ЛЕКЦИЯ I

JAVA OOP:
Многонишково програмиране

Теодор Костадинов

Съдържание

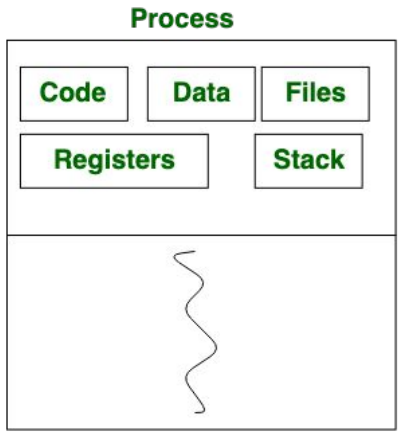
Нишки и процеси

Асинхронно изпълнение

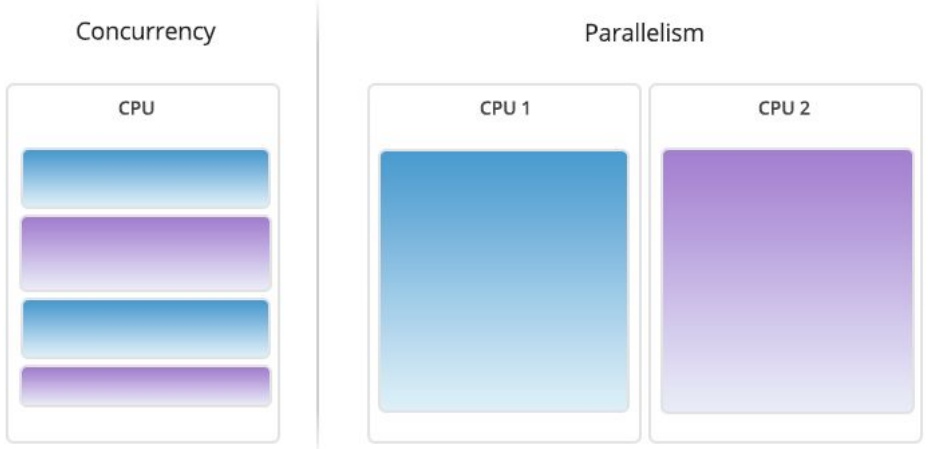
SOFTWARE ACADEMY
Educational Center Net It

- Преди появата на операционните системи: програмираш компютъра за всяко пускане
- Еднозадачни операционни системи: компютъра изпълнява само една програма, която се задава при стартирането му
- Многозадачни операционни системи. Процеси. Комуникация между процеси: компютрите могат да изпълняват повече от една задача едновременно
- Едно- и многопроцесорни системи, hyper-threading (2002), многоядрени процесори (2005)

