

УПРАВЛІННЯ ПОТОКАМИ

Детальніше: <http://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html>

1. Створення та запуск потоку

Створіть клас MyCalculations:

```
public class MyCalculations implements Runnable {
    private int number;
    public MyCalculations(int number) {
        this.number = number;
    }
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.printf("%s: %d * %d = %d\n", Thread.
                currentThread().getName(), number, i, i * number);
        }
    }
}
```

Додайте до проекту клас Main:

```
public class Main {
    public static void main(String[] args) {
        for (int i = 0; i <= 5; i++) {
            MyCalculations calculations = new MyCalculations(i);
            Thread thread = new Thread(calculations);
            thread.start();
        }
    }
}
```

Створені потоки виконують свою роботу паралельно.

Кожна Java-програма має, принаймні, один потік. При запускові програми JVM виконує цей потік, котрий викликає метод main().

Кожен черговий виклик методу start() створює новий потік. Кількість потоків програми дорівнює кількості викликів методу start().

Виконання програми завершиться тоді, коли усі її потоки (non-daemon) завершать виконання. Якщо початковий потік (той, котрий виконує метод main()) завершив роботу, інші потоки продовжують виконувати роботу доки не завершать її.

Якщо один із потоків використовує інструкцію System.exit(), усі потоки припинять виконання.

Додайте наприкінці методу run() наступний рядок

```
if(Thread.currentThread().getName().equals("Thread-2")) System.exit(0);
```

Результат буде схожим на наступний (щоразу він буде іншим):

```
Thread-0: 0 * 0 = 0
Thread-4: 4 * 0 = 0
Thread-5: 5 * 0 = 0
Thread-3: 3 * 0 = 0
Thread-3: 3 * 1 = 3
Thread-3: 3 * 2 = 6
Thread-3: 3 * 3 = 9
Thread-3: 3 * 4 = 12
Thread-2: 2 * 0 = 0
```

Аналогічного результату (при цьому між варіантами існує принципова відмінність) можна досягти, якщо оголосити клас, що спадкує Thread, реалізувати у ньому @Override методу run(), а у головному потоці створити екземпляр цього класу. Метод start(), викликаний через даний екземпляр, запустить новий потік.

```
public class MyCalculations extends Thread {
    private int number;
    public MyCalculations(int number) {
        this.number = number;
    }
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.printf("%s: %d * %d = %d\n", Thread.
                currentThread().getName(), number, i, i * number);
        }
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            MyCalculations calculations = new MyCalculations(i);
            calculations.start();
        }
    }
}
```

2. Переривання виконання потоку

Java-програма з більш ніж одним потоком завершує виконання, коли усі її потоки

завершати роботу (точніше, коли усі її потоки-не демони завершать роботу або коли один з потоків викличе метод `System.exit()`).

Часом потрібно завершити виконання потоку для завершення роботи програми або завершення завдання, що виконує потік.

Java надає механізм переривання для повідомлення потоку про те, що йому слід завершити роботу. Особливістю такого механізму є те, що потік повинен перевірити, чи його роботу бажають перервати, чи ні. Потік сам вирішує, чи реагувати йому на запит завершення роботи, чи не проігнорувати його і продовжити виконання.

Розглянемо приклад:

```
public class MyThread extends Thread {
    @Override
    public void run() {
        while (true) {
            if (isInterrupted()) {
                System.out.printf("Виконання потоку - перервано");
                return;
            }
        }
    }
}
```

Метод `run()` класу не виконує ніякої роботи окрім того, що перевіряє, чи отримано запит на припинення роботи потоку.

Надішлемо йому такий запит у класі `Main`:

```
public class Main {
    public static void main(String[] args) {
        Thread task = new MyThread();
        task.start();
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        task.interrupt();
    }
}
```

Результат: через 5 с у консоль потік виводить повідомлення

`/usr/local/java/jdk1.8.0_66/bin/java`

Виконання потоку - перервано

Додамо деякого функціоналу до потоку:

```

public class MyThread extends Thread {
    @Override
    public void run() {
        int delay = 0;
        while (true) {
            try {
                Thread.sleep(500);
                delay += 500;
                System.out.println("delay: " + delay);
            } catch (InterruptedException e) {
                //e.printStackTrace();
                // на цей момент interrupt status == false
                System.out.println("Sleep потоку було перервано іншим потоком");
                return;
            }
            if (isInterrupted()) {
                System.out.println("Виконання потоку - перервано");
                return;
            }
        }
    }
}

```

Результат:

```

delay: 500
delay: 1000
delay: 1500
delay: 2000
delay: 2500
delay: 3000
delay: 3500
delay: 4000
delay: 4500
Sleep потоку було перервано іншим потоком

```

Переривання Sleep відбувається негайно щойно надсилається іншим потоком. Потік надсилає переривання шляхом виклику [interrupt](#) над об'єктом Thread потоку, дія якого повинна бути перервана. Для того, щоб механізм переривання працював коректно, потік повинне підтримувати власне переривання. Потік може сам перервати власне виконання.

