Programowanie współbieżne

JAVA – współbieżność i mechanizmy synchronizacji

Prowadzący: dr inż. Jarosław Rulka
jaroslaw.rulka@wat.edu.pl

- W przypadku języka Java mamy do czynienia z dwoma programowymi typami jednostek kodu wyrażającymi współbieżność. Są to:
  - procesy – komunikacja i synchronizacja realizowana z wykorzystaniem mechanizmów systemu operacyjnego,
  - wątki (ang. *threads*) – komunikacja i synchronizacja (wewnątrz procesu) wspierana przez mechanizmy języka.

Tworzenie i uruchamianie procesów w języku Java

```
1.  try {
2.     ProcessBuilder pb =
3.         new ProcessBuilder("c:/apps/bin/prog.exe", "");
4.     Process p1 = pb.start();
5.     Process p2 = pb.start();
6.     Process p3 =
7.         new ProcessBuilder("c:/apps/bin/prog2.exe", "").start();
8.  } catch (Exception e) {
9.     System.out.println("Błąd: _" + e.getMessage());
10. }
```

Definicja klasy, tworzenie i uruchomienie wątku – klasa ***Thread*** (1/2)

```
1. public class MyThread extends Thread {
2.   public MyThread(String num) {
3.     super(num);
4.   }
5.   public void run() { // instrukcje wykonywane
                         // w ramach wątku
6.     System.out.println("Pozdrowienia z wątku "
           + Thread.currentThread().getName());
7.   }
8. }
```

Definicja klasy, tworzenie i uruchomienie wątku – klasa ***Thread*** (2/2)

```
1. public class Test {
2.   public static void main(String[] args) {
3.     Thread[] w = new Thread[10];
4.     for(int i = 0; i < w.length; i++) {
5.       w[i] = new MyThread("" + i);
6.     }
7.     for(int i = 0; i < 10; i++) {
8.       w[i].start();
9.     }
10.  }
11.}
```

```
Pozdrowienia z wątku 5
Pozdrowienia z wątku 3
Pozdrowienia z wątku 7
Pozdrowienia z wątku 0
Pozdrowienia z wątku 1
Pozdrowienia z wątku 2
Pozdrowienia z wątku 8
Pozdrowienia z wątku 4
Pozdrowienia z wątku 9
Pozdrowienia z wątku 6
```

Definicja klasy, tworzenie i uruchomienie wątku – klasa ***Thread*** (2/2)

```
1. public class Test {
2.   public static void main(String[] args) {
3.     Thread[] w = new Thread[10];
4.     for(int i = 0; i < w.length; i++) {
5.       w[i] = new MyThread("" + i);
6.     }
7.     for(int i = 0; i < 10; i++) {
8.       w[i].run();
9.     }
10.  }
11.}
```

```
Pozdrowienia z wątku 0
Pozdrowienia z wątku 1
Pozdrowienia z wątku 2
Pozdrowienia z wątku 3
Pozdrowienia z wątku 4
Pozdrowienia z wątku 5
Pozdrowienia z wątku 6
Pozdrowienia z wątku 7
Pozdrowienia z wątku 8
Pozdrowienia z wątku 9
```

Definicja klasy, tworzenie i uruchamianie wątku – interfejs ***Runnable*** (1/2)

```
1. public class MyThread implements Runnable {
2.   public void run() { // instrukcje wykonywane
                         // w ramach wątku
3.     System.out.println("Pozdrowienia z wątku "
           + Thread.currentThread().getName());
4.   }
5. }
```

Definicja klasy, tworzenie i uruchamianie wątku – interfejs ***Runnable*** (2/2)

```
1. public class Test {
2.   public static void main(String[] args) {
3.     Thread[] w = new Thread[10];
4.     for(int i = 0; i < 10; i++) {
5.       MyThread t = new MyThread();
6.       w[i] = new Thread(t, "" + i);
7. // lub      w[i] = new Thread(new MyThread());
8.     }
9.     for(int i = 0; i < 10; i++) {
10.      w[i].start();
11.    }
12.  }
13.}
```

```
Pozdrowienia z wątku 1
Pozdrowienia z wątku 9
Pozdrowienia z wątku 6
Pozdrowienia z wątku 8
Pozdrowienia z wątku 5
Pozdrowienia z wątku 2
Pozdrowienia z wątku 0
Pozdrowienia z wątku 4
Pozdrowienia z wątku 3
Pozdrowienia z wątku 7
```

- **Nowy** – chwilowy stan w trakcie tworzenia wątku; Przydzielane zasoby systemowe i inicjalizacja wątku.
- **Gotowy** (ang. *runnable*) – stan oznaczający, że wątek może działać, kiedy tylko mechanizm przydziału czasu udostępni mu procesor. Wątek przez nic nie jest powstrzymywany.
- **Zablokowany** (zawieszony) – jest coś, co mu przeszkadza działać. Jest pomijany przez zarządcę podziału czasu do chwili, aż będzie w stanie *gotowy*.
- **Uśmiercony** – wątek nie otrzymuje już i nie otrzyma czasu procesora (nie jest ani *gotowy*, ani *zablokowany*). Jego zadanie zakończyło się normalnie (zakończenie metody `run()`) lub został przerwany bez obsługi wyjątku.