Нишка или Процес или Ядро

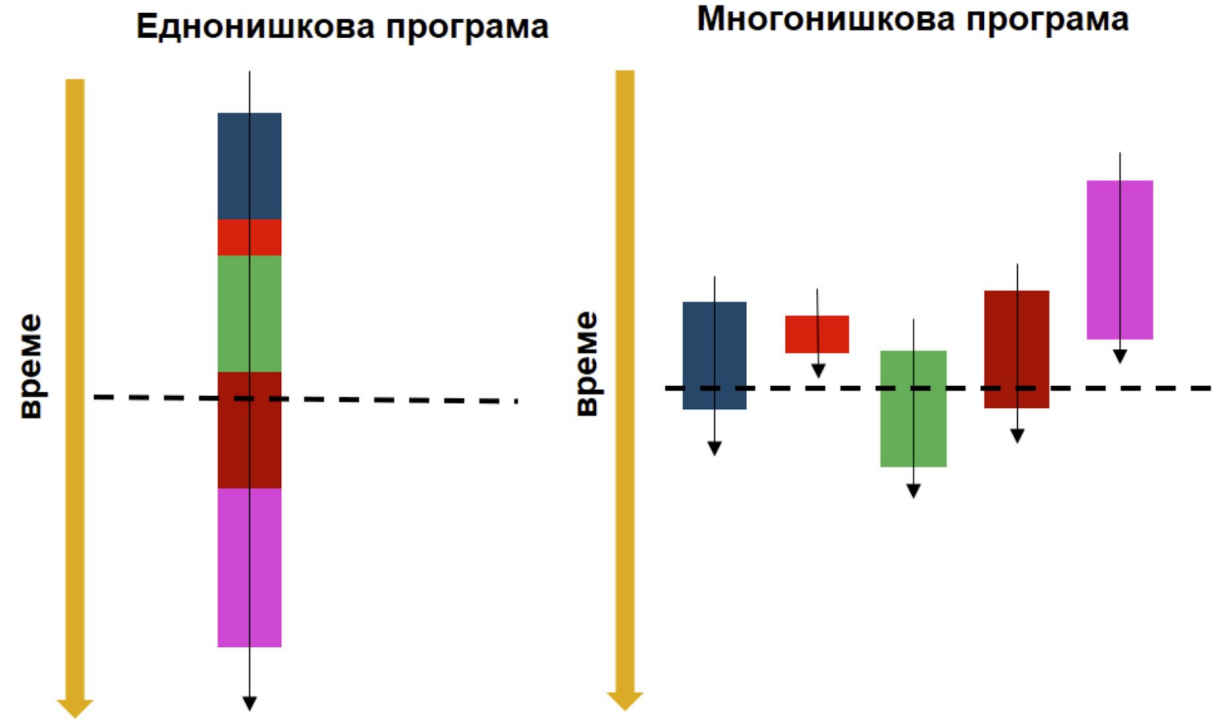


- Нишка е отделен път на изпълнение в програма, който се изпълнява конкурентно.
- Нишката е част от процес.
- В един процес може да има много нишки.
- Процесът е комбинацията от код на програмата, файлове на програмата, стек и хийп на програмата.
- Една програма се изпълнява в един процес.
- Благодарение на нишките, можем да изпълняваме няколко парчета код едновременно (или почти)
  - Ако имаме само 1 ядро - симулираме асинхронност, чрез много бързо превключване между нишките
  - Ако имаме 2 и повече ядра - няколко нишки могат реално да се изпълняват едновременно



Нишка или Процес или Ядро

- Когато в една програма изпълняваме кода на много нишки, това означава, че той може да се изпълни паралелно, което намалява времето
- Пълноценна употреба на наличните ресурси
- Подобрено потребителско изживяване
- По-проста архитектура



Вместо сложен, асинхронен код → няколко прости, последователни workflows, изпълнявани в отделни нишки

Безопасност

- Thread-safety касае управлението на достъпа до състоянието на обектите
- Thread-safe данни са такива, които сме предпазили от счупване, ако се използват от няколко нишки едновременно
- Ако две нишки достъпят non-thread safe метод/променлива, има шанс да я счупят
- Клас е thread-safe, ако работи коректно при конкурентен достъп от множество нишки
- thread safety цели защитата на данните от неконтролиран конкурентен достъп
- thread-safe класовете енкапсулират евентуалната необходима синхронизация, така че ползвателите им да не се грижат за нея
- Когато повече от една нишка достъпва състоянието на обект и поне една нишка може да го промени, всички тези нишки трябва да синхронизират достъпа си до това състояние

