Лекція 4. Реалізація рекурсивного підходу в багатопоточному програмуванні мовою Java.

Тема 1. Організація виконання завдань у пулі за алгоритмом «розділити – виконати – об’єднати».
Тема 2. Клас ForkJoinTask. Алгоритм «крадіжки задач» (work stealing).
Тема 3. Класи RecursiveTask та RecursiveAction — спільні характеристики та відмінності.

ForkJoinPool — це реалізація пулу потоків, створена для підтримки рекурсивного підходу типу "розділяй і володарюй":

Fork (розділення): Задача розбивається на менші підзадачі (рекурсивне ділення роботи).

Join (з'єднання): Результати підзадач з’єднуються після завершення їх виконання.

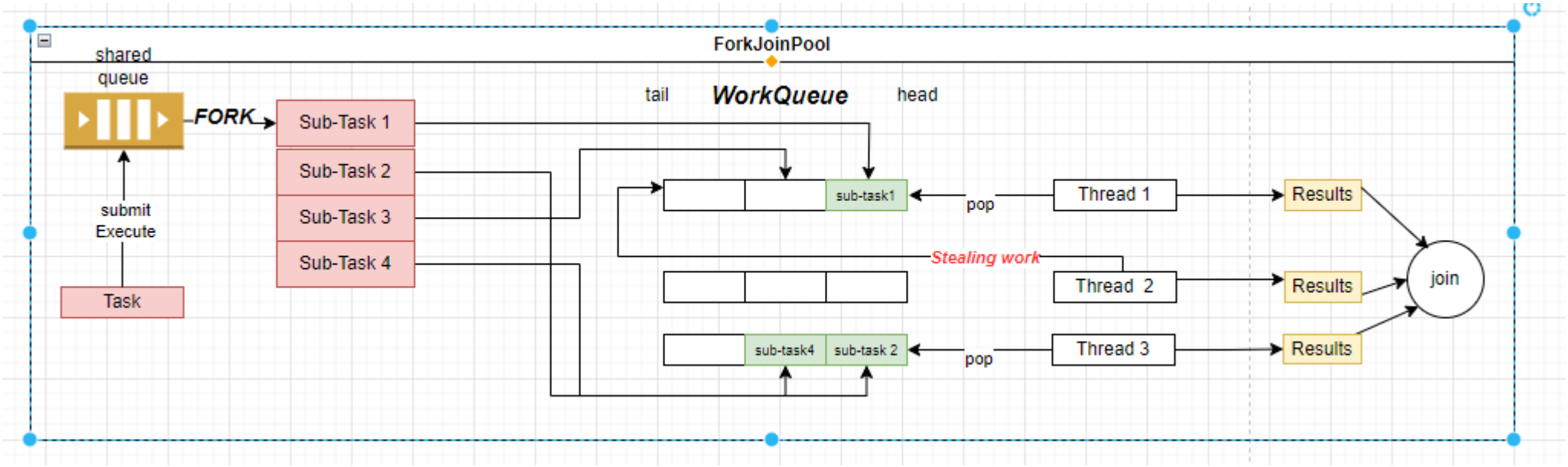
Ці задачі, або одиниці роботи, додаються до спільного пулу потоків для виконання.

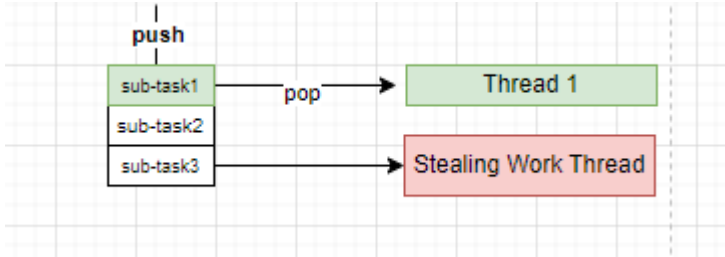
Якщо потік виконав увесь набір задач зі своєї черги, замість того щоб простоювати, він "краде" задачі з черг інших потоків, які досі зайняті. Це називається алгоритм крадіжки роботи «work stealing» - забезпечую баланс навантаження потоків в «загальному пулі».