<https://docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html>

Яким чином потік підтримує власне переривання? Це залежить від того, що він виконує у даний момент. Якщо потік часто викликає методи, здатні генерувати InterruptedException, він повертається з методу run після обробки виняткової ситуації.

Багато методів, що генерують InterruptedException, таких, як sleep, сконструйовані таким чином, що переривають власне виконання негайно, якщо надходить запит на переривання.

3. Interrupt Status Flag

Механізм переривань імплементовано шляхом використання внутрішнього флагу, котрий називають `interrupt status`. Виклик `Thread.interrupt` встановлює цей флаг. Коли потік перевіряє переривання, викликаючи статичний метод `Thread.interrupted`, флаг переривання скидається. Нестатичний метод `isInterrupted`, котрий використовують для перевірки статусу одного потоку іншим, не змінює значення статусного флагу.

Довільний метод, генеруючи `InterruptedException`, скидає `interrupt`. Проте, довільний потік, що викликає переривання, може одразу встановити його знову.

<https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#interrupted-->

`interrupted()`

`public static boolean interrupted()`

Перевіряє, чи поточний потік було перервано. Скидає флаг *interrupted status*. Іншими словам, якщо метод буде викликано двічі, другий виклик поверне `false`, якщо виконання даного потоку не було перервано ще раз після першого виклику `interrupted()`.

Повертає:

`true`, якщо даний потік було перервано, інакше — `false`.

`isInterrupted()`

`public boolean isInterrupted()`

Перевіряє, чи поточний потік було перервано. Не змінює значення флагу *interrupted status*.

Повертає:

`true`, якщо даний потік було перервано, інакше — `false`.

4. Контролювання переривання потоків

Розглянемо приклад з книги *Java 7 Concurrency Cookbook* (Javier Fernández González, Packt Publishing, 2012, С. 15 — 18).

```
import java.util.concurrent.TimeUnit;
public class Main {
    public static void main(String[] args) {
        FileSearch searcher = new FileSearch("/home/user/", "Retrofit.pdf");
        Thread thread = new Thread(searcher);
        thread.start();
        try {
            TimeUnit.SECONDS.sleep(30);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        thread.interrupt();
    }
}
```

```

import java.io.File;
public class FileSearch implements Runnable{
    private String initPath;
    private String fileName;
    public FileSearch(String initPath, String fileName) {
        this.initPath = initPath;
        this.fileName = fileName;
    }
    @Override
    public void run() {
        File file = new File(initPath);
        if (file.isDirectory()) {
            try {
                directoryProcess(file);
            } catch (InterruptedException e) {
                System.out.printf("%s: The search has been " +
                    "interrupted", Thread.currentThread().getName());
            }
        }
    }
    private void directoryProcess(File file) throws
        InterruptedException {
        File list[] = file.listFiles();
        if (list != null) {
            for (int i = 0; i < list.length; i++) {
                if (list[i].isDirectory()) {
                    directoryProcess(list[i]);
                } else {
                    fileProcess(list[i]);
                }
            }
        }
        if (Thread.interrupted()) {
            throw new InterruptedException();
        }
    }
    private void fileProcess(File file) throws InterruptedException
    {
        if (file.getName().equals(fileName)) {
            System.out.printf("%s : %s\n", Thread.currentThread().
                getName() ,file.getAbsolutePath());
        }
        if (Thread.interrupted()) {
            throw new InterruptedException();
        }
    }
}

```

Результат:

Thread-0 : /home/user/Retrofit.pdf

Thread-0 : /home/user/android-sdk-linux/Retrofit.pdf

Thread-0: The search has been interrupted

Метод `main` класу `Main` запускає потік, оголошений у класі `FileSearch`. При цьому відбувається рекурсивний пошук файлу з потрібним іменем. Початкову адресу та ім'я файлу задають за допомогою конструктора `FileSearch`.

Методи `directoryProcess` та `fileProcess` перевіряють, чи не відбулося переривання роботи потоку за допомогою методу `interrupted()`. Якщо флаг переривання встановлено, метод `interrupted()` скидає його і програма генерує виняткову ситуацію `InterruptedException`, яка обробляється методом `run()`.