- Priorytety stanowią dla planisty wątków informację o poziomie ważności danego wątku.
- Priorytety decydują o kolejności wznawiania zablokowanych wątków, które w danej chwili oczekują na wznowienie.
- Wątki o niższych priorytetach nigdy nie będą przyczyną zakleszczenia – są tylko wykonywane rzadziej.
- W zdecydowanej większości przypadków wątki powinny mieć identyczne poziomy priorytetów.

Synchronizacja wątków – `join()`

- Pozwala na synchronizację działania wątków z punktu widzenia zakończenia działania jednego z nich;
- Wywołanie metody **`join()`** innego wątku blokuje (wstrzymuje) wątek wywołujący do czasu zakończenia wykonywania wątku, którego metodę **`join`** wywołano;
- **`join(ms)`** – wersja metody pozwalająca na wstrzymanie wątku wywołującego w oczekiwaniu na zakończenie innego wątku przez zadany czas;
  - Najpóźniej po tym czasie wątek wywołujący zostanie wznowiony i będzie mógł kontynuować działanie.
- Wywołanie metody **`join()`** można przerwać poprzez wywołanie metody **`interrupt()`** wątku wywołującego (konieczność stosowania **`try`** i **`catch`**).

Komunikacja poprzez przerwania (1/2)

```
1. public class MyThread extends Thread {
2.   public void run() {
3.     for(int i = 0; i < 10; i++)
4.       try {
5.         Thread.sleep(2000);
6.         System.out.println("Spałem przez 2 sekundy");
7.       } catch(InterruptedException e) {
8.         System.out.println("Dostałem sygnał interrupt");
           break;
9.       }
10.  }
11.}
```

Komunikacja poprzez przerwania (2/2)

```
1. public class Test {
2.   public static void main(String[] args) throws Exception {
3.     Thread w = new MyThread();
4.     w.start();
5.     Thread.sleep(5000);
6.     w.interrupt();
7.     w.join();
8.     System.out.println(”KONIEC”);
9.   }
10.}
```

```
Spałem przez 2 sekundy
Spałem przez 2 sekundy
Dostałem sygnał interrupt
KONIEC
```

Przeplot

```
1.  public class Licznik {
2.    private long c = 0;
3.    public void inc() {
4.      c++;
5.    }
6.    public long get() {
7.      return c ;
8.    }
9.  }
10. class MyThread extends Thread {
11.   private Licznik licz;
12.   public MyThread( Licznik licz) {
13.     this.licz = licz ;
14.   }
15.   public void run ( ) {
16.     for(int i = 0; i < 5000000;
              i++) {
17.       licz.inc();
18.     }
19.   }
20. }
```

```
21. public class Test {
22.   public static void main (
            String[] args)
            throws Exception {
23.     Licznik licz = new Licznik();
24.     Thread w0 = new MyThread(licz);
25.     Thread w1 = new MyThread(licz);
26.     Thread w2 = new MyThread(licz);
27.     w0.start();
28.     w1.start();
29.     w2.start();
30.     w0.join();
31.     w1.join();
32.     w2.join();
33.     System.out.println("Licznik="
            + licz.get());
34.   }
35. }
```

Operacje atomowe

- Instrukcje atomowe (niepodzielne) – taka instrukcja, która wykonywana jest przez procesor niepodzielnie. Znaczy to że o ile się rozpocznie, musi być w trybie wyłącznym wykonana i zakończona.
- Trudno jest stwierdzić, jakie operacje zapisane w języku wyższego poziomu będą operacjami atomowymi.
- JAVA – następujące odwołania do zmiennych są operacjami atomowymi:
  - odczyt i zapis do zmiennych typów referencyjnych oraz typów prostych zadeklarowanych ze słowem kluczowym **`volatile`** (tzw. zmienne ulotne).

- W Javie mamy dostępną klasę semafora zliczającego oraz uogólnionego o nazwie `Semaphore`. Dziedziczy z klasy `Object`.
- Konstruktory:
  - `Semaphore(int permits)`: tworzy obiekt semafora o zadanym stanie początkowym (dostępne zezwolenia) w wersji nieuczciwej algorytmu synchronizacji (aktywne czekanie)
  - `Semaphore(int permits, boolean fair)`: tworzy obiekt semafora o zadanym stanie początkowym (dostępne zezwolenia) i zadanej wersji algorytmu synchronizacji

Klasa Semaphore cd.

- `void acquire()`: opuszcza semafor o 1 (blokuje gdy stan 0/opuszczony) – wznowienie po wywołaniu `release` lub `interrupt`
- `void acquire(int permits)`: opuszcza semafor o zadaną wartość (blokuje gdy stan mniejszy od zadanej wartości) – wznowienie po wywołaniu `release` (gdy stan 0 lub więcej) lub `interrupt`
- `void acquireUninterruptibly()`, `void acquireUninterruptibly(int permits)`: wersje opuszczania semafora bez przerywania blokowania
- `void release()`: podnosi semafor o 1 (podnosi stan semafora lub odblokowuje wątek)
- `void release(int permits)`: podnosi semafor o zadaną wartość (podnosi stan semafora i/lub odblokowuje wątek)
- `boolean tryAcquire()`: opuszcza semafor o 1, tylko gdy jest podniesiony
- `boolean tryAcquire(int permits)`: opuszcza semafor o zadaną wartość, tylko gdy jest w stanie większym lub równym zadanej wartości
- `int availablePermits()`: zwraca stan semafora