Основна ідея крадіжки роботи полягає у динамічному балансуванні навантаження під час виконання програми шляхом перерозподілу задач, які розподілені нерівномірно між потоками. Це забезпечує максимально ефективне використання всіх доступних ядер процесора.

public class ForkJoinPool extends AbstractExecutorService

В цьому пулі кожен потік працює зі своєю чергою (черга реалізована за принципом деку), яка називається WorkQueue. Якщо потік виконав всі задачі в «своїй» черзі, тоді він «забирає» задачу з «чужої» черги. Така поведінка реалізована за алгоритмом «work stealing» - забезпечую баланс навантаження потоків в «загальному пулі». Потік зі своєї черги бере задачу – з голови черги (head); потік, який забирає задачу з чужої черги – бере її з хвоста (tail). Таким чином, задачі зі своєї черги завжди виконуються відповідно LIFO, тобто остання додана задача виконується першою.





Якщо потік завершує свою поточну задачу й бачить, що його черга порожня, він стає "злодієм", який намагається "красти" задачі з нижньої частини черги іншого потоку (FIFO-замовлення).

Задачі крадуться з протилежного кінця (FIFO), оскільки старіші задачі зазвичай мають більший обсяг і можуть породжувати більше підзадач, що дозволяє перерозподілити більше роботи.

Під час динамічного створення і завершення задач алгоритм крадіжки роботи балансує навантаження між потоками, переміщуючи задачі з більш завантажених потоків до тих, які простоюють.

Переваги алгоритму крадіжки роботи

- Ефективне використання процесора:

Балансує задачі динамічно між потоками, гарантує, що немає простою.

- Мінімізація конфліктів:

Кожен потік в основному обробляє свою чергу (LIFO), що зменшує конфлікти під час доступу до спільних ресурсів.

- Масштабована паралельність:

Під час створення підзадач вони розподіляються більш рівномірно між потоками, що дозволяє покращити масштабування на багатоядерних системах.

- Покращення локальності пам’яті:

Локальні задачі обробляються у порядку LIFO, що покращує локальність кешу пам’яті (оскільки щойно створені задачі скоріше за все ще перебувають у пам’яті).

Обмеження алгоритму крадіжки роботи

- Витрати на крадіжку задач:

Крадіжка задач спричиняє певні витрати, оскільки потоку необхідно перевіряти черги інших потоків. Якщо занадто багато потоків простоюють, продуктивність падає через надлишкову кількість операцій крадіжки задач.

- Гранулярність задач:

Якщо задачі занадто малі (дрібнозернисті), потоки можуть витрачати надто багато часу на крадіжку задач замість їх виконання, що призводить до неефективності.

- Якщо задачі занадто великі (грубозернисті), може виникнути дисбаланс навантаження, оскільки великі задачі не розбиваються на менші частини для розподілу між потоками.
- Витрати на синхронізацію:

```
public class SumTask extends RecursiveTask<Long> {
    private int[] array;
    private int start;
    private int end;
    private static final int THRESHOLD = 10_000;
    public SumTask(int[] array, int start, int end) {
        this.array = array;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Long compute() {
        if (end - start <= THRESHOLD) {
            long sum = 0;
            for (int i = start; i < end; i++) {
                sum += array[i];
            }
            return sum;
        }

        int middle = (start + end) / 2;
        SumTask leftTask = new SumTask(array, start, middle);
        SumTask rightTask = new SumTask(array, middle, end);
        leftTask.fork();
        long rightResult = rightTask.compute();
        long leftResult = leftTask.join();
        return leftResult + rightResult;
    }
}
```

THRESHOLD - Змінна THRESHOLD визначає, коли суму сегмента слід обчислювати безпосередньо замість її подальшого розбиття. У цьому прикладі порогове значення становить 10 000.

SumTask — це спеціальний клас, який розширює RecursiveTask<Long>, щоб розбити задачу підсумовування масиву на менші підзадачі.

Якщо розмір сегмента масиву досить малий (менший за THRESHOLD – у цьому випадку 10 000), сума обчислюється безпосередньо за допомогою циклу for.

В іншому випадку сегмент ділиться навпіл і створюються дві підзадачі:

leftTask обробляє першу половину сегмента. rightTask обробляє другу половину сегмента.

Паралельне виконання: Метод fork() запускає ліве завдання асинхронно в окремому потоці.

Метод compute() обробляє потрібне завдання в поточному потоці. Метод join() очікує на результат лівої підзадачі. Об'єднання результатів: Результати лівого (leftResult) і правого (rightResult) завдань додаються разом, щоб отримати суму для поточного сегмента.