5. Засинання та відновлення роботи потоків

<https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html#sleep-long->

Для призупинення роботи потоку на заданий період часу використовують метод `Thread.Sleep()` або `sleep()` елемента `TimeUnit`.

sleep

`public static void sleep(long millis) throws InterruptedException`

Змушує поточний потік “заснути” (тимчасово припинити виконання) на задану кількість мілісекунд. Точність залежить від системних таймерів та планувальників задач.

Параметри:

`millis` — період часу у мілісекундах.

Виняткові ситуації:

[IllegalArgumentOutOfRangeException](#) — у випадкові, якщо параметр має від'ємне значення;

[InterruptedException](#) — якщо довільний потік перервав виконання даного потоку. Флаг *interrupted status* даного потоку скидається у випадкові появи цієї виняткової ситуації.

sleep

`public static void sleep(long millis, int nanos) throws InterruptedException`

Змушує поточний потік “заснути” (тимчасово припинити виконання) на задану кількість мілісекунд + задану кількість наносекунд. Точність залежить від системних таймерів та планувальників задач.

Параметри: The thread does not lose ownership of any monitors.

Parameters:

`millis` — час у мс;

`nanos` — 0-999999 додаткові нс часу засинання.

Виняткові ситуації:

[IllegalArgumentOutOfRangeException](#) — у випадкові, якщо параметр має від'ємне значення або кількість нс не належить діапазону 0-999999;

[InterruptedException](#) — якщо довільний потік перервав виконання даного потоку. Флаг *interrupted status* даного потоку скидається у випадкові появи цієї виняткової ситуації.

Перелічуваний тип `TimeUnit`

<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TimeUnit.html>

`TimeUnit` представляє інтервали часу із заданим рівнем гранулярності. `TimeUnit` не виконує

жодних дій з інформацією, а тільки допомагає організувати її. TimeUnit використовують для інформування методів про те, як інтерпретувати їхні часові параметри. Константи — наступні:

DAYS — одиниця часу, котра представляє двадцять чотири години;
HOURS — одиниця часу, котра представляє шістдесят хвилин;
MICROSECONDS — одиниця часу, котра представляє одну тисячну мілісекунди;
MILLISECONDS — одиниця часу, котра представляє одну тисячну секунди;
MINUTES — одиниця часу, котра представляє шістдесят хвилин;
NANOSECONDS — одиниця часу, котра представляє одну тисячну мікросекунди;
SECONDS — одиниця часу, котра представляє одну секунду.

Зверніть увагу на те, що немає гарантії того, що конкретна імплементація затримки буде здатною відчувати перебіг часу на рівні гранулярності, заданому TimeUnit.

6. Очікування завершення роботи потоку (join)

Метод join() дозволяє потокові очікувати на завершення роботи іншого потоку. Якщо t — об'єкт Thread, потік котрого у даний момент виконується, у результаті

```
t.join();
```

даний потік повинен призупинити виконання доти, доки потік t завершить свою роботу. Перевантаження join дозволяє вказати період очікування. Проте, як і sleep, join залежить від синхронізації операційної системи, тому не слід очікувати, що join буде очікувати саме вказаний період. Як і sleep, join відповідає на переривання з InterruptedException.

Приклад з <https://docs.oracle.com/javase/tutorial/essential/concurrency/simple.html>

```
public class SimpleThread {
    // Display a message, preceded by
    // the name of the current thread
    static void threadMessage(String message) {
        String threadName =
            Thread.currentThread().getName();
        System.out.format("%s: %s%n",
            threadName,
            message);
    }
    private static class MessageLoop
        implements Runnable {
        public void run() {
            String importantInfo[] = {
                "Рік починається у січні",
                "за січнем йде лютий",
                "за лютим - березень",
                "за березнем - квітень"
            };
            try {
                for (int i = 0;
```



```

        i < importantInfo.length;
        i++) {
            // Pause for 4 seconds
            Thread.sleep(4000);
            // Print a message
            threadMessage(importantInfo[i]);
        }
    } catch (InterruptedException e) {
        threadMessage("Мене - зупинили!");
    }
}
}

public static void main(String args[])
    throws InterruptedException {
    // Delay, in milliseconds before
    // we interrupt MessageLoop
    // thread (default one hour).
    long patience = 1000 * 60 * 60;
    // If command line argument
    // present, gives patience
    // in seconds.
    if (args.length > 0) {
        try {
            patience = Long.parseLong(args[0]) * 1000;
        } catch (NumberFormatException e) {
            System.err.println("Argument must be an integer.");
            System.exit(1);
        }
    }
    threadMessage("Запускаємо потік MessageLoop");
    long startTime = System.currentTimeMillis();
    Thread t = new Thread(new MessageLoop());
    t.start();
    threadMessage("Очікуємо на завершення потоку MessageLoop");
    // loop until MessageLoop
    // thread exits
    while (t.isAlive()) {
        threadMessage("А снігу все нема...");
        // Wait maximum of 1 second
        // for MessageLoop thread
        // to finish.
        t.join(1000);
        if (((System.currentTimeMillis() - startTime) > patience)
            && t.isAlive()) {
            threadMessage("Змучилися чекати!");
            t.interrupt();
            // Shouldn't be long now
            // -- wait indefinitely
            t.join();
        }
    }
}

