

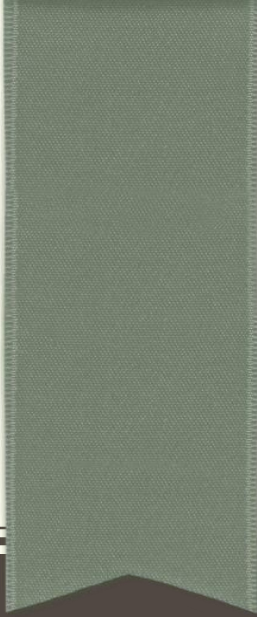
ПРОГРАМУВАННЯ JAVA- ДОДАТКІВ З ПІДТРИМКОЮ БАГАТОПОТОЧНОСТІ

Лекція 06
Java-програмування



План лекції

- Основи реалізації потоків у Java Threads API.
- Низькорівнева синхронізація потоків.
- Використання ексекуторів при розробці багатопоточних додатків.
- Синхронізатори з Java Concurrency Utilities.
- Огляд роботи з багатопоточними колекціями.



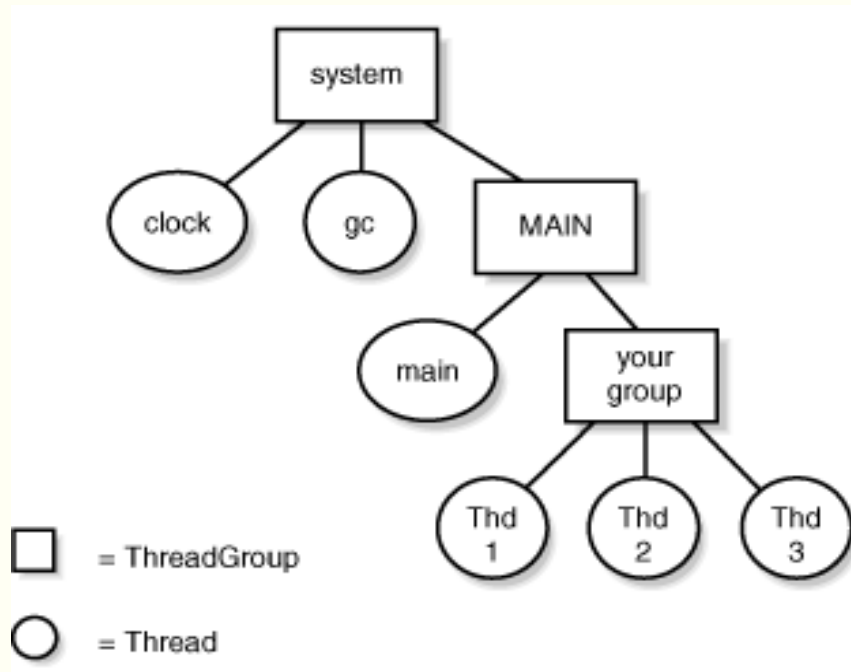
ОСНОВИ РЕАЛІЗАЦІЇ ПОТОКІВ У JAVA THREADS API

Питання 6.1.

Основи багатопоточності

- Застосунки виконуються за допомогою потоків (threads) – незалежних шляхів виконання їх коду.
- При багатопоточному виконанні коду кожен потік має власний шлях.
 - Наприклад, один потік може виконувати один case, а інший – інший case зі switch-умови.
- Додатки використовують потоки для підвищення продуктивності.
 - Деякі застосунки використовують лише головний (master) потік, який виконується в методі main().
 - Решта може потребувати додаткових потоків для тривалих фонових задач – таким чином вони зможуть реагувати на дії користувача.

API для підтримки багатопоточності



- Віртуальна машина дає кожному потоку власний стек виклику методів, щоб потоки не заважали один одному.
 - Також стек забезпечує для кожного потоку власну копію параметрів методу, локальних змінних та значень, що повертаються.
- Засоби підтримки потоків у Java:
 - **Threads API** – низькорівневий API, який складається з одного інтерфейсу (Runnable) та чотирьох класів (Thread, ThreadGroup, ThreadLocal, InheritableThreadLocal) в пакеті java.lang.
 - **Concurrency Utilities Framework** – високорівневий API з більш розвиненими можливостями. Рекомендований для розробки багатопоточних застосунків, знаходиться в пакеті java.util.concurrent.

