Wielowątkowość

dzień 1

v3.0

Plan

1. Wielowątkowość - podstawy
2. Tworzenie wątków
3. InterruptedException
4. Wątki Deamon
5. Cykl życia wątków
6. Wartości lokalne i synchronizacja
7. Synchronizacja
8. Problemy z synchronizacją
9. Komunikacja między wątkami

Wielowątkowość - podstawy

Podstawowe pojęcia

Jednozadaniowość - tylko jeden proces uruchomiony, tylko aktywny proces wykonuje pracę

Wielozadaniowość - uruchamianie kilku procesów na jednej maszynie, każdy może wykonywać pracę

Wielowątkowość - działanie kilku wątków w tym samym czasie i tym samym procesie(ale nie koniecznie równolegle)

Przetwarzanie współbieżne - proces podzielony jest na małe części i jego wykonywanie przebiega równolegle

Zadania, które mogą być wykonywane w wątkach, to np. odczytanie danych z pliku, pobieranie wartości z bazy danych, zapisanie komunikatu do pliku logu

Proces

Przykład procesów oraz cechy szczególne:

- procesem może być program, aplikacja, narzędzie systemowe
- występują również procesy systemu operacyjnego
- z reguły procesy są samodzielne
- z reguły poszczególne procesy nie wiedzą o sobie
- procesy mają swoją alokację pamięci

Wątek

Wątki które mogą działać w procesach mają pewne cechy szczególne:

- dzielą pamięć i zasoby systemowe z innymi wątkami tego samego procesu
- są częścią tego samego procesu
- nie mogą istnieć poza procesem
- nazywane lekkimi procesami
- jeżeli zadanie w procesie można wykonać równolegle do innych zadań to wątki pozwalają na przyspieszenie jego wykonywania

Architektura wielordzeniowa

Procesor wielordzeniowy - procesor posiadający więcej niż jeden rdzeń fizyczny. Technologia ta ma na celu zwiększenie wydajności procesora, zmniejszenie zużycia energii i bardziej efektywnego jednoczesnego przetwarzania wielu zadań.

Procesor ściśle współpracuje z pamięcią operacyjną, która znajduje się na płycie głównej i działa wolniej niż rdzeń procesora. W celu uniknięcia wolnych taktów procesora, przez które CPU czeka na dane z pamięci, wewnątrz rdzenia umieszcza się szybką pamięć podręczną – **Cache**.

Architektura wielordzeniowa

Procesory mogą mieć kilka poziomów Cache:

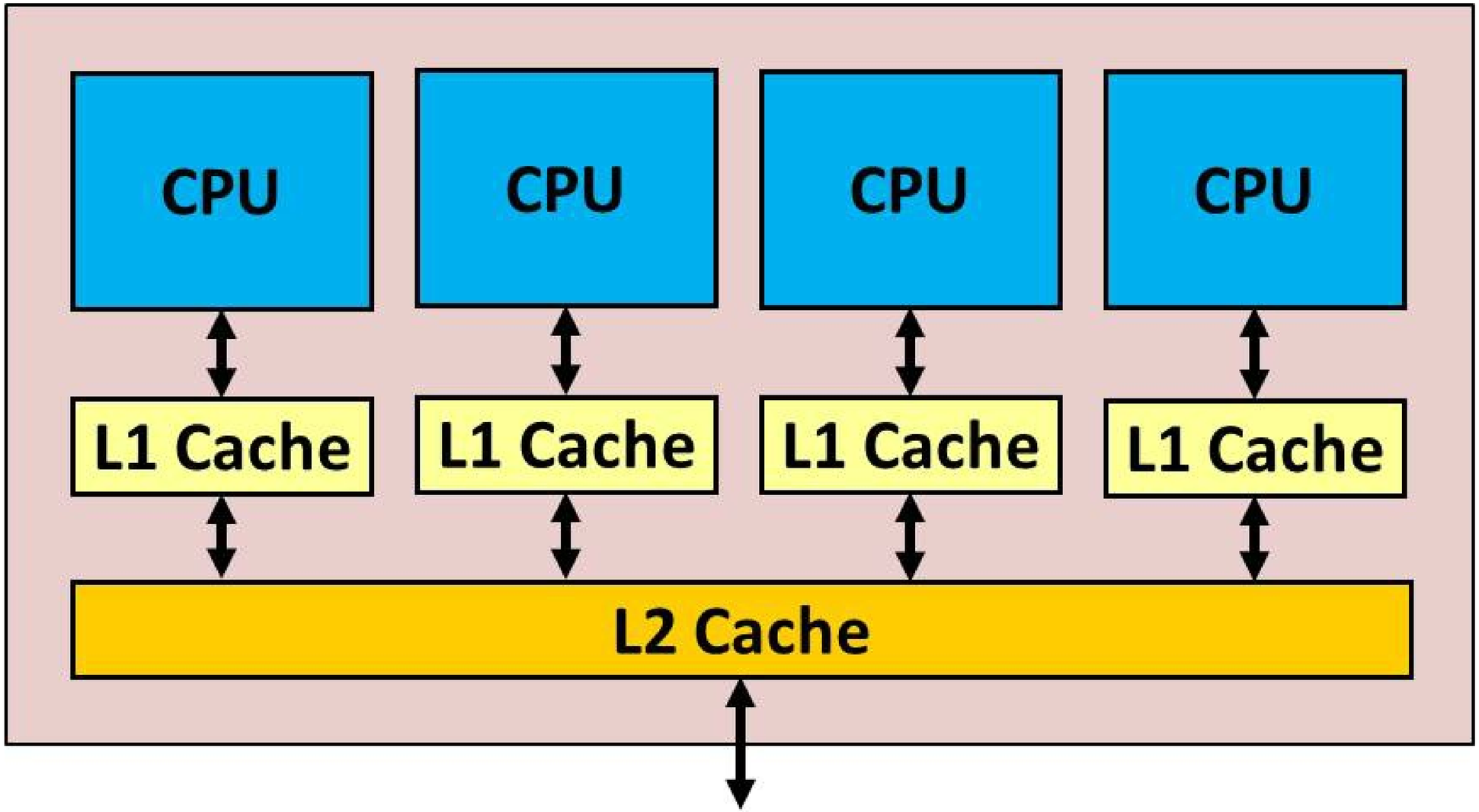
Level 1 (L1) – pamięć Cache jest zintegrowana z rdzeniem procesora.

Level 2 (L2) – pamięć zintegrowana z rdzeniem, co umożliwia wymianę danych z pełną prędkością rdzenia.

Level 3 (L3) – najczęściej montowana w procesorach do zastosowań serwerowych. Umieszczany na płycie głównej lub wewnątrz rdzenia procesora, zwiększa wydajność i trafność pobranych danych.

Im większy rozmiar pamięci podręcznej Cache, tym szybsza praca procesora podczas odczytu danych z RAM. Przy dużej ilości Cache, procesor komunikuje się tylko z Cache.

Architektura wielordzeniowa



Problemy systemów wielowątkowych

Odczyt danych pomiędzy procesorem a pamięcią zajmuje czas

Aby zredukować czas potrzebny na przenoszenie danych stosuje się cache

Obecnie procesory wielordzeniowe mają nawet kilka poziomów cache np. L1, L2, L3

Każdy z poziomów cache może zawierać dane, nawet te przechowywane w cache dla innych rdzeni

Problemy systemów wielowątkowych

Może wystąpić sytuacja w której jeden wątek zmieni dane które trafią do jednego cache, w przypadku odczytu danych przez inny wątek z innego cache pojawią się stare dane

Może wystąpić też sytuacja, że podczas zapisu danych inny wątek zacznie czytać z cache i dane też będą się różniły