Опасен код

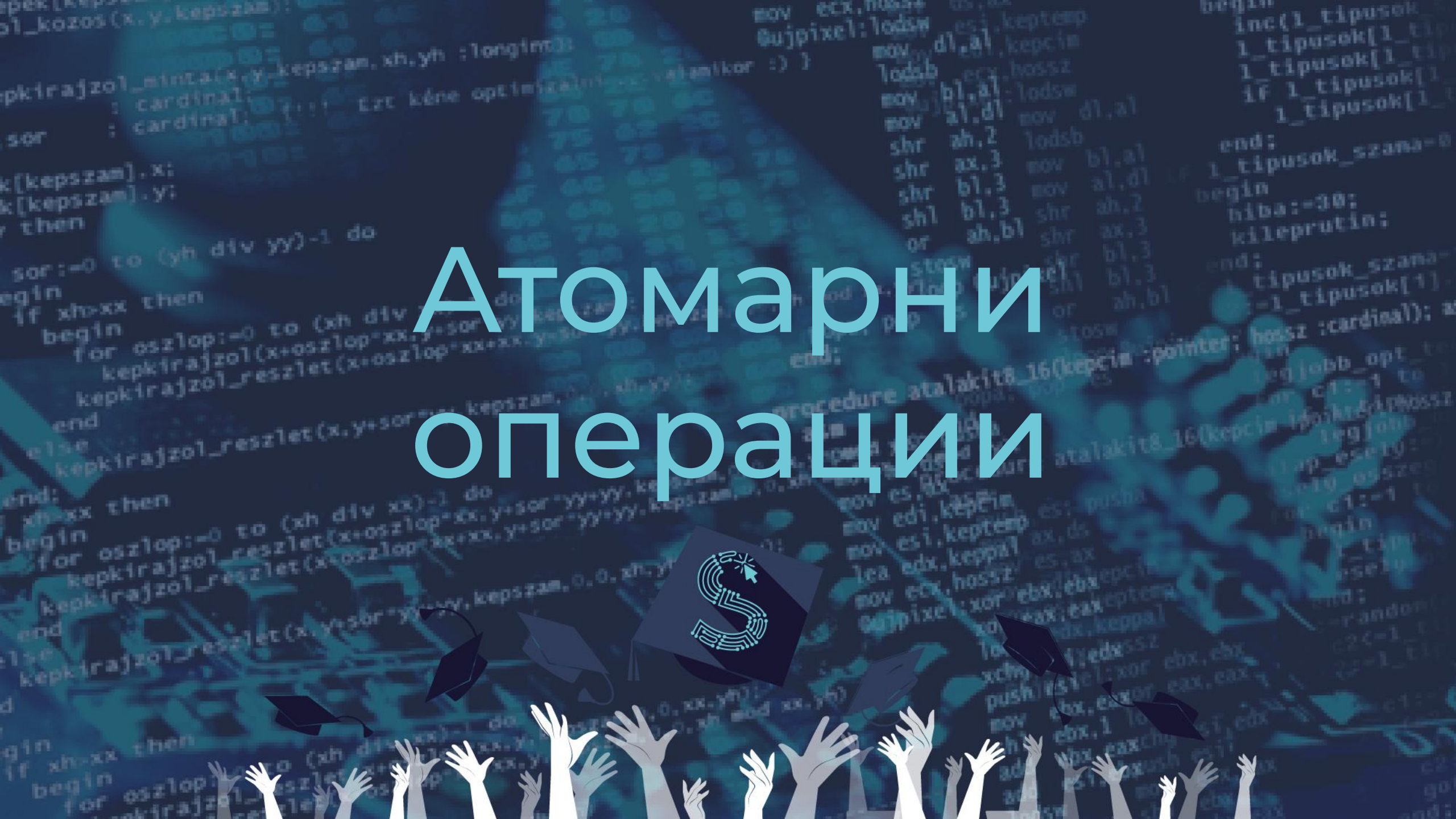
SOFTWARE ACADEMY
Educational Center Net It

```
public class Counter {
    int counter;

    public void increment() {
        counter++; //това всъщност са три операции
    }
}

//another class
Counter counter = new Counter();

//Thread 1: counter.increment();
//Thread 2: counter.increment();
```

БЕЗопасен код


```
public class SafeCounterWithLock {
    private volatile int counter;

   //само една нишка едновременно има достъп до този метод
   //но това е скъпо удоволствие - програмата става бавна,
   //защото другите нишки ще я чакат да приключи
    public synchronized void increment() {
        counter++;
    }
}
```