Java-інтерфейс Runnable

- визначає об'єкти, які забезпечують виконання необхідної частини коду в межах єдиного методу (потoku) цього інтерфейсу – run().
 - не приймає ніяких параметрів та не повертає значень.

```
Runnable r = new Runnable()
{
    @Override
    public void run()
    {
        // perform some work
    }
};
```

```
new Thread(() ->
    System.out.println("Hi, I'm Thread")
).start();
```

Клас Thread і його конструктори

- Клас Thread забезпечує відповідний інтерфейс з архітектурою потоків фізичної операційної системи, яка зазвичай відповідає за створення та управління потоками.
 - Один потік операційної системи асоціюється з об'єктом Thread.
- Thread оголошує декілька конструкторів для ініціалізації об'єктів.
 - Thread(Runnable runnable) ініціалізує новий об'єкт Thread для заданого runnable, код якого буде виконуватись:
Thread t = new Thread(r);
 - Інші конструктори, зокрема Thread(), не приймають Runnable аргументів.
 - Тому необхідно розширити Thread та переозначити відповідний метод run()

Клас Thread і його конструктори

```
class MyThread extends Thread
{
    @Override
    public void run()
    {
    }
}
```

- За відсутності явного аргументу кожен конструктор приписує унікальне ім'я (Thread-...) об'єкту Thread, яке дає змогу відрізнити потоки.
 - На відміну від попередніх конструкторів, Thread(String threadName) дозволяє задати потоку власну назву.

Методи класу для управління потоками

Method	Description
<code>static Thread currentThread()</code>	Returns the Thread object associated with the thread that calls this method.
<code>String getName()</code>	Returns the name associated with this Thread object.
<code>Thread.State getState()</code>	Returns the state of the thread associated with this Thread object. The state is identified by the Thread.State enum as one of BLOCKED (waiting to acquire a lock, discussed later), NEW (created but not started), RUNNABLE (executing), TERMINATED (the thread has died), TIMED_WAITING (waiting for a specified amount of time to elapse), or WAITING (waiting indefinitely).
<code>void interrupt()</code>	Sets the interrupt status flag in this Thread object. If the associated thread is blocked or is waiting, clear this flag and wake up the thread by throwing an instance of the <code>java.lang.InterruptedException</code> class.
<code>static boolean interrupted()</code>	Returns true when the thread associated with this Thread object has a pending interrupt request. Clears the interrupt status flag.
<code>boolean isAlive()</code>	Returns true to indicate that this Thread object's associated thread is alive and not dead. A thread's life span ranges from just before it is actually started within the <code>start()</code> method to just after it leaves the <code>run()</code> method, at which point it dies.

<code>boolean isDaemon()</code>	Returns true when the thread associated with this Thread object is a <i>daemon thread</i> , a thread that acts as a helper to a <i>user thread</i> (nondaemon thread) and dies automatically when the application's last nondaemon thread dies so the application can exit.
<code>boolean isInterrupted()</code>	Returns true when the thread associated with this Thread object has a pending interrupt request.
<code>void join()</code>	The thread that calls this method on this Thread object waits for the thread associated with this object to die. This method throws <code>InterruptedException</code> when this Thread object's <code>interrupt()</code> method is called.
<code>void join(long millis)</code>	The thread that calls this method on this Thread object waits for the thread associated with this object to die, or until <code>millis</code> milliseconds have elapsed, whichever happens first. This method throws <code>InterruptedException</code> when this Thread object's <code>interrupt()</code> method is called.
<code>void setDaemon(boolean isDaemon)</code>	Marks this Thread object's associated thread as a daemon thread when <code>isDaemon</code> is true. This method throws <code>java.lang.IllegalThreadStateException</code> when the thread has not yet been created and started.
<code>void setName(String threadName)</code>	Assigns <code>threadName</code> 's value to this Thread object as the name of its associated thread.
<code>static void sleep(long time)</code>	Pauses the thread associated with this Thread object for time milliseconds. This method throws <code>InterruptedException</code> when this Thread object's <code>interrupt()</code> method is called while the thread is sleeping.
<code>void start()</code>	Creates and starts this Thread object's associated thread. This method throws <code>IllegalThreadStateException</code> when the thread was previously started and is running or has died.

