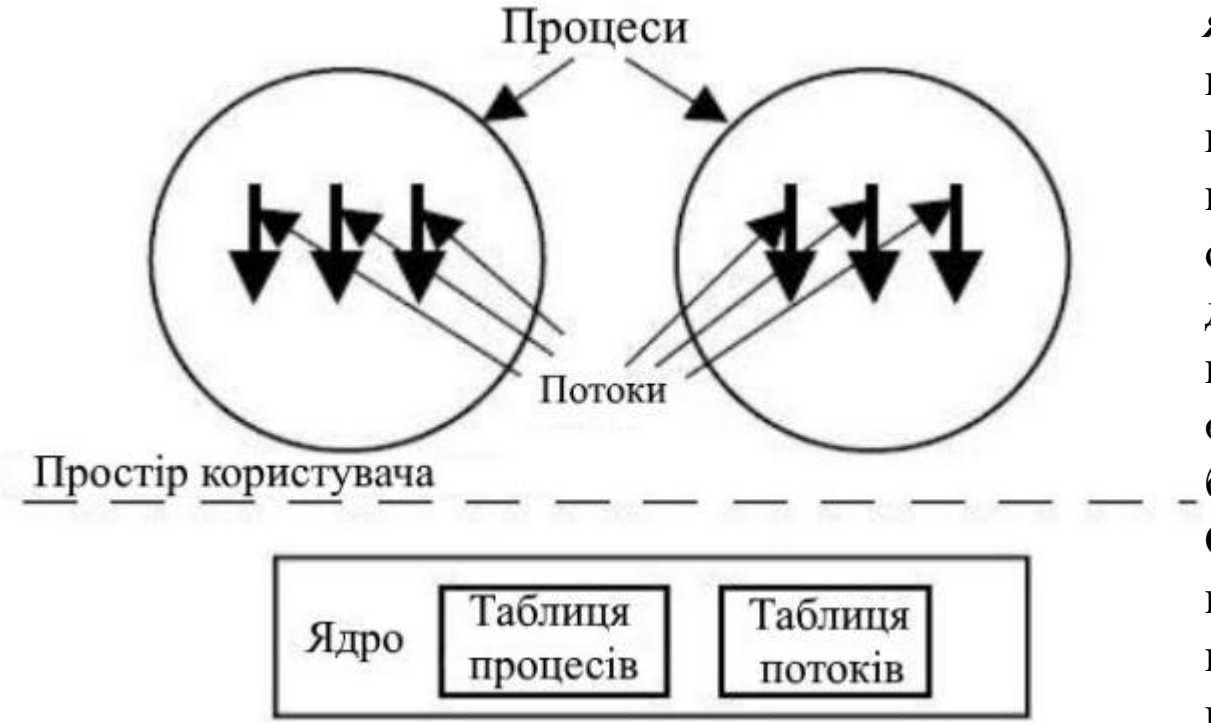
Виділяють дві загальні категорії потоків: ***потоки на рівні користувача (UserLevel Threads – ULT) і потоки на рівні ядра (Kernel-Level Threads – KLT)***. Потоки другого типу називаються потоками, підтримуваними ядром або полегшеними (легковагими) процесами