```

```
    }  
    threadMessage("Нарешті!");  
  }  
}
```

Результат виконання:

```
main: Запускаємо потік MessageLoop  
main: Очікуємо на завершення потоку MessageLoop  
main: А снігу все нема...  
main: А снігу все нема...  
main: А снігу все нема...  
main: А снігу все нема...  
Thread-0: Рік починається у січні  
main: А снігу все нема...  
main: А снігу все нема...  
main: А снігу все нема...  
main: А снігу все нема...  
Thread-0: за січнем йде лютий  
main: А снігу все нема...  
main: А снігу все нема...  
main: А снігу все нема...  
main: А снігу все нема...  
Thread-0: за лютим - березень  
main: А снігу все нема...  
main: А снігу все нема...  
main: А снігу все нема...  
main: А снігу все нема...  
Thread-0: за березнем - квітень  
main: Нарешті!
```

7. Синхронізація

Потоки взаємодіють, надаючи доступ до полів та посилань на об'єкти, на які вказують поля. Така форма взаємодії є ефективною, проте уможливорює 2 типи помилок: інтерференцію потоків та помилки черговості.

Інструмент, який запобігає появі таких помилок, називається синхронізацією. Разом з тим, синхронізація може викликати конкуренцію потоків, котра виникає тоді, коли два або більше потоків намагаються отримати доступ до деякого ресурсу одночасно і у результаті програма виконується повільно або, навіть, взагалі припиняє виконання.

8. Інтерференція потоків

<https://docs.oracle.com/javase/tutorial/essential/concurrency/interfere.html>

Розглянемо деякий клас

```
class Counter {  
    private int c = 0;  
  
    public void increment() {  
        c++;  
    }  
  
    public void decrement() {  
        c--;  
    }  
  
    public int value() {  
        return c;  
    }  
}
```

Інтерференція виникає тоді, коли дві операції, що виконуються у різних потоках, діють над одними і тими ж даними. Наприклад, одночасно виконують методи `increment()` та `decrement()`. Така ситуація була б неможливою, якщо б ці методи були неподільними, проте вони складаються із декількох кроків.

9. Помилки консистентності пам'яті

Помилки консистентності (узгодженості) трапляються тоді, коли різні потоки не мають чіткого уявлення про те, чим повинні бути одні і ті ж дані.

Ключ для уникнення помилок консистентності — розуміння зв'язку “happens-before” (“трапилося до”).

Приклад: використання `start()` та `join()`.

10. Синхронізовані методи

<https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html>

Java надає 2 ідіоми синхронізації: синхронізовані методи та синхронізовані блоки. Для того, щоб метод був синхронізованим, слід додати ключове слово `synchronized` до його оголошення:

```
public class SynchronizedCounter {  
    private int c = 0;  
  
    public synchronized void increment() {  
        c++;  
    }  
  
    public synchronized void decrement() {
```

```
        c--;  
    }  
  
    public synchronized int value() {  
        return c;  
    }  
}
```

Якщо count — екземпляр SynchronizedCounter, то результатом оголошення методів синхронізованими є два ефекти:

- стає неможливою інтерференція потоків. Якщо один потік виконує синхронізований метод для об'єкта, інші потоки, що викликають синхронізовані методи для того ж об'єкта, повинні дочекатися завершення виконання методу потоком, котрий отримав доступ до даних швидше;

- якщо синхронізований метод існує, він автоматично встановлює зв'язок “happens-before” з будь-яким іншим послідовним викликом синхронізованого методу для цього самого об'єкта. Дане гарантує, що зміни стану об'єкта будуть видимими для інших потоків.