Демонстрація процесу створення потоків

```
public class CountingThreads
{
    public static void main(String[] args)
    {
        Runnable r = new Runnable()
        {
            @Override
            public void run()
            {
                String name = Thread.currentThread().getName();
                int count = 0;
                while (true)
                    System.out.println(name + ": " + count++);
            }
        };
        Thread thdA = new Thread(r);
        Thread thdB = new Thread(r);
        thdA.start();
        thdB.start();
    }
}
```

- головна нитка, що виконує метод `main()`, спочатку створює екземпляр анонімного класу, який реалізує `Runnable`.
 - Потім вона створює два `Thread`-об'єкти, ініціалізуючи їх `runnable`-об'єктом `r` та викликає метод `start()` класу `Thread` для створення обох потоків.
 - Після завершення цих задач головний потік виходить завершує `main()` та зникає.

Приклад виводу

```
<terminated> CountingThreads [Java Application] C
```

```
Thread-1: 0  
Thread-0: 0  
Thread-1: 1  
Thread-0: 1  
Thread-1: 2  
Thread-1: 3  
Thread-0: 2  
Thread-1: 4  
Thread-1: 5  
Thread-0: 3  
Thread-0: 4  
Thread-0: 5  
Thread-0: 6  
Thread-0: 7  
Thread-0: 8  
Thread-1: 6  
Thread-1: 7  
Thread-1: 8
```

- Кожен з двох запущених потоків виконує `Runnable` метод `run()`:
 - викликає `Thread`-метод `currentThread()` для отримання екземпляру класу `Thread`,
 - використовує даний екземпляр для виклику методу `getName()`, що повертає його назву, ініціалізує лічильник `count = 0`, а потім заходить у нескінченний цикл, який виводить ці дані та інкрементує `count` на кожній ітерації.
- Для того, щоб вийти з циклу після запуску програми, необхідно натиснути `Ctrl+C` у `Windows`.

-
- Якщо комп'ютер має достатньо процесорів або ядер, його операційна система виділяє окремий потік кожному процесору або ядру, тому потоки виконуються паралельно.
 - Якщо достатніх ресурсів немає, різні потоки повинні **чекати своєї черги** для використання спільних (shared) процесорів/ядер.
 - Встановленням черговості потоків займається планувальник (scheduler) операційної системи.
 - Планувальники Windows та багатьох інших операційних систем беруть до уваги *пріоритетність* потоку.
 - Часто вони поєднують
 - пріоритетне планування (*preemptive scheduling* – більш пріоритетні потоки можуть переривати та виконуватись замість менш пріоритетних)
 - циклічну диспетчеризацію (*round robin scheduling* – рівнопріоритетні потоки отримують однакові проміжки часу (*time slices*) для почергового виконання).

Concurrency vs parallelism

- В контексті багатопоточності в англomовній літературі використовують два схожих терміни: одночасне виконання (concurrency) та паралелізм (parallelism).
 - одночасне виконання передбачає стан, у якому кілька потоків існують та прогресують, а паралелізм – лише паралельне виконання.
 - concurrency є більш широким поняттям, яке включає не лише паралельне виконання (паралелізм), але й віртуальний (витісняючий) паралелізм, коли потоки виконуються по чергово протягом деяких періодів часу.

Клас Thread підтримує пріоритизацію за допомогою

- методу `void setPriority(int priority)`, де значення `priority` коливається в межах `[Thread.MIN_PRIORITY...Thread.MAX_PRIORITY]`.
 - Значення за замовчуванням встановлюється в `Thread.NORMAL_PRIORITY`.
 - Використання `setPriority()` може вплинути на переносимість застосунку на різні платформи, оскільки різні планувальники можуть обробляти зміну пріоритету по-різному.
 - Наприклад, один планувальник може відкладати виконання потоків з нижчим пріоритетом до того моменту, поки не виконаються більш пріоритетні потоки.
 - Це називається невизначеною відстрочкою (*indefinite postponement*) або голодуванням (*starvation*) і може значно знизити продуктивність додатку.
 - Інший планувальник може не використовувати цю схему.
- методу `int getPriority()`, який повертає поточний пріоритет.


```

public class CountingThreads
{
    public static void main(String[] args)
    {
        Runnable r = new Runnable()
        {
            @Override
            public void run()
            {
                String name = Thread.currentThread().getName();
                int count = 0;
                while (true)
                {
                    System.out.println(name + ": " + count++);
                    try
                    {
                        Thread.sleep(100);
                    }
                    catch (InterruptedException ie)
                    {
                    }
                }
            }
        };
        Thread thdA = new Thread(r);
        thdA.setName("A");
        Thread thdB = new Thread(r);
        thdB.setName("B");
        thdA.start();
        thdB.start();
    }
}