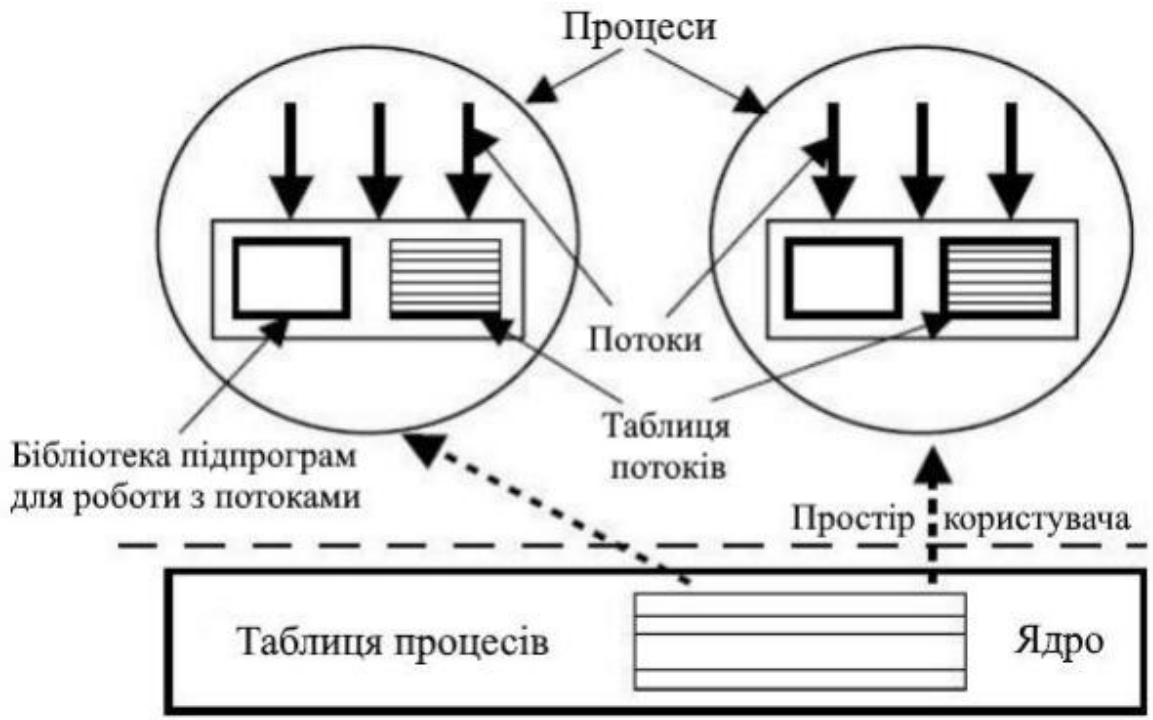


Потоки - в адресному просторі ядра

стратегія використання потоків на рівні ядра. ядро може одночасно здійснювати планування роботи декількох потоків одного і того ж процесу на декількох процесорах. По-друге, при блокуванні одного із потоків процесу ядро може вибрати для виконання інший потік цього ж процесу. Ще однією перевагою такого підходу є те, що самі процедури ядра можуть бути багатопоточними.

Основним недоліком підходу з використанням потоків на рівні ядра, в порівнянні з використанням потоків на рівні користувача, є те, що для передачі управління від одного потоку до іншого в рамках одного і того ж процесу доводитися перемикатися в режим ядра

Виділяють дві загальні категорії потоків: ***потоки на рівні користувача (UserLevel Threads – ULT) і потоки на рівні ядра (Kernel-Level Threads – KLT)***. Потоки другого типу називаються потоками, підтримуваними ядром або полегшеними (легковагими) процесами