Клас ForkJoinPool наслідує методи класу AbstractExecutorService

Метод	Опис
ForkJoinPool()	Створює ForkJoinPool із паралелізмом, рівним Runtime.availableProcessors(), використовуючи фабрику потоків за замовчуванням, без UncaughtExceptionHandler і неасинхронний режим обробки LIFO.
ForkJoinPool(int parallelism)	Створює ForkJoinPool із зазначеним рівнем паралелізму, фабрикою потоків за замовчуванням, без UncaughtExceptionHandler і неасинхронним режимом обробки LIFO.
ForkJoinPool(int parallelism, **ForkJoinPool.ForkJoinWorkerThreadFactory** factory, **Thread.UncaughtExceptionHandler** handler, boolean asyncMode)	Конструктор створює екземпляр класу ForkJoinPool, в який включено кількість потоків, яка дорівнює кількості потоків «ядра», які доступні при виконанні програми. MAX_CAP – константа, із значенням 1. Якщо всі потоки «ядра» зайняті, тоді принаймні один потік буде створено. Час протягом якого потік може буди без задачі і «не забирати» з чужої черги 60m перед тим як він Черга типу WorkQueue створюється в конструкторі, її розмір визначається залежно від кількості потоків і типу задач в пулі.
ForkJoinPool.commonPool();	Викликається з конструктору класу ForkJoinPool для створення «Загального» пулу з параметрами: Async Mode = True – задачі такого пулу виконуються asynchronous Uncaught Exception Handler – пул не визначає окремих виняткових ситуацій
public final ForkJoinTask<V> fork()	Додає задачу у чергу пулу, якщо в черги є місце, якщо ні – тоді додає задачу до загальної черги (shared queue)

```
public ForkJoinPool() {
   this(Math.min(MAX_CAP, Runtime.getRuntime().availableProcessors()),
        defaultForkJoinWorkerThreadFactory, null, false,
        0, MAX_CAP, 1, null, DEFAULT_KEEPALIVE, TimeUnit.MILLISECONDS);
}
```

Math.min(MAX_CAP, Runtime.getRuntime().availableProcessors())
Це перший параметр конструктора, який визначає цільову кількість потоків ("паралелізм"), що використовуються пулом.
Runtime.getRuntime().availableProcessors():
Повертає кількість доступних процесорів (або ядер) на машині, де виконується програмний код.
Це дозволяє адаптувати пул потоків до апаратних можливостей системи.
Math.min(MAX_CAP, ...):
Уразі обмеження апаратного середовища, максимальна кількість потоків не може перевищувати MAX_CAP — це значення визначене класом і зазвичай дорівнює 32767.
Наприклад:
Якщо у вашій системі 8 ядер, availableProcessors() поверне 8.
Якщо у системі більше 32 767 ядер (це, практично, лише теоретичний випадок), реальна кількість потоків буде обмежена MAX_CAP = 32767 потоків.
Ось як виглядає типова логіка:
Якщо доступно 8 ядер: Math.min(32767, 8) = 8.
Якщо доступно 32 768 ядер: Math.min(32767, 32768) = 32767.

defaultForkJoinWorkerThreadFactory - Цей параметр визначає фабрику потоків, яка відповідає за створення нових потоків у пулі.
Вона забезпечує створення стандартних потоків типу ForkJoinWorkerThread для пулу.
Якщо необхідно створювати спеціалізовані потоки з додатковою логікою, ви можете передати свою реалізацію ForkJoinWorkerThreadFactory.

```
public static class CustomForkJoinThreadFactory implements ForkJoinPool.ForkJoinWorkerThreadFactory {

    private final int maxThreads;
    private int createdThreads = 0; // Counter to track the number of created threads

    public CustomForkJoinThreadFactory(int maxThreads) {
        this.maxThreads = maxThreads;
    }

    @Override
    public synchronized ForkJoinWorkerThread newThread(ForkJoinPool pool) {
        if (createdThreads >= maxThreads) {
            throw new IllegalStateException("Exceeded maximum number of threads: " + maxThreads);
        }
        createdThreads++;
        return new CustomForkJoinWorkerThread(pool);
    }
```

```
public ForkJoinPool(int parallelism) {
   this(parallelism, defaultForkJoinWorkerThreadFactory, null, false,
       0, MAX_CAP, 1, null, DEFAULT_KEEPALIVE, TimeUnit.MILLISECONDS);
}
```

перший параметр конструктора, який визначає цільову кількість потоків ("паралелізм"),

```
public ForkJoinPool(int parallelism,
                    ForkJoinWorkerThreadFactory factory,
                    UncaughtExceptionHandler handler,
                    boolean asyncMode) {
   this(parallelism, factory, handler, asyncMode,
       0, MAX_CAP, 1, null, DEFAULT_KEEPALIVE, TimeUnit.MILLISECONDS);
}
```