Semaphore – przykład 1/2

```
1.  public class MyThread extends Thread {
2.     Semaphore sem;
3.     public MyThread(Semaphore sem, String name) {
4.        super(name);
5.        this.sem = sem;
6.     }
7.     public void run() {
8.        ...
9.        try {
10.          // Protokół wstępny
11.          sem.aquire();
12.          // Sekcja krytyczna
13.          ...
14.       } catch (InterruptedException e) {
15.          ...
16.       }
17.       // Protokół końcowy
18.       sem.release();
19.    }
20. }
```

Semaphore – przykład 2/2

```
1. public class Test {
2.   public static void main(String[] args) throws Exception {
3.     Semaphore sem = new Semaphore(1);
4.     Thread w1 = new MyThread(sem, ”W-1”);
5.     Thread w2 = new MyThread(sem, ”W-2”);
6.     w1.start();
7.     w2.start();
8.     w1.join();
9.     w2.join();
10.    System.out.println(”KONIEC”);
11.  }
12.}
```

Blok synchronizacji – `synchronized` (1/2)

- Każdy obiekt w Javie zawiera pojedynczą blokadę zwaną także monitorem (również sama klasa posiada blokadę).
- Synchronizacja na tej blokadzie realizowana jest poprzez:
  - synchronizowaną metodę (statyczną i dynamiczną);
  - synchronizowany blok instrukcji (w metodzie statycznej i dynamicznej).
- **`synchronized`** – słowo kluczowe do definicji bloku programu chronionego przez mechanizm wzajemnego wykluczania;
- Składnia metody synchronizowanej
  - `public synchronized void synchFunc1() { /*…*/ }`
- Wywołanie metody synchronizowanej powoduje zajęcie blokady/monitora obiektu przez wątek i wstrzymanie innych wywołań metod/bloków synchronizowanych tego obiektu przez inne wątki do czasu zwolnienia blokady;
- Pola chronione muszą być oznaczone jako **`private`**.

Synchronizowana metoda (2/2)

```
1. public class Licznik {
2.    private long c = 0;
3.    public synchronized void inc() {
4.       c++;
5.    }
6.    public synchronized long get() {
7.       return c;
8.    }
9. }
```

Synchronizowany blok instrukcji (1/2)

- Synchronizowany dostęp do fragmentu kodu (bloku instrukcji) – nie zaś całej metody;
- Synchronizowany fragment kodu nazywa się blokiem synchronizującym;
  - `synchronized(this) {/*..*/}`
  - `synchronized(syncObject) {/*..*/}`

Synchronizowany blok instrukcji (2/2)

```
1.  public class Counter {
2.    private long c = 0;
3.    public void inc() {
4.      ...
5.      synchronized(this) {
6.        c++;
7.      }
8.    }
9.    public long get() {
10.     synchronized(this) {
11.       return c;
12.     }
13.   }
14. }
```

```
1.  public class DoubleCounter {
2.    private long c1 = 0;
3.    private long c2 = 0;
4.    private Object lock1 =
5.        new Object();
6.    private Object lock2 =
7.        new Object();
8.
9.    public void inc1() {
10.     synchronized(lock1) {
11.       c1++;
12.     }
13.   }
14.   public void inc2() {
15.     synchronized(lock2) {
16.       c2++;
17.     }
18.   }
19. }
```

Wielokrotne zajęcie blokady

- Dany wątek może wielokrotnie zajmować tę samą blokadę (związaną z tym samym obiektem) bez jej uprzedniego zwalniania.

```
1. public class SomeClass {
2.   public void method1() {
3.     //...
4.     synchronized(this) {/* ... */}
5.   }
6.   public synchronized method2() {
7.     method1();
8.     //...
9.   }
10.  public void method3() {
11.    //...
12.    synchronized(this) {
13.      method2();
14.      //...
15.    }
16.  }
17.}
```

`wait()`, `notify()`, `notifyAll()` (1/3)

- Metody są składowymi klasy bazowej **`Object`** – nie zaś klasy **`Thread`**;
- Instancję klasy **`Object`** można traktować, jako zmienną warunkową i wraz z metodami synchronizowanymi stanowi implementację mechanizmu monitora.
- **`wait()`** – metoda pozwalająca na zawieszenie wykonania wątku;
  - **`wait()`** – bez ograniczenia na czas zawieszenia;
  - **`wait(ms)`** – metoda się kończy w wyniku wywołania **`notify()`** lub **`notifyAll()`** lub po upłynięciu podanego czasu;
  - wywołanie metody zwalnia blokadę obiektu;
- **`notify()`**, **`notifyAll()`** – metody sygnalizujące i odwieszające wątek/wątki zawieszony na obiekcie;
- **Ważne!** Wywołania metod można umieszczać wyłącznie wewnątrz metody synchronizowanej lub bloku synchronizowanego dot. tego samego obiektu.

wait(), notify(), notifyAll() (2/3)

```
1.  public class MojaKlasa {
2.    public synchronized void metoda1() {
3.      wait();
4.      ...
5.      notify();
6.    }
7.    public void metoda2() {
8.      ...
9.      synchronized (this) {
10.       wait();
11.     }
12.     ...
13.     synchronized (this) {
14.       notify();
15.     }
16.   }
17. }
```

`wait()`, `notify()`, `notifyAll()` (3/3)