стратегія використання потоків на рівні користувача

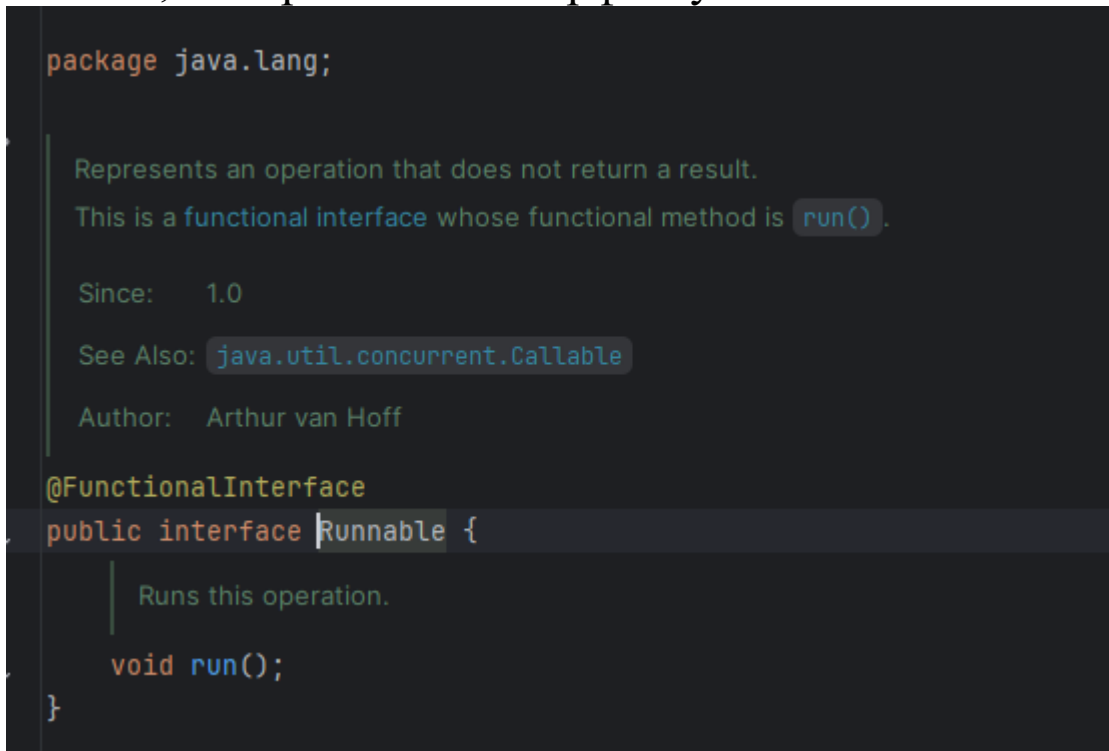
Для перемикання потоків не потрібно переходити в режим ядра, основної структури дані по управлінню потоками знаходяться в адресному просторі одного і того ж процес.

Потоки - в адресному просторі одного і того ж процесу.

Існує два способи визначення послідовності операторів, які повинні виконуватись у потоці:

1) написання класу, що реалізовує інтерфейс java.lang.Runnable
2) написання класу користувача, що успадковує клас java.lang.Thread (останній огортає private Runnable target та реалізує метод run() інтерфейсу Runnable), і перевизначення методу run(), вказавши в ньому логічну послідовність операторів, які повинні виконуватись у потоці (в даному випадку клас користувача можна вважати потоком). Зазначимо, що запуск потоку виконується викликом метода start() класу Thread, а не безпосередньо метода run(), метод виконує додаткові функції і запускає метод run() потоку.

Таким чином, інтерфейс Runnable - є абстракцією над завданням, що виконується у потоці, і дозволяє розмежити виконання завдання від логіки управління потоками. Окрім того, використання інтерфейсу Runnable дозволяє класу користувача бути спадкоємцем іншого класу.



Клас Thread у Java використовується для створення і керування потоками (threads). Потоки дозволяють виконувати кілька завдань одночасно (паралельне або конкурентне програмування). Є два основні способи створення потоку:

Успадкування від Thread

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - " + i);
            try {
                Thread.sleep(500); // Пауза на 500 мс
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.start(); // Запускає потік
        t2.start();
    }
}
```

Реалізація Runnable (рекомендований підхід)

```
class MyRunnable implements Runnable {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(Thread.currentThread().getName() +
" - " + i);
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable());
        Thread t2 = new Thread(new MyRunnable());