Клас ForkJoinPool наслідує методи класу AbstractExecutorService

Метод	Опис
ForkJoinPool.commonPool();	Викликається з конструктору класу ForkJoinPool для створення «Загального» пулу з параметрами: Async Mode = True – задачі такого пулу виконуються asynchronous Uncaught Exception Handler – пул не визначає окремих виняткових ситуацій

public void
execute(ForkJoinTask<?> task) {
Objects.*requireNonNull*(task);
poolSubmit(true, task);
}
	Ініціює виконання завдання типу ForkJoinTask

public final ForkJoinTask<V> fork()	Додає задачу у чергу пулу, якщо в черги є місце, якщо ні – тоді додає задачу до загальної черги (shared queue)

Клас ForkJoinTask – абстрактний клас для опису об'єкту «задача», яка виконуватиметься в ForkJoinPool.

Задача, починається виконуватись в ForkJoinPool, як тільки її надіслано методом submit(). Якщо задача надсилається першою, тоді використовують метод ForkJoinPool.commonPool() або ForkJoinTask може бути виконана в ForkJoinPool, тільки викликом 2х методів – fork(); join().

- ✓ Метод fork() – організовує асинхронне виконання під-задач;
- ✓ Метод join() – не виконується доки, не отримано відповідні результати задачі. Таким чином метод join() є аналогом Future.get().

Статуси задачі: isDone() – виконано, але включає випадок, коли задачу «відкликано» (cancelled).

- ✓ isCompletedNormally() - виконано, без виникнення особливих ситуацій.
- ✓ isCancelled - включає випадок, коли задачу «відкликано» (cancelled).
- ✓ isCompletedAbnormally - включає випадки, коли задачу «відкликано» (cancelled) або помилка при виконанні обчислень.

Типи задач на основі ForkJoinTask

ForkJoinTask має двох стандартних дочірніх класів, які спрощують реалізацію завдань:

RecursiveTask<V>:

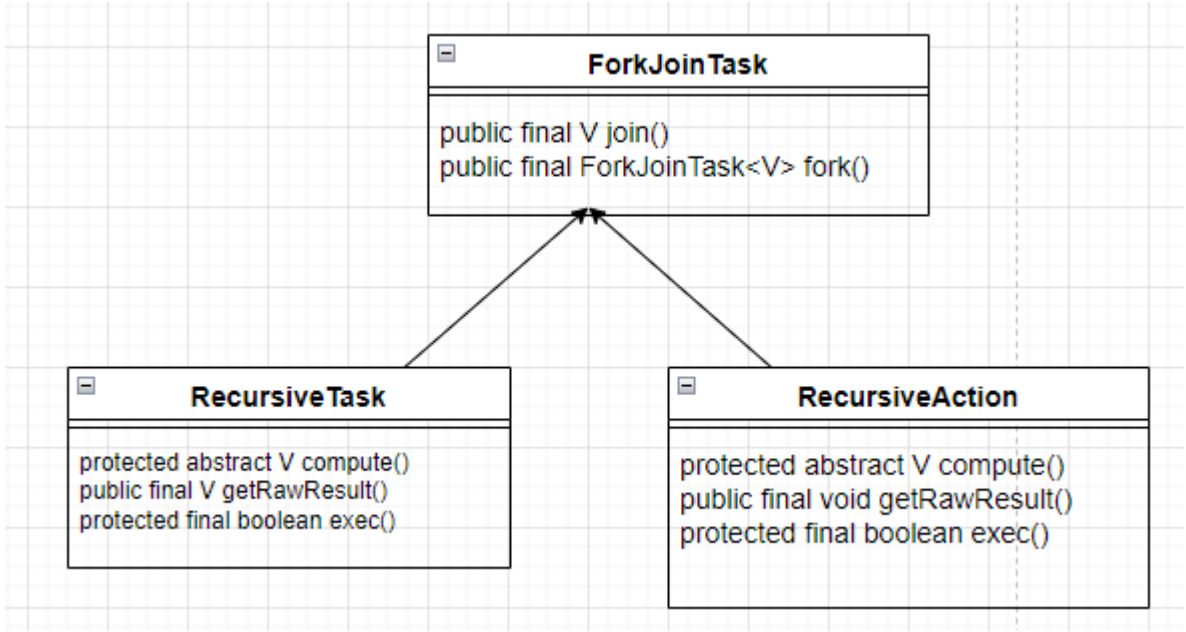
Використовується для завдань, які повертають результат (з типом V).