Można to rozwiązać synchronizacją, gdzie przy działaniu poprzez jeden wątek wszystkie cache są aktualizowane lub tylko jeden wątek zapisuje a reszta odczytuje

Tworzenie wątków

Tworzenie wątków

W kilku przykładach na najbliższych slajdach poznamy kilka metod na tworzenie wątków w Javie

Pierwszym z przykładów będzie rozszerzenie klasy **`Thread`** oraz nadpisanie metody **`run()`** tej klasy

Kolejne przykłady pokażą także praktyczne zastosowanie utworzonych klas

Tworzenie wątków

```
public class ThreadExample extends Thread{
    @Override
    public void run() {
        Random r = new Random();
        int rand = r.nextInt(2000) + 1000;
        for (int i = 0; i < 3; i++) {
            System.out.println(Thread.currentThread().getName() + " - " + i);
            try {
                Thread.sleep(rand);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Tworzenie wątków

```
public class ThreadExample extends Thread{
    @Override
    public void run() {
        Random r = new Random();
        int rand = r.nextInt(2000) + 1000;
        for (int i = 0; i < 3; i++) {
            System.out.println(Thread.currentThread().getName() + " - " + i);
            try {
                Thread.sleep(rand);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Rozszerzamy klasę Thread

Tworzenie wątków

```
public class ThreadExample extends Thread{
    @Override
    public void run() {
        Random r = new Random();
        int rand = r.nextInt(2000) + 1000;
        for (int i = 0; i < 3; i++) {
            System.out.println(Thread.currentThread().getName() + " - " + i);
            try {
                Thread.sleep(rand);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Nadpisujemy metodę run()

Tworzenie wątków

Użycie utworzonej klasy ThreadExample(w metodzie main())

```
ThreadExample te1 = new ThreadExample();
ThreadExample te2 = new ThreadExample();
ThreadExample te3 = new ThreadExample();
te1.start();
te2.start();
te3.start();
```

Tworzenie wątków

Użycie utworzonej klasy ThreadExample(w metodzie main())

```
ThreadExample te1 = new ThreadExample();
ThreadExample te2 = new ThreadExample();
ThreadExample te3 = new ThreadExample();
te1.start();
te2.start();
te3.start();
```

Utworzenie obiektów wcześniej stworzonej klasy ThreadExample

Tworzenie wątków

Użycie utworzonej klasy ThreadExample(w metodzie main())

```
ThreadExample te1 = new ThreadExample();
ThreadExample te2 = new ThreadExample();
ThreadExample te3 = new ThreadExample();
te1.start();
te2.start();
te3.start();
```

Utworzenie obiektów wcześniej stworzonej klasy ThreadExample

Uruchomienie wątków za pomocą metod start()

Tworzenie wątków

Przykładowy wynik działania w konsoli(za każdym razem inny):

```
Thread-0 - 0
Thread-1 - 0
Thread-2 - 0
Thread-2 - 1
Thread-1 - 1
Thread-0 - 1
Thread-2 - 2
Thread-1 - 2
Thread-0 - 2
```

Tworzenie wątków

Jest również możliwość utworzenia klasy anonimowej która będzie użyta tylko na potrzeby jednego obiektu:

```
Thread thread = new Thread(){
    public void run(){
      System.out.println("Thread is running");
    }
};
thread.start();
```

Tworzenie wątków

Jest również możliwość utworzenia klasy anonimowej która będzie użyta tylko na potrzeby jednego obiektu:

```
Thread thread = new Thread(){
    public void run(){
      System.out.println("Thread is running");
    }
};
thread.start();
```

Dla zmiennej thread od razu tworzymy nowy obiekt klasy Thread

Tworzenie wątków

Jest również możliwość utworzenia klasy anonimowej która będzie użyta tylko na potrzeby jednego obiektu:

```
Thread thread = new Thread(){
    public void run(){
      System.out.println("Thread is running");
    }
};
thread.start();
```

Dla zmiennej thread od razu tworzymy nowy obiekt klasy Thread

W definicji anonimowej klasy od razu nadpisujemy metodę run()

Tworzenie wątków

Jest również możliwość utworzenia klasy anonimowej która będzie użyta tylko na potrzeby jednego obiektu:

```
Thread thread = new Thread(){
    public void run(){
      System.out.println("Thread is running");
    }
};
thread.start();
```

Dla zmiennej thread od razu tworzymy nowy obiekt klasy Thread

W definicji anonimowej klasy od razu nadpisujemy metodę run()

Na obiekcie thread wywołujemy metodę start() aby uruchomić działanie metody run()

Tworzenie wątków

Innym sposobem jest implementacja interfejsu Runnable

Implementowanie interfejsu Runnable to preferowany sposób tworzenia wątków, ponieważ w przyszłości nie blokujemy możliwości dziedziczenia.

```
public class MyRunnable implements Runnable {
    public void run(){
       System.out.println("Thread is running");
    }
}
Thread thread = new Thread(new MyRunnable());
thread.start();
```

Tworzenie wątków

Innym sposobem jest implementacja interfejsu Runnable

Implementowanie interfejsu Runnable to preferowany sposób tworzenia wątków, ponieważ w przyszłości nie blokujemy możliwości dziedziczenia.

```
public class MyRunnable implements Runnable {
    public void run(){
       System.out.println("Thread is running");
    }
}
Thread thread = new Thread(new MyRunnable());
thread.start();
```

Implementacja interfejsu Runnable

Tworzenie wątków

Innym sposobem jest implementacja interfejsu Runnable

Implementowanie interfejsu Runnable to preferowany sposób tworzenia wątków, ponieważ w przyszłości nie blokujemy możliwości dziedziczenia.

```
public class MyRunnable implements Runnable {
    public void run(){
        System.out.println("Thread is running");
    }
}
Thread thread = new Thread(new MyRunnable());
thread.start();
```

Nadpisujemy metodę run()

Tworzenie wątków

Innym sposobem jest implementacja interfejsu Runnable

Implementowanie interfejsu Runnable to preferowany sposób tworzenia wątków, ponieważ w przyszłości nie blokujemy możliwości dziedziczenia.

```
public class MyRunnable implements Runnable {
    public void run(){
       System.out.println("Thread is running");
    }
}
Thread thread = new Thread(new MyRunnable());
thread.start();
```

Tworzymy nowy obiekt klasy Thread podając jako argument do konstruktora utworzoną klasę MyRunnable

Tworzenie wątków

Innym sposobem jest implementacja interfejsu Runnable

Implementowanie interfejsu Runnable to preferowany sposób tworzenia wątków, ponieważ w przyszłości nie blokujemy możliwości dziedziczenia.

```
public class MyRunnable implements Runnable {
    public void run(){
       System.out.println("Thread is running");
    }
}
Thread thread = new Thread(new MyRunnable());
thread.start();
```

Wywołujemy metodę start() na obiekcie thread aby uruchomić działanie metody run()

Tworzenie wątków

Innym sposobem jest implementacja interfejsu Runnable

Implementowanie interfejsu Runnable to preferowany sposób tworzenia wątków, ponieważ w przyszłości nie blokujemy możliwości dziedziczenia.

```
public class MyRunnable implements Runnable {
    public void run(){
       System.out.println("Thread is running");
    }
}
Thread thread = new Thread(new MyRunnable());
thread.start();
```

Częstym błędem jest wywoływanie metody **run()** zamiast **start()** aby uruchomić wątek - należy zwrócić na to uwagę

Tworzenie wątków

Możliwe jest również wykonanie anonimowej implementacji interfejsu Runnable

```
Runnable myRunnable = new Runnable(){
      public void run(){
         System.out.println("Runnable is running");
      }
};
Thread thread = new Thread(myRunnable);
thread.start();
```

Tworzenie wątków

Dobrą praktyką jest również nadawanie nazw wątkom, tak aby można było odnosić się do nich

W przypadku implementacji interfejsu Runnable dodajemy nazwę po przekazaniu naszej klasy w konstruktorze wątku:

```
public class MyRunnable implements Runnable {
    public void run(){
       System.out.println("Thread is running");
    }
}
Thread thread = new Thread(new MyRunnable(), "CustomThreadName");
thread.start();
```

Tworzenie wątków

Dobrą praktyką jest również nadawanie nazw wątkom, tak aby można było odnosić się do nich

W przypadku implementacji interfejsu Runnable dodajemy nazwę po przekazaniu naszej klasy w konstruktorze wątku:

```
public class MyRunnable implements Runnable {
    public void run(){
       System.out.println("Thread is running");
    }
}
Thread thread = new Thread(new MyRunnable(), "CustomThreadName");
thread.start();
```

W tym przypadku wątek będzie miał nazwę "CustomThreadName"

Tworzenie wątków

W przypadku tworzenia klasy anonimowej dodajemy nazwę jako parametr konstruktora klasy Thread:

```
Thread thread = new Thread("NewThreadName") {
      public void run(){
        System.out.println("Thread is running");
      }
};
thread.start();
```

Tworzenie wątków

W przypadku tworzenia klasy anonimowej dodajemy nazwę jako parametr konstruktora klasy Thread:

```
Thread thread = new Thread("NewThreadName") {
      public void run(){
        System.out.println("Thread is running");
      }
};
thread.start();
```

W tym przypadku wątek będzie miał nazwę "NewThreadName"

Tworzenie wątków

Przydatną metodą jest metoda **getName()** za pomocą której możemy pobrać ustawioną wcześniej nazwę wątku:

```
Thread thread = new Thread("New Thread") {
      public void run(){
        System.out.println("Name of thread: " + getName());
      }
};
thread.start();
System.out.println(thread.getName());
```

Tworzenie wątków

Przydatną metodą jest metoda **getName()** za pomocą której możemy pobrać ustawioną wcześniej nazwę wątku:

```
Thread thread = new Thread("New Thread") {
      public void run(){
        System.out.println("Name of thread: " + getName());
      }
};
thread.start();
System.out.println(thread.getName());
```

Użycie metody getName() w metodzie run()

Tworzenie wątków

Przydatną metodą jest metoda **getName()** za pomocą której możemy pobrać ustawioną wcześniej nazwę wątku:

```
Thread thread = new Thread("New Thread") {
      public void run(){
        System.out.println("Name of thread: " + getName());
      }
};
thread.start();
System.out.println(thread.getName());
```

Użycie metody getName() na obiekcie klasy Thread

Interrupted Exception

Interrupted Exception

W naszym kodzie w wielu miejscach stosujemy obsługę błędu **`InterruptedException`**.
Porównajmy dwa poniższe przypadki:

```
Thread loop1 = new Thread(
  new Runnable() {
    @Override
    public void run() {
      while (true) { /* some work */  }
    }
  }
);
loop1.start();
```

```
Thread loop2 = new Thread(
  new Runnable() {
    @Override
    public void run() {
      while(true) {
        if(Thread.interrupted()) {
          break;
        }
        // some work
      }
    }
  }
);
loop2.start();
loop2.interrupt();
```

Interrupted Exception

W naszym kodzie w wielu miejscach stosujemy obsługę błędu **`InterruptedException`**.
Porównajmy dwa poniższe przypadki:

```
Thread loop1 = new Thread(
  new Runnable() {
    @Override
    public void run() {
      while (true) { /* some work */  }
    }
  }
);
loop1.start();
```

Uruchomienie wątku loop1

```
Thread loop2 = new Thread(
  new Runnable() {
    @Override
    public void run() {
      while(true) {
        if(Thread.interrupted()) {
          break;
        }
        // some work
      }
    }
  }
);
loop2.start();
loop2.interrupt();
```

Interrupted Exception

W naszym kodzie w wielu miejscach stosujemy obsługę błędu **`InterruptedException`**.
Porównajmy dwa poniższe przypadki:

```
Thread loop1 = new Thread(
  new Runnable() {
    @Override
    public void run() {
      while (true) { /* some work */  }
    }
  }
);
loop1.start();
```

Wątku loop1 nie można zatrzymać inaczej niż przez kombinację ctrl+c

```
Thread loop2 = new Thread(
  new Runnable() {
    @Override
    public void run() {
      while(true) {
        if(Thread.interrupted()) {
          break;
        }
        // some work
      }
    }
  }
);
loop2.start();
loop2.interrupt();
```

Interrupted Exception

W naszym kodzie w wielu miejscach stosujemy obsługę błędu **`InterruptedException`**.
Porównajmy dwa poniższe przypadki:

```
Thread loop1 = new Thread(
  new Runnable() {
    @Override
    public void run() {
      while (true) { /* some work */  }
    }
  }
);
loop1.start();
```

Uruchomienie wątku loop2

```
Thread loop2 = new Thread(
  new Runnable() {
    @Override
    public void run() {
      while(true) {
        if(Thread.interrupted()) {
          break;
        }
        // some work
      }
    }
  }
);
loop2.start();
loop2.interrupt();
```

Interrupted Exception

W naszym kodzie w wielu miejscach stosujemy obsługę błędu **`InterruptedException`**.
Porównajmy dwa poniższe przypadki:

```
Thread loop1 = new Thread(
  new Runnable() {
    @Override
    public void run() {
      while (true) { /* some work */  }
    }
  }
);
loop1.start();
```

Wykonanie metody interrupt() przerywającej pracę wątku loop2

```
Thread loop2 = new Thread(
  new Runnable() {
    @Override
    public void run() {
      while(true) {
        if(Thread.interrupted()) {
          break;
        }
        // some work
      }
    }
  }
);
loop2.start();
loop2.interrupt();
```

Interrupted Exception

W naszym kodzie w wielu miejscach stosujemy obsługę błędu **`InterruptedException`**.
Porównajmy dwa poniższe przypadki:

```
Thread loop1 = new Thread(
  new Runnable() {
    @Override
    public void run() {
      while (true) { /* some work */  }
    }
  }
);
loop1.start();
```

Za pomocą metody Thread.interrupted() sprawdzamy czy nastąpiło przerwanie pracy wątku

```
Thread loop2 = new Thread(
  new Runnable() {
    @Override
    public void run() {
      while(true) {
        if(Thread.interrupted()) {
          break;
        }
        // some work
      }
    }
  }
);
loop2.start();
loop2.interrupt();
```

Interrupted Exception

Podobnie np. w poznanej metodzie **`Thread.sleep()`** w jej definicji **`public static void sleep(long milisec) throws InterruptedException`** metoda wyrzuca błąd InterruptedException, sygnalizując przerwanie wykonywania metody. W ciele metody **`Thread.sleep()`** następuje sprawdzenie podobne do poniższego:

```
public static void sleep(long milisec)
  throws InterruptedException {
    while(true) {
      if(Thread.interrupted()) {
        throw new InterruptedException();
      }
      // still waiting
    }
}
```

Interrupted Exception

W tym przypadku wyrzucenie błędu **`InterruptedException`** poprzez **`throw new InterruptedException();`** spowoduje zatrzymanie pracy tej metody.

Co warte zapamiętania błędy typu **`InterruptedException`** są błędami typu **Checked exceptions**, co jak już wiemy z poprzednich modułów wymusza na nas obsługę tego typu błędów przy wywołaniu metod wyrzucających takie właśnie błędy. Najprościej to zrobić poprzez otoczenie danej metody blokiem **`try-catch`**:

```
try {
      Thread.sleep(1000);
} catch (InterruptedException ex) {
  // handle thread stopping
}
```

Wątki deamon

Wątki Deamon

Normalne wątki

- JVM czeka na zakończenie działania pracy wątków. Nie zakończy działania dopóki wszystkie wątki nie zakończą pracy.
- Zwykłe wątki nie działają w tle.
- Zwykłe wątki mają wysoki priorytet.
- Są tworzone przez aplikację
- Są zaprojektowane by wykonać pewne specyficzne zadania.
- JVM nie przerwie działania wątków, będzie czekał dopóki wątek ewentualnie sam przerwie swoje działanie.

Wątki Daemon

- JVM nie będzie czekał na zakończenie pracy wątków typu daemon. JVM zakończy ich działanie jak tylko swoje działanie zakończą normalne wątki.
- Wątki typu daemon działają w tle.
- Mają niski priorytet.
- W większości przypadków są tworzone przez JVM.
- Są zaprojektowane by wspomagać pracę normalnych wątków.
- JVM przerwie działanie wątków typu daemon kiedy zwykłe wątki zakończą swoje działanie.

Wątki Deamon

Możemy sprawdzić czy wątek jest typu daemon za pomocą metody **isDaemon()** na obiekcie klasy Thread

```
Thread thread = new Thread("DaemonThread") {
    public void run(){
        System.out.println("Name of thread: " + getName());
    }
};
thread.start();
System.out.println(thread.isDaemon());
```

Wątki Deamon

Możemy sprawdzić czy wątek jest typu daemon za pomocą metody **isDaemon()** na obiekcie klasy Thread

```
Thread thread = new Thread("DaemonThread") {
    public void run(){
        System.out.println("Name of thread: " + getName());
    }
};
thread.start();
System.out.println(thread.isDaemon());
```

Wywołanie metody isDaemon() która zwróci prawdę lub fałsz

Wątki Deamon

Możemy sprawdzić czy wątek jest typu daemon za pomocą metody **isDaemon()** na obiekcie klasy Thread

```
Thread thread = new Thread("DaemonThread") {
    public void run(){
        System.out.println("Name of thread: " + getName());
    }
};
thread.start();
System.out.println(thread.isDaemon());
```

W tym przypadku będzie to fałsz

Wątki Deamon

Aby ustawić typ wątku jako daemon korzystamy z metody **setDeamon()** na obiekcie klasy Thread przed użyciem metody start()

Uwaga! Metoda start() może być użyta tylko raz, ponowne wywołanie spowoduje błąd **`IllegalThreadStateException`**

```
Thread thread = new Thread("DaemonThread") {
    public void run(){
        System.out.println("Name of thread: " + getName());
    }
};
thread.setDaemon(true);
thread.start();
System.out.println(thread.isDaemon());
```

Wątki Deamon

Aby ustawić typ wątku jako daemon korzystamy z metody **setDeamon()** na obiekcie klasy Thread przed użyciem metody start()

Uwaga! Metoda start() może być użyta tylko raz, ponowne wywołanie spowoduje błąd **`IllegalThreadStateException`**