        t1.start();
        t2.start();
    }
}
```

Потік (thread) — це потік виконання в програмі. Віртуальна машина Java дозволяє програмі мати декілька потоків виконання, які працюють одночасно.

Клас Thread містить конструктори та Thread.Builder для створення потоків. Запуск потоку планує його виконання через метод run(). Щойно запущений потік виконується одночасно з потоком, який ініціював його запуск.

Потік завершується, якщо його метод run() завершується нормально або якщо він завершується раптово, і відповідний обробник неперехоплених винятків (uncaught exception handler) також завершується нормально або з помилкою. Якщо для потоку не залишилося коду для виконання, він вважається завершеним. Метод join() можна використовувати для очікування завершення потоку.

Кожен потік має унікальний ідентифікатор і ім'я. Ідентифікатор генерується під час створення потоку і не може бути змінений. Ім'я потоку можна вказати при створенні або змінити пізніше.

Потоки підтримують змінні ThreadLocal. Це змінні, які є локальними для потоку, тобто кожен потік має власну копію змінної, значення якої не залежить від значень у інших потоках. Також Thread підтримує змінні InheritableThreadLocal, які є локальними для потоку, але наслідуються від батьківського потоку під час створення нового потоку. Потік також підтримує спеціальну спадкову змінну ThreadLocal для завантажувача контексту класів (thread context-class-loader).

Клас Thread підтримує створення платформних потоків, які зазвичай відображаються 1:1 на потоки ядра, що керуються операційною системою. Платформні потоки зазвичай мають великий стек і інші ресурси, які підтримуються операційною системою. Вони підходять для виконання всіх типів завдань, але можуть бути обмеженим ресурсом.
За замовчуванням платформні потоки отримують автоматично згенероване ім'я.
Платформні потоки можуть бути демонними (daemon) або недемонними (non-daemon). Коли віртуальна машина Java запускається, зазвичай існує хоча б один недемонний потік (зазвичай це потік, який викликає метод main() програми). Процес завершення роботи JVM починається, коли всі запущені недемонні потоки завершуються. Незапущені недемонні потоки не перешкоджають завершенню JVM.

Окрім статусу демона, платформні потоки мають пріоритет потоку і належать до групи потоків.

Демони потоки (daemon threads) – це потоки, які працюють у фоновому режимі та автоматично завершуються, коли всі недемонні потоки (non-daemon threads) у програмі закінчують роботу.

Основні характеристики демонних потоків:
Автоматичне завершення:
Якщо у програмі залишилися тільки демонні потоки, віртуальна машина Java (JVM) завершує виконання.
Недемонні потоки не залежать від демонних і можуть працювати незалежно від них.
Призначення:
Використовуються для виконання фонових задач, які не повинні блокувати завершення програми.
Приклади:
Збірник сміття (Garbage Collector, GC)
Потоки, що обслуговують кеш
Фонові моніторингові задачі
Створення демонного потоку:
Перед запуском (start()) можна зробити потік демонним за допомогою setDaemon(true).
Після запуску start() змінити цей статус не можна!

Віртуальні потоки
Клас Thread також підтримує створення віртуальних потоків. Віртуальні потоки зазвичай є потоками в режимі користувача, які плануються середовищем виконання Java, а не операційною системою. Вони зазвичай споживають мінімальні ресурси, і одна віртуальна машина Java може підтримувати мільйони віртуальних потоків.

Віртуальні потоки підходять для виконання завдань, які більшу частину часу перебувають у стані блокування, часто очікуючи завершення операцій введення/виведення (I/O). Вони не призначені для тривалих обчислювально інтенсивних операцій.

Віртуальні потоки зазвичай використовують невеликий набір платформних потоків, які діють як транспортувальні потоки (carrier threads). Наприклад, під час блокування або виконання операцій введення/виведення транспортувальний потік може бути перепризначений з одного віртуального потоку на інший. Код, що виконується у віртуальному потоці, не має доступу до транспортувального потоку.

Метод currentThread(), який використовується для отримання посилання на поточний потік, завжди повертає об'єкт Thread віртуального потоку.

За замовчуванням віртуальні потоки не мають імені. Якщо ім'я потоку не встановлене, метод getName() повертає порожній рядок.

Віртуальні потоки є демонними потоками, тому вони не заважають процесу завершення роботи JVM. Вони також мають фіксований пріоритет потоку, який не можна змінити.

Успадкування під час створення потоків

Потік (Thread) успадковує початкові значення змінних InheritableThreadLocal (включаючи контекстний завантажувач класів) від батьківського потоку на момент створення дочірнього потоку.

Якщо потрібно створити потік, який не успадковує початкові значення від потоку, що його створює, можна використати конструктор із п’ятьма параметрами.

При використанні Thread.Builder метод inheritInheritableThreadLocals() дозволяє визначити, чи успадковуватимуться початкові значення змінних InheritableThreadLocal.

Успадкування параметрів у платформних потоках

Платформні потоки (Platform threads) успадковують:

Статус демонного потоку (daemon або non-daemon).

Пріоритет потоку.

Групу потоків, якщо вона не була явно вказана (або не визначена менеджером безпеки).

При створенні платформного потоку контекст виклику зберігається, щоб обмежити права нового потоку при виконанні коду, який виконує привілейовану дію. Цей збережений контекст називається "успадкованим AccessControlContext" (Inherited AccessControlContext).

Успадкування параметрів у віртуальних потоках

Віртуальні потоки (Virtual threads) не успадковують контекст виклику.

Вони не мають жодних дозволів при виконанні коду, який виконує привілейовану дію.

Обробка null аргументів

Якщо конструктору або методу цього класу передати значення null, буде викинуто виняток NullPointerException, якщо інше не зазначено явно.

Ми використовуємо статичний метод currentThread() класу java.lang.Thread для отримання посилання на поточний потік, після чого використовуємо нестатичні методи getName() та getId() для отримання унікальних ідентифікаторів потоку. Зазначимо, що безпосередній запуск об’єкта MyRunnable викликом з нього методу run() приводить довиконання завдання у потоці метода main без створення нового потоку. Тому правильно запускати об’єкт Runnable через створення об’єкту Thread з передачею його конструктору об’єкта Runnable.

Основні методи Thread

2.1. Запуск потоку
start() – запускає потік і викликає метод run().
2.2. Сон потоку
sleep(ms) – призупиняє виконання потоку на вказану кількість мілісекунд.
Thread.sleep(1000); // 1 секунда
2.3. Очікування завершення потоку
join() – зупиняє виконання поточного потоку, поки інший не завершиться.
2.4. Зупинка потоку
interrupt() – подає сигнал потоку про зупинку.
2.5. Пріоритет потоку
setPriority(int) – встановлює пріоритет (мінімум 1, максимум 10).
t1.setPriority(Thread.MAX_PRIORITY);
2.6. Демон-потоки
setDaemon(true) – потік автоматично завершується разом із головним потоком.