Конструктор класу не може бути синхронізованим. Синхронізація конструктора не має сенсу, оскільки під час створення екземпляра класу тільки один потік може мати доступ до конструктора.

Конструюючи об'єкт, доступний різним класам, слід бути уважним і не поширювати посилання на цей об'єкт раніше, ніж він стане доступним.

Final-поля, котрі не можуть бути змінені після синхронізації, доступні після створення екземпляру не синхронізованим методам.

11. Синхронізовані оператори

Синхронізація у Java побудована навколо внутрішньої сутності, відомої як “внутрішнє блокування” або блокування за допомогою “моніторів”.

Монітори запобігають доступу до об'єкта, до якого отримує доступ синхронізований метод.

```
public void addName(String name) {  
    synchronized(this) {  
        lastName = name;  
        nameCount++;  
    }  
    nameList.add(name);  
}
```

У даному випадкові оператор addName потребує синхронізації змін до lastName та nameCount.

Синхронізовані блоки зручні для покращення конкурування потоків у випадкові добре гранульованих даних:

```
public class MsLunch {
```

```

private long c1 = 0;
private long c2 = 0;
private Object lock1 = new Object();
private Object lock2 = new Object();

public void inc1() {
    synchronized(lock1) {
        c1++;
    }
}

public void inc2() {
    synchronized(lock2) {
        c2++;
    }
}
}

```

12. Атомарні дії

У програмуванні атомарному дія є тією, яка ефективно відбувається раптово. Атомарна дія не може зупинитися посередині. Вона або відбувається повністю, або не відбувається узагалі. Жодні побічні дії атомарної операції не будуть видимі, поки вона не завершиться.

Приклад, операції читання-запису змінних-посилань та більшості примітивних типів — атомарні. Читання та запис усіх змінних, оголошених як `volatile` — атомарні.

А оператор `++` не є атомарним.

13. Живучість

Здатність паралельної програми бути виконаною вчасно, називається живучістю.

14. Тупик (Deadlock)

```

public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s"
                + " has bowed to me!\n",
                this.name, bower.getName());

```

```

        bower.bowBack(this);
    }
    public synchronized void bowBack(Friend bower) {
        System.out.format("%s: %s"
            + " has bowed back to me!\n",
            this.name, bower.getName());
    }
}

public static void main(String[] args) {
    final Friend alphonse =
        new Friend("Alphonse");
    final Friend gaston =
        new Friend("Gaston");
    new Thread(new Runnable() {
        public void run() { alphonse.bow(gaston); }
    }).start();
    new Thread(new Runnable() {
        public void run() { gaston.bow(alphonse); }
    }).start();
}
}

```

15. Створення та запуск потоків-демонів

Java володіє спеціальним типом потоків — демонів. Такі потоки мають дуже низький пріоритет і зазвичай виконуються у випадкові, якщо на даний час не виконуються інші потоки процесу. Коли демон залишається єдиним потоком програми, котрий виконується, віртуальна машина припиняє роботу такого потоку.

Розглянемо приклад з книги *Java 7 Concurrency Cookbook* (Javier Fernández González, Packt Publishing, 2012, С. 24 — 27).

```

import java.util.ArrayDeque;
import java.util.Date;
import java.util.Deque;
import java.util.concurrent.TimeUnit;
public class Chap1 {
    private static class Event {
        public Date getDate() {
            return date;
        }
        public void setDate(Date date) {
            this.date = date;
        }
        public String getEvent() {
            return event;
        }
    }
}

```

```

    }
    public void setEvent(String event) {
        this.event = event;
    }
    private Date date;
    private String event;
}

public static class WriterTask implements Runnable {
    private Deque<Event> deque;
    public WriterTask(Deque<Event> deque) {
        this.deque = deque;
    }
    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            Event event = new Event();
            event.setDate(new Date());
            event.setEvent(String.format("The thread %s has generated an event",
                Thread.currentThread().getId()));
            synchronized(this) {
                deque.addFirst(event);
                System.out.println("deque size: " + deque.size() + " thread: " +
                    Thread.currentThread().getId());
            }
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public static class CleanerTask extends Thread {
    private Deque<Event> deque;
    public CleanerTask(Deque<Event> deque) {
        this.deque = deque;
        setDaemon(true);
    }
    @Override
    public void run() {
        while (true) {
            Date date = new Date();
            clean(date);
        }
    }
    private void clean(Date date) {
        long difference;
        boolean delete;
        if (deque.isEmpty()) {
            return;
        }
    }
}