Атомарни операции



Атомарност

- Атомарност ще рече единичност - няколко операции са събрани в една
- Има няколко предварително създадени атомарни операции, които можем да използваме наготово.
- Те ни позволяват да избегнем синхронизирането на кода (което го прави бавен), но и избягваме проблеми при множество от нишки (защото имаме само по 1 операция)
- При използването на атомарни операции
  - Ако много нишки правят промени по променлива, само една нишка ѝ се дава разрешение
  - Другите получават отговор - фейлнала операция и могат да опитат отново
  - По този начин нишките не се приспиват и програмата работи бързо

Атомарни класове

- Има няколко готови атомарни класа - AtomicBoolean, AtomicInteger, AtomicLong, AtomicIntegerArray, AtomicLongArray, AtomicReference<ActualType>

```
public class SafeCounterWithoutLock {
    private final AtomicInteger counter = new AtomicInteger(0);

    public int getValue() {
        return counter.get();
    }
    public void increment() {
        while(true) {
            int existingValue = getValue();
            int newValue = existingValue + 1;
            if(counter.compareAndSet(existingValue, newValue)) {
                return;
            }
        }
    }
}
```

Примери

- Има няколко готови атомарни класа - AtomicBoolean, AtomicInteger, AtomicLong, AtomicIntegerArray, AtomicLongArray, AtomicReference<ActualType>

```
// thread-safe вариант на ++i (i=i+1)
System.out.println(atomicInt.incrementAndGet()); // 2020

// thread-safe вариант на i += x (i=i+x)
System.out.println(atomicInt.addAndGet(5)); // 2025

AtomicReference<String> atomicRef = new AtomicReference<>();
// atomicRef не е null, но стойността, която съдържа (wrap-ва), е

// thread-safe вариант на if (ref == expected) { ref = update; }
// NB! Сравняваме референции затова използваме `==`, а не equals
atomicRef.compareAndSet(null, "Happy new year!");
System.out.println(atomicRef); // Happy new year!
```