```
private static void priorityExample() {
      Thread myRunnable = new Thread(new
MyRunnable(), "priority2");
      myRunnable.setPriority(2);

      Thread myRunnable7_1 = new Thread(new
MyRunnable(), "priority7_1");
      myRunnable7_1.setPriority(7);

      Thread myRunnable7_2 = new Thread(new
MyRunnable(), "priority7_2");
      myRunnable7_2.setPriority(7);

      myRunnable.start();
      myRunnable7_1.start();
      myRunnable7_2.start();

   }
}
```

```
priority7_1 RUNNABLE 7
priority7_2 RUNNABLE 7
priority2 RUNNABLE 2
```

setPriority(2) призначає низький пріоритет (за шкалою від 1 до 10).
setPriority(8) призначає вищий пріоритет, що підвищує ймовірність отримання процесорного часу.Планування потоків залежить від ОС і JVM. Навіть із пріоритетами порядок виконання не гарантується.

```
public static final int MIN_PRIORITY = 1;
   /**
    * The default priority that is assigned to a thread.
    */
   public static final int NORM_PRIORITY = 5;

   /**
    * The maximum priority that a thread can have.
    */
   public static final int MAX_PRIORITY = 10;
```

Статичний метод yield() класу Thread викликається тільки для поточного потоку і інформує планувальник потоків про те, що поточний потік готовий відмовитись від використання процесора на момент виклику метода, але бажав би, щоб його запланували якомога скоріше. Планувальник потоків може вільно дотримуватися або ігнорувати цю інформацію і насправді має різну поведінку в залежності від операційної системи. Метод yield() викликається тільки для поточного потоку і при його виклику планувальник потоків призупиняє роботу поточного потоку віддає квант часу іншому потоку, поточний потік переміщується вниз черги потоків з рівним пріоритетом. Насправді, метод yield() не гарантує що поточний потік буде призупинений, і навіть якщо це буде виконано, поточний потік може бути знову обраний для виконання.

Як працює yield()?

Поточний потік добровільно відмовляється від процесорного часу.

Планувальник потоків (JVM + ОС) може:

Перемкнути виконання на інший потік.

Або ігнорувати yield() і продовжити виконання потоку.

Якщо немає інших готових до виконання потоків, потік продовжить виконання.

Приклад використання yield()