```

Рефакторинг лістингу дозволяє встановлювати ім'я кожному потоку та «приспати» потік після виводу імені та значення count

■ Вивід:

```

A: 0
B: 0
A: 1
B: 1
B: 2
A: 2
B: 3
A: 3
B: 4
A: 4
B: 5
A: 5
B: 6
A: 6
B: 7
A: 7

```



```

public class JoinDemo
{
    public static void main(String[] args)
    {
        Runnable r = new Runnable()
        {
            @Override
            public void run()
            {
                System.out.println("Worker thread is simulating " +
                                   "work by sleeping for 5 seconds.");

                try
                {
                    Thread.sleep(5000);
                }
                catch (InterruptedException ie)
                {
                }
                System.out.println("Worker thread is dying");
            }
        };

        Thread thd = new Thread(r);
        thd.start();
        System.out.println("Default main thread is doing work.");
        try
        {
            Thread.sleep(2000);
        }
        catch (InterruptedException ie)
        {
        }
        System.out.println("Default main thread has finished its work.");
        System.out.println("Default main thread is waiting for worker thread " +
                           "to die.");

        try
        {
            thd.join();
        }
        catch (InterruptedException ie)
        {
        }
        System.out.println("Main thread is dying");
    }
}

```

Основний потік у випадку тривалих обчислень, завантаження великого файлу або виконання інших довготривалих дій має запустити для них ще один (робочий) потік.

- Після закінчення інших задач основний потік готовий обробити результати робочого потоку і чекає моменту, коли робочий потік закінчить своє виконання та помре.
 - Це можна робити за допомогою циклу while, який періодично буде викликати метод `isAlive()` для робочого потоку, та спати між цими викликами до моменту, коли повернеться значення `true`.
 - Проте наступний лістинг показує більш підходящу альтернативу: метод `join()`.
- Основний потік:
 - запускає робочий потік,
 - виконує деяку роботу,
 - чекає на його завершення шляхом виклику методу `join()` для об'єкту робочого потоку `thd`.

Результати виводу

```
Default main thread is doing work.  
Worker thread is simulating work by sleeping for 5 seconds.  
Default main thread has finished its work.  
Default main thread is waiting for worker thread to die.  
Worker thread is dying  
Main thread is dying
```

ThreadGroup та його проблеми

- Кожен об'єкт класу Thread належить деякому ThreadGroup-об'єкту; отримати його можна за допомогою виклику методу `getThreadGroup()`.
 - Слід ігнорувати групи потоків (thread groups), тому що вони не настільки корисні.
 - Якщо потрібно логічно згрупувати Thread-об'єкти, краще використовувати масив або колекцію.
- Багато методів ThreadGroup є проблемними.
 - Наприклад, `int enumerate(Thread[] threads)` не включатиме всі активні потоки в перелічення, коли аргумент `threads` надто малий для зберігання Thread-об'єктів.
 - Хоч Ви можете подумати, що можна використовувати значення, що повертається від методу `int activeCount()` для визначення правильного розміру масиву, немає гарантії, що він буде достатньо великим через флуктуації, пов'язані зі смертю та народженням програмних потоків.

```
public class ExceptionThread
{
    public static void main(String[] args)
    {
        Runnable r = new Runnable()
        {
            @Override
            public void run()
            {
                int x = 1 / 0; // Line 10
            }
        };
        Thread thd = new Thread(r);
        thd.start();
    }
}
```