- Java ma zaimplementowaną semantykę Mesa (mechanizm monitorów).
- Wywołanie metody **`wait()`** należy zatem ujmować w pętli **`while`** sprawdzającej wystąpienie oczekiwanych okoliczności, ponieważ:
  - ta sama blokada może wstrzymywać kilka wątków z tego samego powodu; pierwszy wybudzony wątek może zmienić sytuację; W takim przypadku kolejne wybudzone wątki powinny zawiesić swoje wykonanie do powtórnej zmiany warunków;
  - kiedy wątek jest wybudzany z **`wait()`**, jakieś inne wątki mogą zmienić sytuację powodując nieaktualność działań wątku i konieczność ponownego zawieszenia;
  - na blokadzie obiektu może oczekiwać kilka wątków zawieszonych z różnych powodów i wybudzanych wywołaniem **`notifyAll()`**; Wątek musi zatem sprawdzić, czy zmiany okoliczności odnoszą się do niego.

- W monitorze klasycznym możemy zadeklarować wiele zmiennych warunkowych.
- Klasa lub obiekt w Javie w przybliżeniu odpowiada monitorowi z tylko jedną zmienną warunkową.
- Specyfikacja Javy mówi, że wątek wywołujący metodę `notify()` kontynuuje pierwszy swoje działanie (przed wątkiem odblokowanym).
  - Odpowiada to semantyce Mesa.

Interfejs **`Lock`** (1/2)

- Interfejs **`Lock`** – jest rdzeniem mechanizmu zamków (mutexów) jawnych;
- Zdefiniowany w pakiecie `java.util.concurrent.locks`;
- Obiekt jest tworzony jawnie i następnie jawnie zamykany i otwierany przez wątki;
- Operacje interfejsu `Lock`:
  - `lock()`: zajmuje/zamyka/rygluje zamek (jeżeli zamek jest zajęty przez inny wątek operacja blokuje wątek wywołujący)
  - `tryLock()`: warunkowo rygluje zamek (jeżeli nie jest zajęty przez inny wątek)
  - `unLock`: zwalnia/otwiera/odryglowuje zamek
  - `Condition newCondition()`: zwraca nową instancję obiektu zmiennej warunkowej związanej z tym zamkiem

Interfejs **Lock** (2/2)

```
1. public class Licznik {
2.    private long c = 0;
3.    private Lock lock = new ReentrantLock();

4.    public void inc() {
5.      lock.lock()
6.      try {
7.        c++;
8.      } finally {
9.        lock.unlock();
10.     }
11.   }
12.   public long get() {
13.     return c;
14.   }
15. }
```

Interfejs `Condition`

- Interfejs **`Condition`** – przeznaczony do implementacji mechanizmu zmiennych warunkowych ściśle związany z mechanizmem zamków;
- Zdefiniowany w pakiecie `java.util.concurrent.locks`;
- Obiekt zmiennej warunkowej tworzony przez funkcję `newCondition()` danego obiektu zamka i ściśle z nim powiązany;
- Operacje interfejsu `Condition`:
  - `await()`: blokuje zawsze/bezwzględnie wątek wywołujący zwalniając jednocześnie (w sposób atomowy) powiązany z nim zamek. Zawieszenie wątku trwa do czasu odebrania sygnału wysłanego przez inny wątek na tym samym obiekcie zmiennej warunkowej.
  - `signal()`: wznawia jeden wybrany wątek.
  - `signalAll()`: wznawia wszystkie wątki.

Przykład definicji monitora

```
1.  public class MyMonitor {
2.    ...
3.    final Lock dostep = new ReetrantLock();
4.    final Condition zmWar1 = dostep.newCondition();
5.    final Condition zmWar2 = dostep.newCondition();

6.    public void metoda1() {
7.      dostep.lock();
4.      try {
5.        ...
1.        while (warunek1)
2.          zmWar1.await();
3.        ...
4.        zmWar2.signal(); // zmWar2.signalAll();
5.        ...
6.      } finally {
4.        dostep.unlock();
5.      }    ...
1.    }
2.    public void metoda2() {
3.      ...
4.    }
5.  }
```

Wątek a zadanie 1/2

- Pojęcie wątku (w szczególności obiektu klasy `Thread`) odnosi się do bytu programowego i systemowego reprezentującego jednostkę sekwencyjnego wykonywania zdefiniowanej części kodu programu w ramach współbieżnego środowiska procesu systemowego reprezentującego uruchomiony program.
- Zdefiniowaną część kodu realizowaną sekwencyjnie możemy traktować w kategoriach **zadania do wykonania,** wątki zaś możemy zatem widzieć w kategoriach **wykonawców zadań**.

Wątek a zadanie 2/2

- Zarządzanie wątkami (tworzenie, uruchamianie, przełączanie kontekstu) jest relatywnie czasochłonne.
- Wątek, który zakończył wykonywanie swojej metody `run` nie jest już zdolny do jej ponownego wykonania.
- Wykonywanie współbieżnie „niedużych” zadań w postaci osobnych obiektów wątków dedykowanych do ich realizacji jest **nieefektywne** oraz kłopotliwe programistycznie (np. konieczność synchronizacji w celu odebrania wyniku).
- W przeważającej liczbie praktycznych sytuacji dobrą praktyką jest odseparowanie zadań do wykonania od mechanizmów zarządzania wątkami.
- Dobrą praktyką jest również wykorzystywanie wątku do wykonywania wielu zadań po sobie (*reusing*).
- Koncepcja tej separacji i wielokrotnego użycia wątku opiera się na interfejsach: `Executor`, `ExecutorService`.