```
class MyThread extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {

System.out.println(Thread.currentThread
().getName() + " - " + i);
            Thread.yield(); // Дати
можливість іншим потокам виконатися
        }
    }
}

public class Main {
    public static void main(String[] args) {
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.start();
        t2.start();
    }
}
```

Можливий вивід (не гарантований) Але планувальник може проігнорувати yield(), і один потік виконається повністю перед іншим.

```
Thread-0 - 1
Thread-1 - 1
Thread-0 - 2
Thread-1 - 2
Thread-0 - 3
Thread-1 - 3
Thread-0 - 4
Thread-1 - 4
Thread-0 - 5
Thread-1 - 5
```

Стани потоків

Потік може існувати в одному з наступних станів (рис. 5.2):

NEW - об'єкт Thread створений, але потік ще не запущений. Метод isAlive(), викликаний об'єкта-потоку, повертає false.

RUNNABLE - Потік готовий виконуватися і знаходиться в пулі потоків планувальника, але планувальник потоків не обрав його для виконання. Метод isAlive(), викликаний з об'єкта-потоку, повертає true.

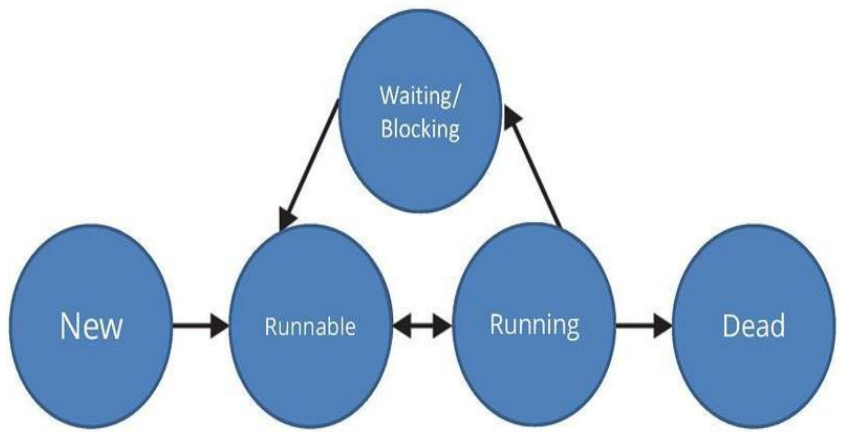
RUNNINIG - Потік обраний планувальником потоків з пулу і виконується. Метод isAlive(), викликаний з об'єкта-потоку, повертає true.

BLOCKED - Робота потоку припинена доки не буде доступний монітор об'єкта, до якого забезпечується доступ з потоку. Метод isAlive(), викликаний з об'єкта-потоку, повертає true.

WAITING - Потік очікує певної дії іншого потоку невизначений час. Метод isAlive(), викликаний з об'єкта-потоку, повертає true.

TIMED_WAITING - Потік очікує певної дії іншого потоку протягом зазначеного часу. Метод isAlive(), викликаний з об'єкта-потоку, повертає true.

TERMINATED (DEAD) - Потік завершений. Метод isAlive(), викликаний з об'єкта-потоку, повертає false.



Переривання потоку
Метод interrupt() перериває виконання потоку.

Якщо потік перериває сам себе, це завжди дозволено. В інших випадках викликається метод checkAccess(), який може призвести до викидання винятку SecurityException, якщо поточний потік не має прав на зміну цільового потоку.

Як interrupt() впливає на потік у різних станах?
Якщо потік заблокований у виклику методів wait(), wait(long), wait(long, int), join(), join(long), join(long, int), sleep(long), sleep(long, int)

Його статус переривання буде очищено.
Буде викинуто виняток InterruptedException.
Якщо потік заблокований у операції введення/виведення (I/O) на InterruptibleChannel
Канал буде закрито.
Потік отримає статус перерваного (interrupted).
Буде викинуто виняток ClosedByInterruptException з пакету java.nio.channels.

Якщо потік заблокований у Selector (java.nio.channels.Selector)
Статус потоку буде встановлений у перерваний.
Потік негайно повернеться з операції вибору (selection operation), можливо, з ненульовим результатом, так само, як якщо б було викликано метод wakeup().
Якщо жодна з вищезазначених умов не виконується
Потік просто отримає статус перерваного (interrupted), але не буде негайно зупинений.
Що станеться, якщо перервати потік, який не є "живим"?
Це не матиме жодного ефекту.
Можливі винятки
SecurityException – якщо поточний потік не має прав на зміну цільового потоку.
Примітка щодо реалізації (JDK Reference Implementation)
Якщо переривається потік, який не є "живим", інформація про запит переривання все одно записується.
Виклики interrupted() або isInterrupted() можуть повідомити про це пізніше.