Підходить для розрахунків, результатом яких є, наприклад, сума, добуток, максимальне значення тощо.

RecursiveAction:

Використовується для задач, які не повертають результат.

Підходить для задач, пов'язаних з операціями, які впливають лише на зовнішні ресурси, наприклад, сортування масиву.



```
public final V get() throws InterruptedException, ExecutionException {
    int stat = status;
    int s = ((stat < 0) ? stat :
             (Thread.interrupted()) ? ABNORMAL :
             awaitDone(true, 0L));
    if (s == ABNORMAL)
        throw new InterruptedException();
    else if ((s & ABNORMAL) != 0)
        reportException(true);
    return getRawResult();
}
```

використовується для отримання результату виконання завдання, що обробляється у ForkJoinPool.
get() блокується, поки завдання не завершиться (якщо воно ще виконується).
Якщо завдання завершилося помилкою або було перерване, метод генерує виняток (InterruptedException або ExecutionException).
Якщо завдання завершено успішно, метод повертає обчислений результат.
Поле status у ForkJoinTask визначає стан (статус) виконання завдання. Стани можуть бути наступними:
NEW (значення > 0): Завдання ще не завершене.
DONE (значення < 0): Завдання завершене.
ABNORMAL: Завдання завершилося помилкою або було скасоване.
Якщо stat < 0: Завдання завершено, і ми використовуємо статус, який уже встановлено.
Якщо Thread.interrupted() повертає true:
Поточний потік виклику був перерваний (перевірка Thread.interrupted()).
Статус завдання встановлюється в ABNORMAL, оскільки перерваний потік означає, що завдання не може бути коректно завершено.
У всіх інших випадках буде викликаний приватний метод awaitDone(true, 0L), який блокується до моменту завершення завдання.

```
@Override
public State state() {
   int s = status;
   return (s >= 0) ? State.RUNNING :
      ((s & (DONE | ABNORMAL)) == DONE) ? State.SUCCESS:
      ((s & (ABNORMAL | THROWN)) == (ABNORMAL | THROWN)) ? State.FAILED :
      State.CANCELLED;
```

```
public final V get(long timeout, TimeUnit unit)
    throws InterruptedException, ExecutionException, TimeoutException {
    long nanos = unit.toNanos(timeout);
    int stat = status;
    int s = ((stat < 0) ? stat :
             (Thread.interrupted()) ? ABNORMAL :
             (nanos <= 0L) ? 0 :
             awaitDone(true,  (System.nanoTime() + nanos) | 1L));
    if (s == ABNORMAL)
        throw new InterruptedException();
    else if (s >= 0)
        throw new TimeoutException();
    else if ((s & ABNORMAL) != 0)
        reportException(true);
    return getRawResult();
}
```

```
public static int getQueuedTaskCount() {
   Thread t; ForkJoinPool.WorkQueue q;
   if ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread)
      q = ((ForkJoinWorkerThread)t).workQueue;
   else
      q = ForkJoinPool.commonQueue();
   return (q == null) ? 0 : q.queueSize();
}
```

Метод getQueuedTaskCount(), що належить класу ForkJoinTask, використовується для отримання кількості задач у черзі поточного потоку в контексті ForkJoinPool.
Спочатку отримується поточний потік за допомогою Thread.currentThread().
Перевіряється, чи є поточний потік екземпляром ForkJoinWorkerThread:
Потоки типу ForkJoinWorkerThread керуються ForkJoinPool і мають спеціальну чергу завдань workQueue.
Цей тип потоків створюється ForkJoinPool спеціально для виконання завдань.
Якщо поточний потік є частиною ForkJoinPool (тобто є ForkJoinWorkerThread), метод отримує чергу завдань (workQueue) для цього потоку.
Якщо поточний потік не є частиною ForkJoinPool (наприклад, це потік main або звичайний потік), метод використовує загальну чергу (ForkJoinPool.commonQueue()).
Спільна черга (commonQueue) належить до пулу загального використання (commonPool), де виконуються завдання без прив'язки до конкретного потоку.

```
public final ForkJoinTask<V> fork() {
    Thread t;
    ForkJoinWorkerThread wt;
    ForkJoinPool p;
    ForkJoinPool.WorkQueue q;
    boolean internal;
    if (internal =
      (t = Thread.currentThread()) instanceof
ForkJoinWorkerThread) {
      q = (wt = (ForkJoinWorkerThread)t).workQueue;
      p = wt.pool;
    }
    else
      q = (p =
ForkJoinPool.common).externalSubmissionQueue();
    q.push(this, p, internal);
    return this;
  }
```