Definiowanie zadań do wykonania

- W Javie zadania dla wątków możemy definiować poprzez:
  - nadpisanie metody `run()` w klasie dziedziczącej z klasy `Thread`,
  - implementację interfejsu `Runnable`:
    - obiekt anonimowy,
    - obiekt klasy implementującej interfejs.

```
public interface Runnable {
   void run();
}
```

Executor, ExecutorService – wykonawcy

- Dedykowane implementacje interfejsów Wykonawców (nie programista) winny zajmować się zarządzaniem wątkami.
- Sposób, polityka tworzenia i uruchamiania wątków spoczywa na Wykonawcach (klasach implementujących interfejsy `Executor`, `ExecutorService`).
- Wykonawcy często utrzymują pule wątków pozwalającą na ponowne użycie wolnych wątków, a także na ewentualne limitowanie maksymalnej liczby wątków w puli.
- Wątki tworzone i zarządzane wewnętrznie przez Wykonawców są przygotowane do ponawiania swego działania związanego z realizacją kolejnego przydzielonego zadania.

Executor, ExecutorService – wykonawcy

```
1. public interface Executor {
2.   void execute(Runnable task);
3. }
4. public interface ExecutorService extends Executor, AutoCloseable
   {
5.   ...
6.   void shutdown();
7.   List<Runnable> shutdownNow();
8.   boolean awaitTermination(long timeout, TimeUnit unit)
9.         throws InterruptedException;
10.  ...
11. }
```

ExecutorService – opis metod

- `execute` – Przyjmuje podane zadanie do wykonania w przyszłości. Polecenie może zostać wykonane w nowym wątku, w wątku z puli lub w wątku wywołującym, zgodnie z implementacją `Executora`.
- `shutdown` – Inicjuje uporządkowane zamknięcie, podczas którego wykonywane są wcześniej przesłane zadania, ale żadne nowe zadania nie są już akceptowane. Wywołanie nie ma żadnego dodatkowego efektu, jeśli zostało już wywołane.
- `shutdownNow` – Próbuje zatrzymać wszystkie aktywnie wykonywane zadania, wstrzymuje przetwarzanie oczekujących zadań i zwraca listę zadań oczekujących na wykonanie. Ta metoda nie czeka na zakończenie wykonywania zadań.
- `awaitTermination` – Blokuje bieżący wątek do momentu zakończenia wykonywania wszystkich zadań po żądaniu zamknięcia, upłynięcia limitu czasu lub przerwania bieżącego wątku, w zależności od tego, co nastąpi wcześniej.

Rodzaje wykonawców (Executors)

- W Javie mamy do dyspozycji kilka rodzajów gotowych Wykonawców, fabrykowanych przez odpowiednie klasy implementujące interfejs `Executors` m.in.:
- `Executors.newSingleThreadExecutor()`: wykonawca uruchamiający podane mu zadania w jednym wątku (po kolei),
- `Executors.newFixedThreadPool()`: wykonawca, prowadzący pulę wątków o zadanym, maksymalnym rozmiarze,
- `Executors.newCachedThreadPool()`: wykonawca, prowadzący pulę wątków o dynamicznym rozmiarze,
- `Executors.newScheduledThreadPool()`: wykonawcy zarządzający tworzeniem i wykonaniem wątków w określonym czasie lub z określoną periodycznością.

Executors – przykład 1

```
1.  class MyTask implements Runnable {
2.    private String name;
3.    public MyTask(String name) {
4.      this.name = name;
5.    }
6.    public void run() {
7.      for (int i = 1; i <= 4; i++) {
8.        System.out.println(name + " " + i);
9.        Thread.yield();
10.     }
11.   }
12. }

13. public class Test {
14.   public static void main(String[] args) {
15.     Executor exec = Executors.newFixedThreadPool(2);
16.     for (int i = 1; i <= 4; i++) {
17.       exec.execute(new MyTask("Task " + i));
18.     }
19.   }
20. }
```

```
Task 1 1
Task 2 1
Task 1 2
Task 2 2
Task 1 3
Task 2 3
Task 1 4
Task 2 4
Task 3 1
Task 4 1
Task 3 2
Task 4 2
Task 3 3
Task 4 3
Task 3 4
Task 4 4
```

ExecutorService – przykład 2

```
1.  public static void main(String[] args) {
2.      ExecutorService exec = Executors.newFixedThreadPool(2);
3.      for (int i = 1; i <= 4; i++) {
4.        exec.execute(new Task("Task " + i));
5.      }
6.      Thread.yield();
7.      exec.shutdown();
8.      try {
9.        exec.execute(new Task("Task after shutdown"));
10.     } catch (RejectedExecutionException exc) {
11.       exc.printStackTrace();
12.     }
13.     try {
14.       exec.awaitTermination(5, TimeUnit.SECONDS);
15.     } catch(InterruptedException exc) { exc.printStackTrace(); }
16.     System.out.println("Terminated: " + exec.isTerminated());
17.   }
```

