужда всички нишки

Ако имаме няколко чакащи нишки за текущия монитор, нямаме възможност да кажем коя/кои нишки да бъдат събудени или да бъдат събудени първи

## notify() vs notifyAll()

`notify()` – събужда една произволна нишка, чакаща за този монитор. Полезно е само когато сме сигурни, че само една нишка може да продължи изпълнението си, и не искаме да „платим“ цената да събудим всички

`notifyAll()` – събужда всички нишки. В много случаи, повече от една нишка може да продължи действието си след известяването си. Нишките се изпълняват последователно в синхронизираната секция по монитора след събуждането си.

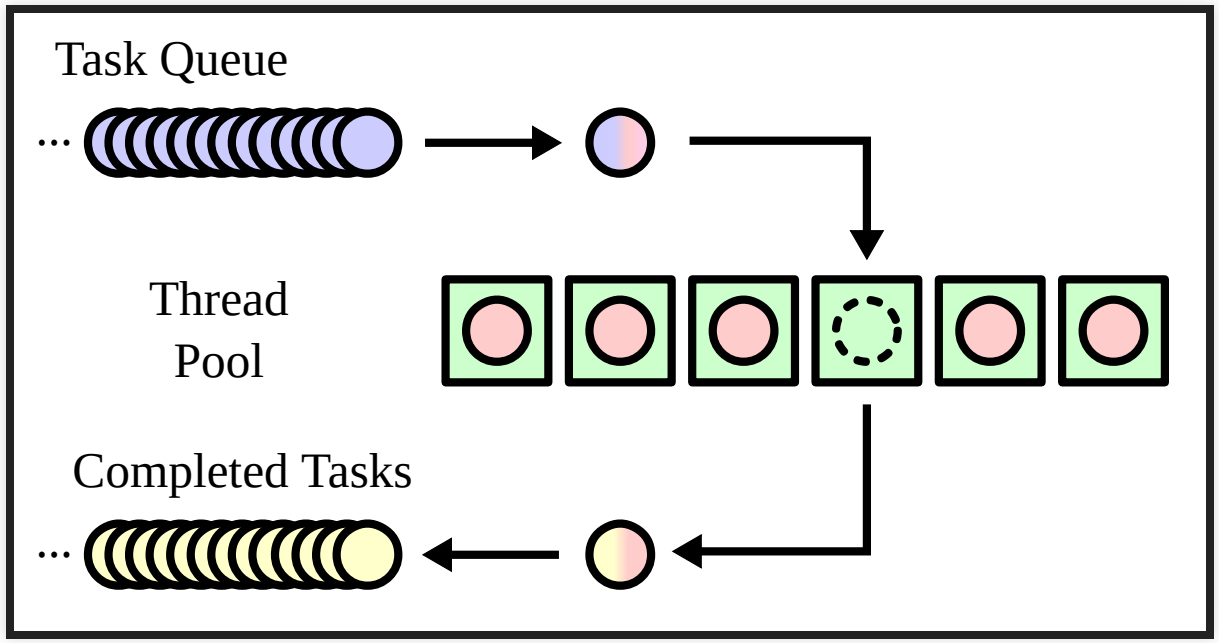
В практиката по-често се ползва `notifyAll()`. Правилно имплементирана проверка в `while` може да ни гарантира същото поведение като при `notify()` – макар и събудени повече нишки – те ще заспят отново, след като видят, че не могат да продължат.

# ПРЕИЗПОЛЗВАНЕ НА НИШКИ

## Thread pool (executor)

Концепция в конкурентното програмиране, при която „рециклираме“ нишките след края на тяхното изпълнение, с цел оптимизация.

# Задава абстракция от управлението на жизнения цикъл на нишката (създаване, старт/стоп, преизползване)



Отделните runnable обекти се третират като „задачи“ и се трупат в опашка, и когато има свободни нишки в pool-а, те изпълняват задачите на базата на зададени правила.