- Проте знати про ThreadGroup все-одно потрібно у зв'язку з його внеском в обробку винятків, які викидаються протягом виконання потоку.

```
Exception in thread "Thread-0" java.lang.ArithmeticException: / by zero
    at ExceptionThread$1.run(ExceptionThread.java:10)
    at java.lang.Thread.run(Unknown Source)
```

Дії після викидання винятку та переривання програмного потоку

- JVM переглядає екземпляр `Thread.UncaughtExceptionHandler`, встановлений за допомогою методу `void setUncaughtExceptionHandler (Thread.UncaughtExceptionHandler eh)` класу `Thread`.
 - Коли знайдено обробник, він передається (it passes execution) на виконання методу
 - `void uncaughtException(Thread t, Throwable e)`,
 - `t` ідентифікує `Thread`-об'єкт потоку, який викинув виключення,
 - `e` – викинуте виключення/помилку, наприклад, `java.lang.OutOfMemoryError`.
 - Якщо цей метод викидає `exception/error`, вони ігноруються віртуальною машиною.
- Припускаючи, що `setUncaughtExceptionHandler()` не викликався для встановлення обробника, віртуальна машина передає управління методу `uncaughtException(Thread t, Throwable e)` відповідного `TreadGroup`-об'єкта.
 - Припускаючи, що `ThreadGroup` не розширювався, а його метод для обробки виключення `uncaughtException()` не переозначався, `uncaughtException()` передає управління цьому методу з батьківського класу, якщо наявний батьківський `ThreadGroup`.

Дії після викидання винятку та переривання програмного потоку

- Інакше він перевіряє, чи встановлено обробник неперехопленого виключення за умовчанням (за допомогою методу
 - `static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)` класу `Thread`)
 - Якщо обробник неперехоплених виключень за замовчуванням було встановлено, його метод `uncaughtException()` викликається з тими ж 2 аргументами.
 - Інакше `uncaughtException()` перевіряє свій `Throwable`-аргумент, щоб визначити, чи є він екземпляром `java.lang.ThreadDeath`.
 - Якщо так, нічого особливого не відбувається.
 - Інакше повідомлення винятку з лістингу відображає назву потоку, отриману від його методу `getName()` та `stack backtrace`, використовуючи метод `printStackTrace()`, яким володіє `Throwable`-аргумент.

```

public class ExceptionThread
{
    public static void main(String[] args)
    {
        Runnable r = new Runnable()
        {
            @Override
            public void run()
            {
                int x = 1 / 0;
            }
        };
        Thread thd = new Thread(r);
        Thread.UncaughtExceptionHandler uceh;
        uceh = new Thread.UncaughtExceptionHandler()
        {
            @Override
            public void uncaughtException(Thread t, Throwable e)
            {
                System.out.println("Caught throwable " + e + " for thread "
                                   + t);
            }
        };
        thd.setUncaughtExceptionHandler(uceh);
        uceh = new Thread.UncaughtExceptionHandler()
        {
            @Override
            public void uncaughtException(Thread t, Throwable e)
            {
                System.out.println("Default uncaught exception handler");
                System.out.println("Caught throwable " + e + " for thread "
                                   + t);
            }
        };
        thd.setDefaultUncaughtExceptionHandler(uceh);
        thd.start();
    }
}

```

Методи `setUncaughtExceptionHandler()` та `setDefaultUncaughtExceptionHandler()` класу `Thread`

- Ви не побачите виведення `uncaught exception handler` за умовчанням, оскільки обробник за умовчанням не викликається.
 - Для того, щоб побачити вивід (внизу слайду), закоментуйте рядок `thd.setUncaughtExceptionHandler(uceh)`
 - або
 - `thd.setDefaultUncaughtExceptionHandler(uceh);`

Обережно!

- Клас Thread оголошує кілька застарілих методів, включаючи `stop()` – зупинку потоку, що виконується.
 - Оскільки це `unsafe`-методи, їх вважають застарілими.
 - Не використовуйте ці застарілі методи.
- Також краще уникати методу `static void yield()` – перемикає виконання з поточного потоку на інший – оскільки це впливає на переносимість та знижує продуктивність додатку.
 - На одних платформах `yield()` перемикається на інший потік, що може покращити продуктивність, а на інших – повертається в поточний потік, просто витрачаючи ресурси.