ExecutorService – przykład 2

```
1.  public static void main(String[] args) {
2.      ExecutorService exec = Executors.newFixedThreadPool(2);
3.      for (int i = 1; i <= 4; i++) {
4.        exec.execute(new Task("Task " + i));
5.      }
6.      Thr
7.      exe
8.      try
9.        e
10.     } c
11.
12.     }
13.     try
14.       e
15.     } c
16.     Sys
17.   }
```

```
java.util.concurrent.RejectedExecutionException
  at java.base/java.util.concurrent.ThreadPoolExecutor...
  ...
...
Task 1 4
Task 2 4
Task 3 1
Task 4 1
Task 3 2
Task 4 2
Task 3 3
Task 4 3
Task 3 4
Task 4 4
Terminated: true
```

Executors – przykład 3

```
1. class MyTask implements Runnable {
2.    ...
3.    public void run() {
4.      for (byte i = 0; i <= 128; i++) {
5.        System.out.println(name + " " + i);
6.        Thread.yield();
7.      }
8.    }
9. }
```

```
...
Task 2 28
Task 2 29
Task 2 30
Task 2 31
Task 2 32
Task 1 -63
Task 1 -62
Task 1 -61
Task 1 -60
Task 1 -59
Task 1 -58
...
...
```

`Interrupt` a kończenie wątków

- Przerwany wątek otrzymuje status przerwanego.
- W kodzie zadania należy poprzez sprawdzanie tego statusu (`isInterrupted()`) umożliwić zakończenie wykonywania metody `run()`.
- Pamiętać należy, że w sytuacjach gdy wątek jest uśpiony lub zablokowany z możliwością przerwania blokady
  - `join`, `sleep`, `wait` i jego odpowiedniki w java.util.concurrent,
  - przerywalne synchronizatory (interruptible locks),
  - przerywalne operacje we-wy (interruptible channels),
- to wymuszone kończenie zadań poprzez wywołanie `interrupt()` – powoduje zgłoszenie wyjątku `InterruptedException` i w obsłudze tego wyjątku należy zakończyć wykonanie kodu wątku.

Zwracanie wyniku zadania

- W celu umożliwienia zwrócenia wyników przez zadania ewentualnie sygnalizowania różnych wyjątków należy wykorzystać interfejsy:
  - `Future`, `Callable`
- `lub klasę`
  - `FutureTask`.

Zwracanie wyniku zadania

```
1. public interface Callable<V> {
2.   V call() throws Exception;
3. }

4. public interface Future<V> {
5.   ...
6.   boolean cancel(boolean mayInterruptIfRunning);
7.   boolean isCancelled();
8.   boolean isDone();
9.   V get() throws InterruptedException, ExecutionException;
10.  V get(long timeout, TimeUnit unit)
11.        throws InterruptedException, ExecutionException,
12.        TimeoutException;
13.  ...
14. }
```

- `cancel` – Próbuje anulować wykonywanie zadania związanego z tym wynikiem.
- `isCancelled` – Zwraca `true`, jeśli zadanie zostało anulowane zanim się zakończyło w sposób normalny.
- `isDone` – Zwraca `true`, jeśli zadanie zakończyło się. Zakończenie może być normalne lub w skutek wyjątku, lub anulowania. We wszystkich przypadkach wynik zwracany to `true`.
- `get` – Wstrzymuje bieżący wątek, jeśli konieczne, do zakończenia zadania i następnie zwraca wynik.

Zwracanie wyniku zadania

```
1. public class FutureTask<V> implements Runnable, Future<V> {
2.    ...
3.    public FutureTask(Callable<V> task) {...}
4.    public FutureTask(Runnable task, V result) {...}
5.    ...
6. }
```