```

```

    }
    delete = false;
    do {
        Event e = deque.getLast();
        difference = date.getTime() - e.getDate().getTime();
        if (difference > 10000) {
            System.out.printf("Cleaner: %s\n", e.getEvent());
            deque.removeLast();
            delete = true;
        }
    } while (difference > 10000);
    if (delete) {
        System.out.printf("Cleaner: Size of the deque: %d\n", deque.size());
    }
}
}

public static void main(String[] args) {
    Deque<Event> deque = new ArrayDeque<>();
    WriterTask writerTask = new WriterTask(deque);
    for (int i = 0; i < 3; i++) {
        Thread thread = new Thread(writerTask);
        thread.start();
    }
    CleanerTask cleanerTask = new CleanerTask(deque);
    cleanerTask.start();
}
}

```

Результат виконання:

```

deque size: 1 thread: 10
deque size: 2 thread: 11
deque size: 3 thread: 12
deque size: 4 thread: 10
deque size: 5 thread: 11
deque size: 6 thread: 12
deque size: 7 thread: 10
deque size: 8 thread: 12
deque size: 9 thread: 11
deque size: 10 thread: 10
deque size: 11 thread: 12
deque size: 12 thread: 11
deque size: 13 thread: 10
deque size: 14 thread: 12
deque size: 15 thread: 11
deque size: 16 thread: 10
deque size: 17 thread: 12
deque size: 18 thread: 11
deque size: 19 thread: 12
deque size: 20 thread: 10

```


deque size: 21 thread: 11
deque size: 22 thread: 10
deque size: 23 thread: 12
deque size: 24 thread: 11
deque size: 25 thread: 12
deque size: 26 thread: 10
deque size: 27 thread: 11
deque size: 28 thread: 12
deque size: 29 thread: 10
deque size: 30 thread: 11
Cleaner: The thread 10 has generated an event
Cleaner: The thread 11 has generated an event
Cleaner: Size of the deque: 28
Cleaner: The thread 12 has generated an event
Cleaner: Size of the deque: 27
deque size: 28 thread: 10
deque size: 29 thread: 11
deque size: 30 thread: 12
Cleaner: The thread 10 has generated an event
Cleaner: The thread 11 has generated an event
Cleaner: Size of the deque: 28
Cleaner: The thread 12 has generated an event
Cleaner: Size of the deque: 27
deque size: 28 thread: 10
deque size: 29 thread: 11
deque size: 30 thread: 12
(результат наведено не повністю)

16. Обробка неконтрольованих виняткових ситуації у потоках

У Java існують два типи виняткових ситуації:

checked
unchecked

```
public class Chap1 {  
    public static class ExceptionHandler implements Thread.UncaughtExceptionHandler {  
        @Override  
        public void uncaughtException(Thread t, Throwable e) {  
            System.out.printf("An exception has been captured.\n");  
            System.out.printf("Thread: %s\n", t.getId());  
            System.out.printf("Exception: %s: %s\n", e.getClass().getName(), e.getMessage());  
            System.out.printf("Stack Trace: \n");  
            e.printStackTrace(System.out);  
            System.out.printf("Thread status: %s\n", t.getState());  
        }  
    }  
    public static class Task implements Runnable {
```

```

@Override
public void run() {
    int n = Integer.parseInt("TT");
}
}
public static void main(String[] args) {
    Thread thread = new Thread(new Task());
    thread.setUncaughtExceptionHandler(new ExceptionHandler());
    thread.start();
}
}

```

Результат:

An exception has been captured.

Thread: 10

Exception: java.lang.NumberFormatException: For input string: "TT"

Stack Trace:

java.lang.NumberFormatException: For input string: "TT"

at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)

at java.lang.Integer.parseInt(Integer.java:580)

at java.lang.Integer.parseInt(Integer.java:615)

at Chap1\$Task.run(Chap1.java:20)

at java.lang.Thread.run(Thread.java:745)

Thread status: RUNNABLE

Якщо закоментувати `thread.setUncaughtExceptionHandler(new ExceptionHandler())`, результат буде наступним:

Exception in thread "Thread-0" java.lang.NumberFormatException: For input string: "TT"

at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)

at java.lang.Integer.parseInt(Integer.java:580)

at java.lang.Integer.parseInt(Integer.java:615)

at Chap1\$Task.run(Chap1.java:20)

at java.lang.Thread.run(Thread.java:745)

17. Використання локальних змінних у потоках

```

import java.util.Date;
import java.util.concurrent.TimeUnit;
public class Chap1 {
    public static class UnsafeTask implements Runnable {
        private Date startDate;
        @Override
        public void run() {
            startDate = new Date();