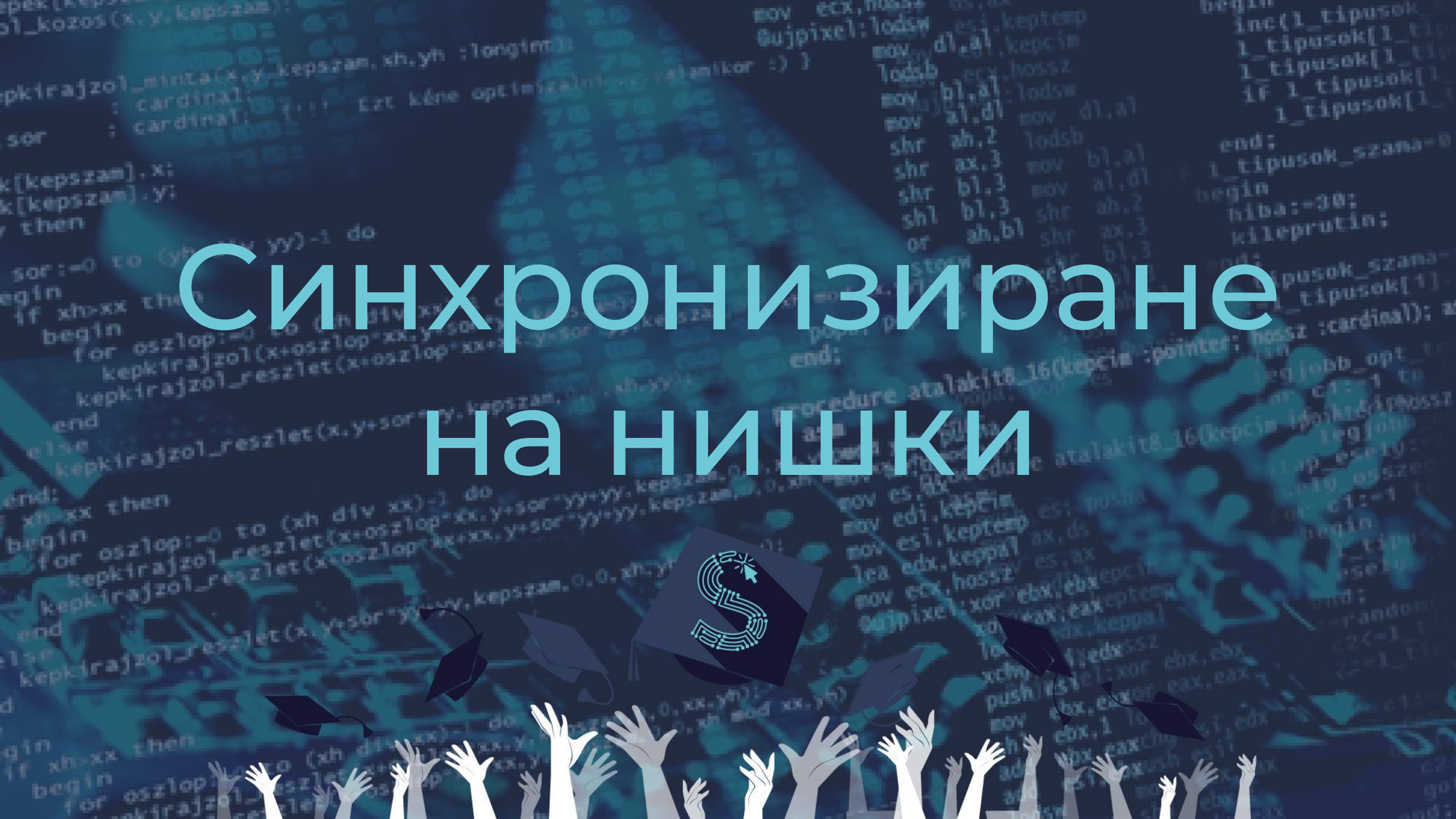
Операции над Атомарни класове

- get() – взима стойността от паметта
- set() – записва нова стойност в паметта
- lazySet() – записва нова стойност в паметта неприоритетно. Операцията може да бъде комбинирана с други подобни операции върху паметта на тази променлива.
- compareAndSet() – прави опит за записване на нова стойност, връща true ако е успешен

Почивка

до 20:20

Синхронизиране на нишки



Синхронизиране

- Когато две или повече нишки достъпват конкурентно даден ресурс, който може да бъде променян, е необходима синхронизация
- Постига се 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 сметка
    // acc може да изпълнява кода в синхронизираната секция
    synchronized (acc) {
        acc.deposit(amount);
    }

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

Монитори

- 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 == mutual exclusion)
- Всеки достъп до дадено състояние, което може да бъде променено от друга нишка, трябва винаги да става в синхронизирана секция по един и същ монитор
- Монитора работи като:
  - при влизане, ако е свободен, се маркира за „зает“ от съответната нишка
  - при влизане, ако не е свободен, нишката блокира и чака
  - при излизане, lock-ът се освобождава и, ако има блокирани нишки, те могат да се опитат да вземат ключалката

Монитори

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

```
// 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
    }
}
```

Безопасни обекти

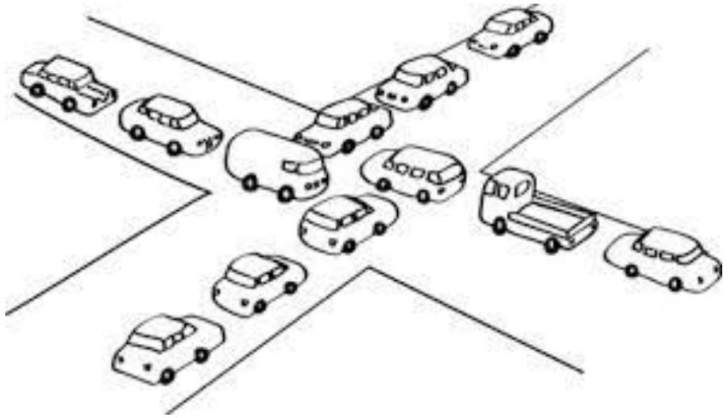
SOFTWARE ACADEMY
Educational Center Net It