# Executors API

```
// централен интерфейс
java.util.concurrent.Executor
void execute(Runnable command)

// добавя възможност и за изпълнение на Callable обекти,
// които, за разлика от Runnable, могат да върнат резултат
java.util.concurrent.ExecutorService
<T> Future<T> submit(Callable<T> task)

// задачите могат да се пускат след опредено закъснение
// или периодично на зададен интервал
java.util.concurrent.ScheduledExecutorService
ScheduledFuture schedule(Runnable r,long delay, TimeUnit tu)
ScheduledFuture scheduleAtFixedRate(Runnable r,long delay,
                                    long period, TimeUnit tu)
```

# Създаване на Executor

```
// предоставя статични методи фабрики за създаването на pools
java.util.concurrent.Executors

// pool-ът ще се състои само от една нишка, следователно
// задачите ще се изпълняват последователно
static ExecutorService newSingleThreadExecutor()

// създава pool, който ще се състои от фиксиран брой нишки.
// Ако в опашката има повече задачи, отколкото налични нишки,
// задачите не се обработват, докато не се освободи нишка
static ExecutorService newFixedThreadPool(int n)

// създава pool от нишки, който ще преизползва нишките,
// ако има налични, в противен случай ще направи нова.
// Нишките, които не се използвани през последната минута,
```

## Спиране на Thread pool

Executor обект винаги трябва да бъде експлицитно спрян с метода shutdown()

# Атомарни променливи

- част от `java.util.concurrent.atomic`
- предоставя атомарни имплементации на примитиви, масиви от примитиви и абстракция за атомарна референция
- AtomicBoolean, AtomicInteger, AtomicLong, AtomicIntegerArray, AtomicLongArray, AtomicReference

- Имат същите свойства като volatile променливи, като в допълнение предоставят възможност за атомарни съставни операции
- Използват специални хардуерни инструкции (“compare-and-swap”, CAS), които позволяват избягването на ключалки

# Атомарни операции

```
// Всички имплементации имат методи get() и set() за достъп до
// съхраняваната променлива. Те са еквивалентни на четене и
// модификация на volatile променлива

// Няколко примера за различни атомарни операции

// thread-safe вариант на ++i (i=i+1)
atomicInt.incrementAndGet();

// thread-safe вариант на i += x (i=i+x)
atomicInt.addAndGet(x)

// thread-safe вариант на if (ref == expected) { ref = update;
// NB! Сравняваме референции затова използваме `==`, а не equa
atomicRef.compareAndSet(expected, update)
```

# КОНКУРЕНТНИ КОЛЕКЦИИ

Collections API предоставя имплементации на няколко thread-safe колекции като: Vector, Hashtable

`java.util.Collections` предоставя метод фабрика, с който можем да създадем thread-safe колекция от съществуваща такава.

```
static <T> Collection<T> synchronizedCollection(Collection<T>
```

Имат няколко недостатъка:

- не са достатъчно бързи при много конкурентни ползватели
- не предоставят възможност за атомарни операции

# Конкурентни колекции

- част от `java.util.concurrent`
- създадени специално за работа в конкурентна среда
- добавят възможности за:
  - Lock-free паралелен достъп
  - Fail-safe итератори
  - Атомарни операции (пр. putIfAbsent)

# CopyOnWriteArrayList

- Алтернатива на синхронизираните имплементации на ArrayList
- Позволява lock-free паралелно четене
- “Fail-safe snapshot” итератор
- Всяка модификация предизвиква копиране на масива
- Атомарни операции: boolean addIfAbsent(E e);

NB! Използването на тази структура е подходящо само когато броят на четенията от масива значително надвишава броя на модификациите. В противен случай, структурата е изключително неефективна

## ConcurrentHashMap

- Алтернатива на синхронизираните версии на java.util.HashMap
- Паралелен lock-free достъп за четене
- Паралелен (но лимитиран) достъп за писане
- Fail-safe и “Weekly consistent“ итератор
- Атомарни операции: V putIfAbsent(K key, V value)

` Най-популярната колекция от `java.util.concurrent` библиотеката, почти винаги е подходяща да се използва за замяната на съществуващите thread-safe варианти на HashMap

## *BlockingQueue*

Имплементация на блокираща опашка
(“Producer-Consumer” опашка)
Блокира, когато:

- опашката е празна и някой се опитва да чете от нея. При първи запис, бива нотифицирана
- опашката е пълна и някой се опитва да пише в нея. При първо четене, бива нотифицирана

ArrayBlockingQueue – основна имплементация. Опашката пази елементите си в масив, който не може да променя размера си.

```
BlockingQueue<String> queue = new ArrayBlockingQueue<>(4);
```

## *Използвана литература*

- "Java Concurrency in Practice" by Brian Göetz, Tim Peierls, Joshua Bloch
- http://docs.oracle.com/javase/tutorial/essential/concurren
- http://www.javaworld.com/article/2071214/java-concurrency/java-101--understanding-java-threads-part-3-thread-scheduling-and-wait-notify.html
- https://docs.oracle.com/javase/8/docs/api/