Thread t = Thread.currentThread():
Отримує поточний потік, який викликає метод.
internal = t instanceof ForkJoinWorkerThread: Перевіряє, чи викликаний метод потоком, який є частиною ForkJoinPool (тобто, чи він є об'єктом класу ForkJoinWorkerThread).
Якщо виклик надходить із внутрішнього потоку ForkJoinWorkerThread, тоді:
Завдання додається у власну локальну чергу (workQueue) цього потоку.
Це оптимізує виконання, оскільки завдання, ймовірно, виконуватиметься цим же потоком, не створюючи додаткових витрат на передачу завдання.
Якщо потік не є ForkJoinWorkerThread (тобто виклик "ззовні"), означає, що завдання запускається із зовнішнього середовища і буде додано до зовнішньої черги (додаткова умова).

```
@Override
public V resultNow() {
    int s = status;
    if ((s & DONE) == 0)
        throw new IllegalStateException("Task has not completed");
    if ((s & ABNORMAL) != 0) {
        if ((s & THROWN) != 0)
            throw new IllegalStateException("Task completed with exception");
        else
            throw new IllegalStateException("Task was cancelled");
    }
    return getRawResult();
}
```

Використовується для неблокуючого доступу до результату завдання, наприклад, для періодичного опитування або перевірки

```
ForkJoinPool pool = new ForkJoinPool();
 SimpleSum task = new SimpleSum(1, 10_000);
 pool.execute(task);
 try {
    while (!task.isDone()) {
        System.out.println("Waiting for task to complete...");
        Thread.sleep(100);
    }
    int result = task.resultNow(); // Отримує результат негайно
    System.out.println("Sum result: " + result);
 } catch (IllegalStateException | InterruptedException e) {
    e.printStackTrace();
 }
pool.shutdown();
```

```
protected static ForkJoinTask<?> peekNextLocalTask() {
   Thread t; ForkJoinPool.WorkQueue q;
   if ((t = Thread.currentThread()) instanceof ForkJoinWorkerThread)
      q = ((ForkJoinWorkerThread)t).workQueue;
   else
      q = ForkJoinPool.commonQueue();
   return (q == null) ? null : q.peek();
}
```

Якщо поточний потік ForkJoinWorkerThread, метод отримує його локальну чергу завдань через поле workQueue. Локальна черга завдань - це структура даних, яка використовується для виконання завдань потоком у принципі LIFO (Last In, First Out): останній доданий елемент буде виконаний першим.

Якщо поточний потік не є ForkJoinWorkerThread (наприклад, потік main або будь-який потік, не пов'язаний з ForkJoinPool), метод замість локальної черги звертається до спільної черги завдань пулу: commonQueue(). commonQueue - це черга завдань для асинхронних завдань у рамках загального пулу потоків.

Внутрішній виклик (ForkJoinWorkerThread):

Якщо завдання відправляється до ForkJoinPool з внутрішнього робочого потоку:
Завдання додається до локальної черги, асоційованої з цим потоком.
Це прискорює виконання, оскільки завдання залишиться "локальним" для цього потоку за принципом LIFO.
Зовнішній виклик (не ForkJoinWorkerThread):

Якщо fork() викликається звичайним потоком (наприклад, main), завдання додається до зовнішньої спільної черги.
Потоки з ForkJoinPool зможуть забирати завдання з цієї черги.

public <T> T invoke(ForkJoinTask<T> task) {
  Objects.*requireNonNull*(task);
  poolSubmit(true, task);
  try {
  return task.join();
  } catch (RuntimeException | Error unchecked) {
  throw unchecked;
  } catch (Exception checked) {
  throw new RuntimeException(checked);
  }
}
	Метод invoke(ForkJoinTask<T> task) є частиною класу ForkJoinPool у Java. Він використовується для ***синхронного виконання завдання в межах ForkJoinPool. Це означає, що метод запускає виконання завдання і блокується до тих пір, поки завдання не завершиться, повертаючи результат виконання.***
	Метод приймає об'єкт типу ForkJoinTask<T> для виконання.
	Завдання передається в пул через poolSubmit(true, task).
	Далі task.join() використовується для очікування завершення завдання і отримання результату.
	У разі виникнення винятків вони обробляються і повторно викидаються.
	poolSubmit — це внутрішній метод, який додає завдання task до пулу ForkJoinPool для виконання.
	Параметр true вказує, що це асинхронне виконання. Це стандартний спосіб для запуску завдань у ForkJoinPool.
	Механізм роботи:
	Завдання додається до черги потоків ForkJoinWorkerThread (місцевої або спільної черги в пулі).
	Потоки з ForkJoinPool забирають це завдання з черги для виконання.
	Цей етап є пусковим механізмом, який гарантує, що завдання запускається в пулі безпосередньо з методу invoke.