```

```

        System.out.printf("Starting Thread: %s : %s\n",
            Thread.currentThread().getId(), startDate);
    try {
        TimeUnit.SECONDS.sleep((int) Math rint(Math.random() * 10));
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.printf("Thread finished: %s : %s\n",
        Thread.currentThread().getId(), startDate);
}
}

public static class SafeTask implements Runnable {
    private static ThreadLocal<Date> startDate = new ThreadLocal<Date>(){
        protected Date initialValue() {
            return new Date();
        }
    };
    @Override
    public void run() {
        System.out.printf("Starting Thread: %s : %s\n",
            Thread.currentThread().getId(), startDate.get());
        try {
            TimeUnit.SECONDS.sleep((int) Math rint(Math.random() * 10));
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.printf("Thread finished: %s : %s\n",
            Thread.currentThread().getId(), startDate.get());
    }
}

public static void main(String[] args) {
//    UnsafeTask unsafeTask = new UnsafeTask();
//    for(int i = 0; i < 3; i++) {
//        Thread thread = new Thread(unsafeTask);
//        thread.start();
//
//        try {
//            TimeUnit.SECONDS.sleep(2);
//        } catch (InterruptedException e) {
//            e.printStackTrace();
//        }
//    }
    SafeTask safeTask = new SafeTask();
    for(int i = 0; i < 3; i++) {
        Thread thread = new Thread(safeTask);
        thread.start();
        try {
            TimeUnit.SECONDS.sleep(2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

```
}  
}  
}  
}
```

Результат виконання із безпечним доступом до змінних:

```
Starting Thread: 10 : Fri Apr 15 07:17:52 EEST 2016  
Starting Thread: 11 : Fri Apr 15 07:17:54 EEST 2016  
Starting Thread: 12 : Fri Apr 15 07:17:56 EEST 2016  
Thread finished: 12 : Fri Apr 15 07:17:56 EEST 2016  
Thread finished: 10 : Fri Apr 15 07:17:52 EEST 2016  
Thread finished: 11 : Fri Apr 15 07:17:54 EEST 2016
```

Як видно, кожен потік завершує своє виконання із тим значенням змінної, з яким він почав виконання.

Змінимо код з безпечного на небезпечний, закоментувавши і розкоментувавши відповідні частини.

Результат:

```
Starting Thread: 10 : Fri Apr 15 07:20:04 EEST 2016  
Starting Thread: 11 : Fri Apr 15 07:20:06 EEST 2016  
Thread finished: 10 : Fri Apr 15 07:20:06 EEST 2016  
Starting Thread: 12 : Fri Apr 15 07:20:08 EEST 2016  
Thread finished: 11 : Fri Apr 15 07:20:08 EEST 2016  
Thread finished: 12 : Fri Apr 15 07:20:08 EEST 2016
```

Легко помітити, що потік завершує виконання із тим значенням дати, з яким почав виконання останній на даний момент потік.

18. Групування потоків

```
import java.util.Date;  
import java.util.Random;  
import java.util.concurrent.TimeUnit;  
public class Chap1 {  
    public static class SearchTask implements Runnable {  
        private Result result;  
        public SearchTask(Result result) {  
            this.result = result;  
        }  
        @Override  
        public void run() {  
            String name = Thread.currentThread().getName();  
            System.out.printf("Thread %s: Start\n", name);  
            try {  
                doTask();  
                result.setName(name);  
            }  
        }  
    }  
}
```

```

        } catch (InterruptedException e) {
            System.out.printf("Thread %s: Interrupted.\n", name);
            return;
        }
        System.out.printf("Thread %s: End\n", name);
    }
    private void doTask() throws InterruptedException {
        Random random = new Random(new Date().getTime());
        int value = (int)(random.nextDouble() * 10);
        System.out.printf("Thread %s : %d\n", Thread.currentThread().getName(), value);
        TimeUnit.SECONDS.sleep(value);
    }
}
private static class Result {
    private String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
public static void main(String[] args) {
    ThreadGroup threadGroup = new ThreadGroup("Searcher");
    Result result = new Result();
    SearchTask searchTask = new SearchTask(result);
    for (int i = 0; i < 5; i++) {
        Thread thread = new Thread(threadGroup, searchTask);
        thread.start();
        try {
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    System.out.printf("Number of threads: %d\n", threadGroup.activeCount());
    System.out.printf("Information about the ThreadGroup\n");
    threadGroup.list();
    Thread[] threads = new Thread[threadGroup.activeCount()];
    threadGroup.enumerate(threads);
    for (int i = 0; i < threadGroup.activeCount(); i++) {
        System.out.printf("Thread %s: %s\n", threads[i].getName(), threads[i].getState());
    }
    waitFinish(threadGroup);
    threadGroup.interrupt();
}
private static void waitFinish(ThreadGroup threadGroup) {
    while (threadGroup.activeCount() > 0) {
        try {
            TimeUnit.SECONDS.sleep(1);

```