- Някои обекти са безопасни по подразбиране
  - Локални обекти (локални променливи на метод)
  - Stateless обекти
  - Immutable обекти
  - Обекти, които са ефективно final (read-only)

Deadlock

SOFTWARE ACADEMY
Educational Center Net It

- 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



Живот на нишка

Създаване на нишка


```
// Option 1: extend java.lang.Thread
public class CustomThread extends Thread {
    public void run() {
        System.out.println("Hello asynchronous world!");
    }
}
Thread customThread = new CustomThread();

// Option 2: implement java.lang.Runnable
public class CustomRunnable implements Runnable {
    public void run() {
        System.out.println("Hello asynchronous world!");
    }
}
Thread customThread = new Thread(new CustomRunnable());
```

- Всяка Java програма при стартирането си съдържа една нишка (main)
- За да създадем нова нишка в Java, наследяваме класа java.lang.Thread или имплементираме интерфейса java.lang.Runnable
- Логиката (инструкциите за изпълнение) е в метода run()

Стартиране на нишка


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

- За да стартираме нишка, трябва да:
  - инстанцираме класа, наследяващ Thread или
  - инстанцираме Thread и да подадем като аргумент имплементация на Runnable
- извикаме метода start() (който вътрешно ще извика run())

Спиране на нишка

```
???

customThread
.setDaemon(true);
```

- Нишка не може да бъде спряна експлицитно веднъж щом е стартирана
- Нишката прекратява изпълнението си автоматично след приключването на метода run()
- Нишката не може да бъде стартирана повторно
- Според режима на работа, нишките в Java могат да бъдат два вида:
  - Стандартни (non-daemon) нишки: изпълняват задачи, които са свързани с основната идея на програмата. Всяка JVM работи, докато има поне една работеща стандартна нишка
  - Демон (daemon) нишки: 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, или го задават експлицитно

Управление на нишка

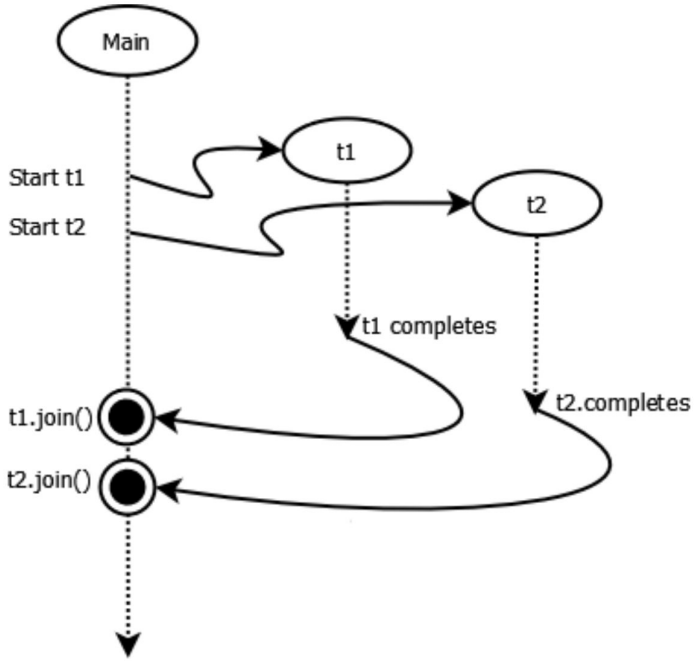
- Нишка могат да си имат имена и група. Те не влияят на работата им.

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

// 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");

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

// Референция към текущата нишка
Thread.currentThread()

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

Присъединяване към нишка

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

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

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

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



ВЪПРОСИ?

Резюме

- Когато използваме много нишки трябва да внимаваме за сигурността на данните
- Многонишковото програмиране може да ускори изпълнението на програмата ни

Ресурси

- Docs
- GitHub Repo with Demos
- https://www.baeldung.com/java-atomic-variables
- Other presentation

БЛАГОДАРЯ
ЗА ВНИМАНИЕТО!