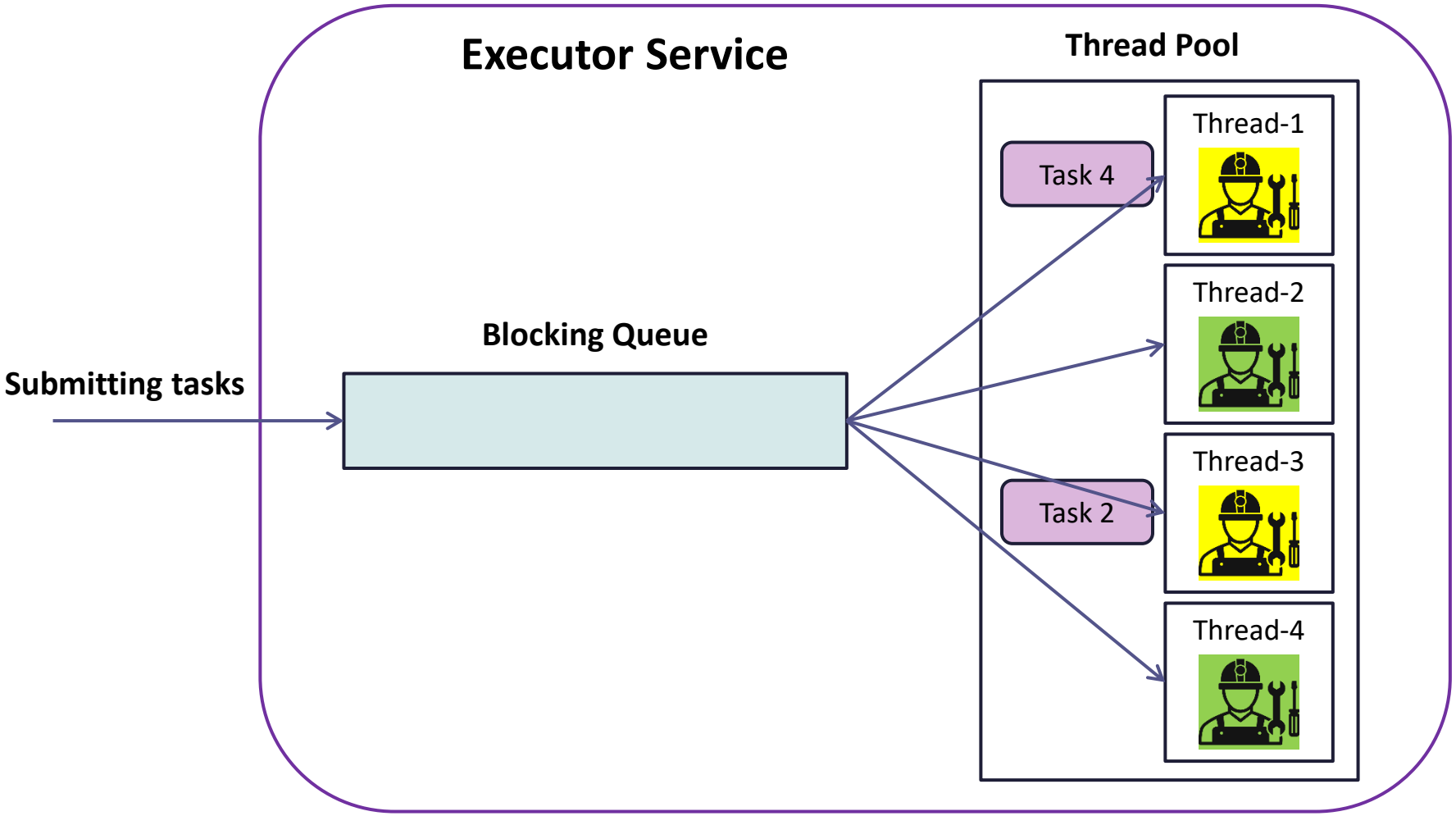
ExecutorService – metody dla zadań z wynikiem

```
1.  public interface ExecutorService extends Executor, AutoCloseable {
2.     ...
3.     <T> Future<T> submit(Callable<T> task);
4.     <T> Future<T> submit(Runnable task, T result);
5.     Future<?> submit(Runnable task);
6.     <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
7.           throws InterruptedException;
8.     <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks ,
9.           long timeout, TimeUnit unit)
10.              throws InterruptedException;
11.    <T> T invokeAny(Collection<? extends Callable<T>> tasks)
12.          throws InterruptedException, ExecutionException;
13.    <T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout,
14.          TimeUnit unit)
15.              throws InterruptedException, ExecutionException;
16.    ...
17. }
```

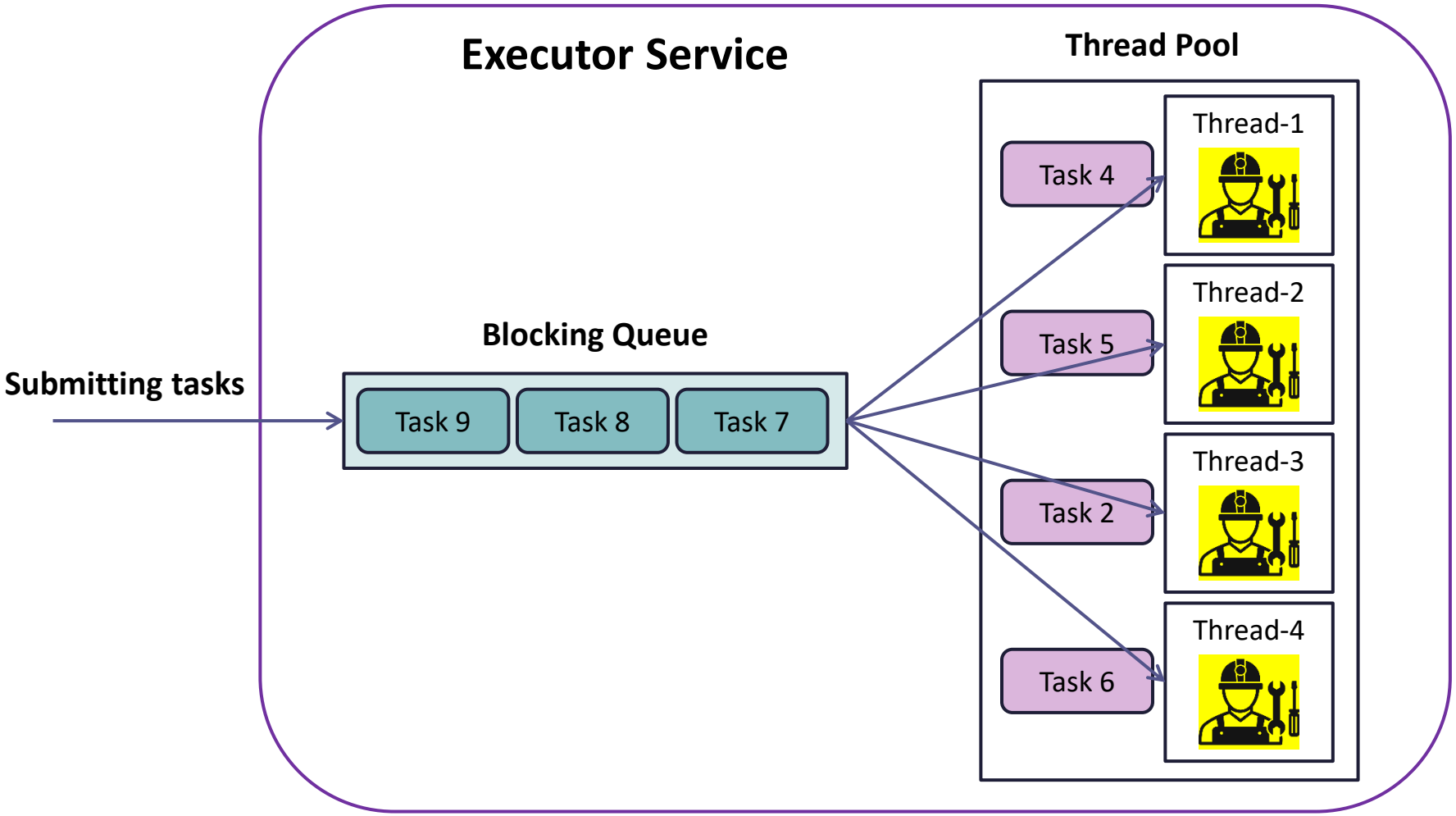
ExecutorService – metody dla zadań z wynikiem