```

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

Результат виконання:

```

Thread Thread-0: Start
Thread Thread-0 : 1
Thread Thread-1: Start
Thread Thread-1 : 4
Thread Thread-0: End
Thread Thread-2: Start
Thread Thread-2 : 3
Thread Thread-3: Start
Thread Thread-3 : 5
Thread Thread-4: Start
Thread Thread-4 : 5
Thread Thread-1: End
Number of threads: 3
Information about the ThreadGroup
Thread Thread-2: End
java.lang.ThreadGroup[name=Searcher,maxpri=10]
  Thread[Thread-2,5,Searcher]
  Thread[Thread-3,5,Searcher]
  Thread[Thread-4,5,Searcher]
Thread Thread-3: TIMED_WAITING
Thread Thread-4: TIMED_WAITING
Thread Thread-4: Interrupted.
Thread Thread-3: Interrupted.

```

19. Обробка неконтрольованих виняткових ситуації у групах потоків

```

import java.util.Random;
public class Chap1 {
    public static class MyThreadGroup extends ThreadGroup {
        public MyThreadGroup(String name) {
            super(name);
        }
        @Override
        public void uncaughtException(Thread t, Throwable e) {
            System.out.printf("The thread %s has thrown an exception\n", t.getId());
            e.printStackTrace(System.out);
            System.out.printf("Terminating the rest of threads\n");
            interrupt();
        }
    }
}

```

```

    }
}
public static class Task implements Runnable {
    @Override
    public void run() {
        int result;
        Random random = new Random(Thread.currentThread().getId());
        while (true) {
            result = 1000 / ((int) (random.nextDouble() * 1000));
            System.out.printf("%s : %d\n", Thread.currentThread().getId(), result);
            if (Thread.currentThread().isInterrupted()) {
                System.out.printf("%d : interrupted\n", Thread.currentThread().getId());
                return;
            }
        }
    }
}

public static void main(String[] args) {
    MyThreadGroup threadGroup = new MyThreadGroup("MyThreadGroup");
    Task task = new Task();
    for (int i = 0; i < 2; i++) {
        Thread t = new Thread(threadGroup, task);
        t.start();
    }
}
}

```

Приклад результату:

```

10 : 1
11 : 1
11 : 2
11 : 1
11 : 38
10 : 3
10 : 16
11 : 5
...
...
...
The thread 11 has thrown an exception
...
...
...
java.lang.ArithmeticException: / by zero
    at Chap1$Task.run(Chap1.java:21)
    at java.lang.Thread.run(Thread.java:745)
Terminating the rest of threads
10 : interrupted

```

20. Використання фабрики для створення потоків

```
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.concurrent.ThreadFactory;
import java.util.concurrent.TimeUnit;

public class Chap1 {

    public static class MyThreadFactory implements ThreadFactory {

        private int counter;
        private String name;
        private List<String> stats;

        public MyThreadFactory(String name) {
            this.counter = 0;
            this.name = name;
            stats = new ArrayList<>();
        }

        @Override
        public Thread newThread(Runnable r) {
            Thread t = new Thread(r, name + "-Thread_" + counter);
            counter++;
            stats.add(String.format("Created thread %d with name %s on %s",
                t.getId(), t.getName(), new Date()));
            return t;
        }

        public String getStats() {
            StringBuffer buffer = new StringBuffer();
            for (String stat : stats) {
                buffer.append(stat);
                buffer.append("\n");
            }
            return buffer.toString();
        }
    }

    public static class Task implements Runnable {

        @Override
        public void run() {
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```



```
    }  
}  
  
public static void main(String[] args) {  
    MyThreadFactory factory = new MyThreadFactory("MyThreadFactory");  
  
    Task task = new Task();  
  
    Thread thread;  
    System.out.printf("Starting the threads\n");  
    for (int i = 0; i < 10; i++) {  
        thread = factory.newThread(task);  
        thread.start();  
    }  
  
    System.out.printf("Factory stats:\n");  
    System.out.printf("%s\n", factory.getStats());  
}  
}
```