public void execute(ForkJoinTask<?> task) {
  Objects.*requireNonNull*(task);
  poolSubmit(true, task);
}
	Використовується для **асинхронного виконання завдань**.
	Не повертає результат безпосередньо.
	Підходить для завдань у фоновому режимі.

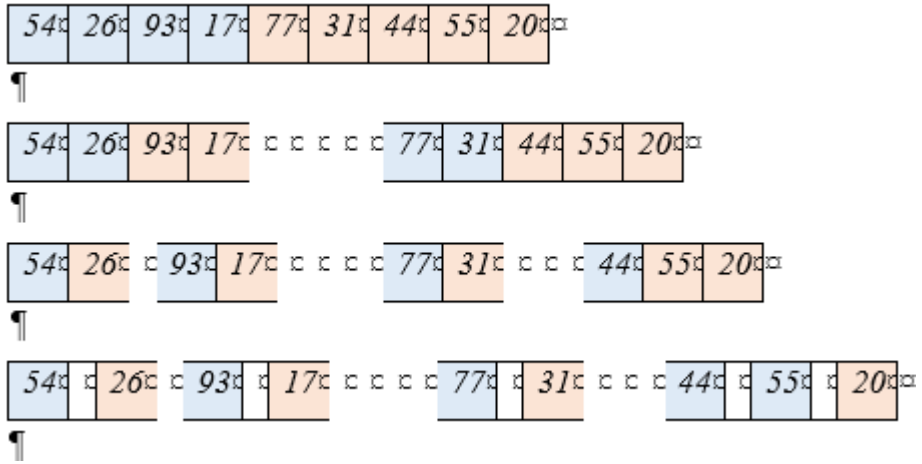
Тип виконання	Асинхронно	Синхронно (блокує потік до завершення завдання)
Очікування завершення завдання	Не чекає	Чекає завершення завдання
Отримання результату	Не повертає результат напряму. Потрібен `task.join()`	Повертає результат прямо (метод `return`)
Повертає	`void`	Значення типу `V` з `ForkJoinTask<V>`
Використання	Коли завдання може виконуватися у фоновому режимі.	Коли необхідно отримати результат негайно.

```
pool.execute(task);
System.out.println("Main продовжує працювати");
int resultExecute = task.join();
```

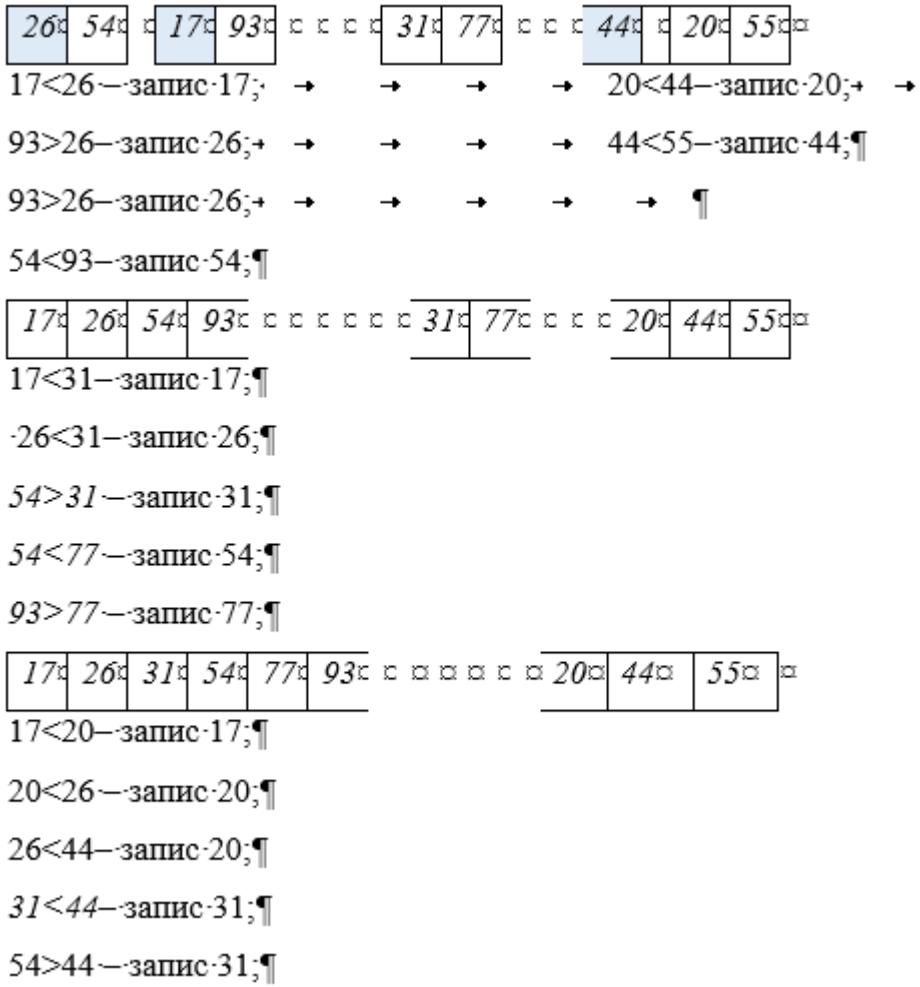
```
int resultInvoke = pool.invoke(taskForInvoke);
System.out.println("Результат:" + resultInvoke);
```