- `submit` – Przesyła zadanie zwracające wynik do wykonania i zwraca wartość typu `Future` reprezentującą wynik zadania. Metoda `Future.get` zwróci wynik zadania po pomyślnym zakończeniu. Zadanie może być przekazane jako obiekt implementujący interfejsy `Runnable` albo `Callable`. Metoda nie blokuje wątku wywołującego.
- `invokeAll` – Przesyła zadania do wykonania (`Callable`) i zwraca ich wyniki w postaci listy obiektów `Future`. Wątek wywołujący jest wstrzymywany do czasu zakończenia wszystkich zadań (w sposób normalny lub nie).
- `invokeAny` – Przesyła zadania do wykonania i zwraca wynik pierwszego z nich zakończonego pomyślnie. Wątek wywołujący jest wstrzymywany do czasu zakończenia pomyślnie dowolnego zadania. Gdy żadne zadanie nie zakończy się pomyślnie, rzucany jest wyjątek `ExecutionException`.

ExecutorService – schemat funkcjonowania – 1



ExecutorService – schemat funkcjonowania – 2



Runnable vs Callable

- Interfejsy `Runnable` i `Callable` są do siebie podobne, gdyż oba służą do definiowania zadania do wykonania poprzez implementację odpowiedniej metody `run()` lub `call()`.
- Różnice:

Runnable	Callable
wykonywany przez dedykowany wątek lub ExecutorService	wykonywany **tylko** przez ExecutorService
nie zwraca wyniku	zwraca wynik
nie może wyrzucać wyjątku	może wyrzucać wyjątek

Zwracanie wyniku zadania – przykłady 1

```
1.  ExecutorService executorService = Executors.newSingleThreadExecutor();

2.  Future future1 = executorService.submit(new Runnable() {
3.      public void run() {
4.          System.out.println("Asynchronous Runnable 1");
5.      }
6.  });

7.  Future future2 = executorService.submit(new Runnable() {
8.      public void run() {
9.          System.out.println("Asynchronous Runnable 2");
10.     }
11. }, "Task2 Result");

12. Future future3 = executorService.submit(new Callable<String>(){
13.     public String call() throws Exception {
14.         System.out.println("Asynchronous Callable");
15.         return "Callable Result";
16.     }
17. });

18. future1.get();  //zwraca null, jeśli zakończyło się poprawnie.

19. System.out.println("future2.get() = " + future2.get() + " future3.get() = " +
20.     future3.get()););
```

Zwracanie wyniku zadania – przykłady 1 (lambda expr.)

```
1.  ExecutorService executorService = Executors.newSingleThreadExecutor();

2.  Future future1 = executorService.submit(() {
3.      System.out.println("Asynchronous Runnable 1");
4.  });
5.  Runnable run2 = () -> {
6.      System.out.println("Asynchronous Runnable 2");
7.  };
8.  Future future2 = executorService.submit(run2, "Task2 Result");

9.  Future future3 = executorService.submit(() -> {
10.     System.out.println("Asynchronous Callable");
11.     return "Callable Result";
12. });

13. future1.get();  //zwraca null, jeśli zakończyło się poprawnie.

14. System.out.println("future2.get() = " + future2.get() + " future3.get() = " +
15.     future3.get()););
```

Zwracanie wyniku zadania – przykłady 2

```
1.  ExecutorService executorService = Executors.newFixedThreadPool(3);

2.  Set<Callable<String>> callables = new HashSet<Callable<String>>();
3.  callables.add(new Callable<String>() {
4.      public String call() throws Exception {
5.          return "Task 1";
6.      }
7.  });
8.  callables.add(new Callable<String>() {
9.      public String call() throws Exception {
10.         return "Task 2";
11.     }
12. });
13. callables.add(() -> {
14.     return "Task 3";
15. });

16. String result = executorService.invokeAny(callables);
17. System.out.println("result = " + result);

18. List<Future<String>> futures = executorService.invokeAll(callables);

19. for(Future<String> future : futures){
20.     System.out.println("future.get = " + future.get());
21. }

22. executorService.shutdown();
```