```
public class InterruptExample {
    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();

        try {
            Thread.sleep(2000); // Головний потік чекає 2 секунди
            thread.interrupt(); // Переривається потік
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Головний потік завершив роботу.");
    }
}
```

```
class MyThread extends Thread {
    public void run() {
        try {
            // Потік виконує деяку роботу
            System.out.println("Потік почав виконання...");
            Thread.sleep(2000); // Потік засинає на 2 секунди
            System.out.println("Потік завершив виконання.");
        } catch (InterruptedException e) {
            System.out.println("Потік був перерваний.");
        }
    }
}

public class WaitNotifyExample {
    private static final Object lock = new Object(); // Об'єкт для синхронізації

    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start(); // Запускаємо потік

        synchronized (lock) {
            try {
                // Використовуємо цикл, щоб чекати, поки потік не завершиться
                while (thread.isAlive()) {
                    System.out.println("Очікування завершення потоку...");
                    lock.wait(); // Чекаємо завершення потоку
                }
                System.out.println("Потік завершився, продовжуємо виконання.");
            } catch (InterruptedException e) {
                System.out.println("Головний потік був перерваний.");
            }
        }

        System.out.println("Головний потік завершив роботу.");
    }
}
```

Створюється та запускається потік MyThread.
Головний потік використовує синхронізований блок з об'єктом lock для того, щоб чекати завершення потоку MyThread.
Цикл перевіряє, чи потік MyThread ще живий за допомогою методу isAlive(). Якщо так, викликається wait() і головний потік чекає.
Коли потік MyThread завершується, викликається метод notifyAll(), що дозволяє головному потоку продовжити виконання.
Ключові моменти:
Метод wait() блокує потік до того моменту, поки не буде сповіщено іншим потоком.
Метод isAlive() перевіряє, чи потік ще працює.
Виклик notifyAll() сигналізує всім потокам, які чекають на об'єкті lock, що вони можуть продовжити виконання.

Питання

1. Що таке потік (thread) у Java та яку роль він відіграє в багатопотоковому програмуванні?
2. У чому полягає різниця між потоками рівня користувача та потоками рівня ядра? Які їхні переваги та недоліки?
3. Як JVM відображає Java-потоки на потоки операційної системи?
4. Яке призначення інтерфейсу Runnable і як він використовується для створення потоку в Java?
5. Які можливості надає інтерфейс Callable порівняно з Runnable?
6. Які основні відмінності між Callable і Runnable щодо повернення результату та обробки винятків?
7. Які ключові методи містить клас java.lang.Thread і для чого вони використовуються (start(), run(), sleep(), join(), interrupt())?
8. Чим відрізняється виклик методу run() від виклику методу start() у класі Thread?
9. Які стани може мати потік (задача) у Java і що означає кожен із них (NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED)?
10. За яких умов потік переходить зі стану RUNNABLE у BLOCKED, WAITING або TIMED_WAITING?