Приклад реалізації алгоритму «Сортування злиттям» методами ForkJoinPool

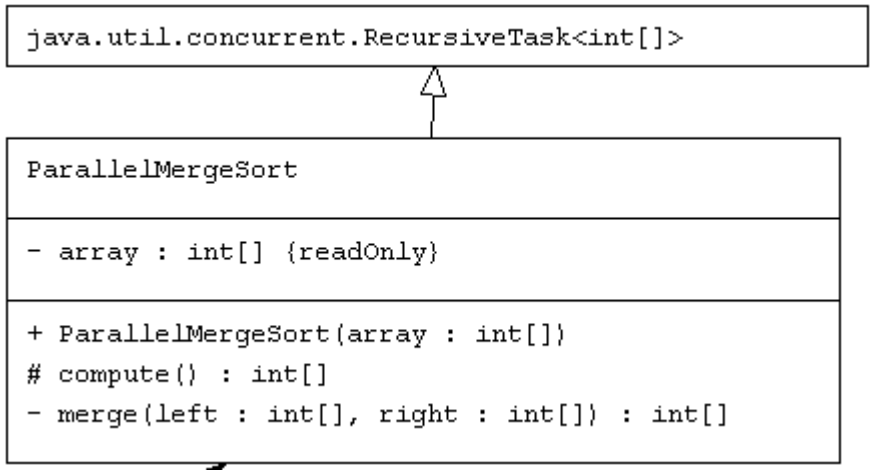
1й етап розділення елементів



2й етап розділення елементів



Класс – ParallelMergeSort, наслідує від RecursiveTask, реалізує метод compute(), в якому викликаємо fork() для передачі задачі на виконання в асинхронному ForkJoin пулі та за ним join() – щоб дочекатись об'єднаного результату виконаних під задач.



В методі main() створюємо екземпляр ForkJoinPool та викликаємо метод invoke (ParallelMergeSort) для початку виконання задачі початкового рівня (тобто не розділеної на підзадачі)

Питання

1. У чому полягає суть рекурсивного підходу до паралельних обчислень у Java?
2. Яка ідея алгоритму «розділити – виконати – об’єднати» (Divide and Conquer) у контексті ForkJoinPool?
3. Які переваги використання ForkJoinPool порівняно зі звичайними пулами потоків?
4. Яке призначення абстрактного класу ForkJoinTask і які основні методи він надає?
5. Як працює алгоритм «крадіжки задач» і яку роль він відіграє в балансуванні навантаження між потоками?
6. Що таке локальна черга задач у ForkJoinPool і як вона використовується під час виконання рекурсивних завдань?
7. У чому полягає відмінність між класами RecursiveTask<V> та RecursiveAction?
8. У яких випадках доцільно використовувати RecursiveTask, а в яких — RecursiveAction?
9. Яке значення має порогове значення (threshold) у рекурсивних задачах і як воно впливає на продуктивність?
10. Які типові помилки виникають при реалізації рекурсивних паралельних алгоритмів у ForkJoinPool?