```
Thread thread = new Thread("DaemonThread") {
    public void run(){
        System.out.println("Name of thread: " + getName());
    }
};
thread.setDaemon(true);
thread.start();
System.out.println(thread.isDaemon());
```

Ustawiamy obiekt thread jako daemon podając jako argument metody setDaemon() prawdę

Wątki Deamon

Aby ustawić typ wątku jako daemon korzystamy z metody **setDeamon()** na obiekcie klasy Thread przed użyciem metody start()

Uwaga! Metoda start() może być użyta tylko raz, ponowne wywołanie spowoduje błąd **`IllegalThreadStateException`**

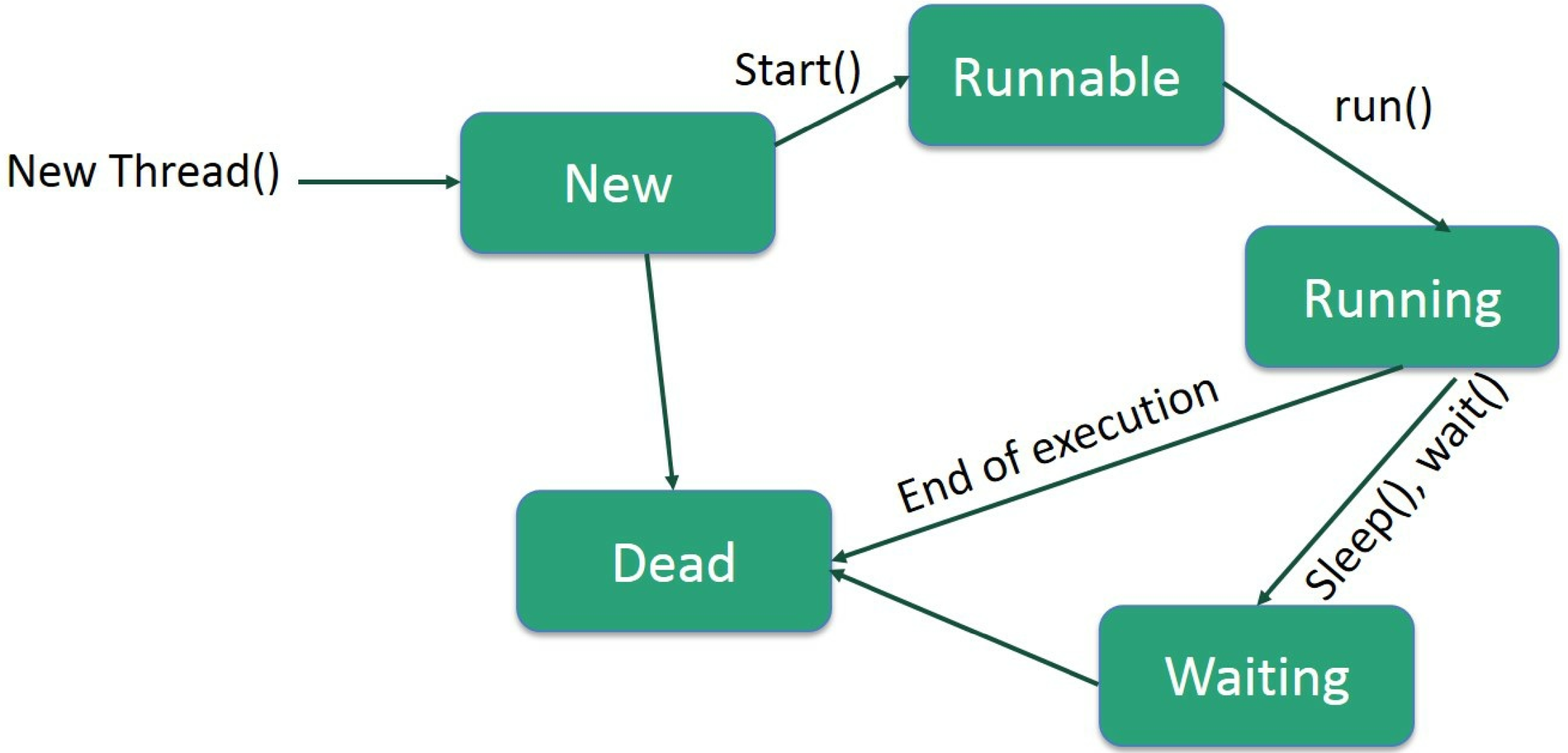
```
Thread thread = new Thread("DaemonThread") {
    public void run(){
        System.out.println("Name of thread: " + getName());
    }
};
thread.setDaemon(true);
thread.start();
System.out.println(thread.isDaemon());
```

Wywołanie metody isDaemon() zwróci w tym przypadku prawdę

Zadania

Cykl życia wątków

Cykl życia wątków - schemat ogólny



Usypianie wątków

Nowy wątek wykonuje metodę **`run()`** i wątek ten istnieje dopóki nie opuści metody run()

Mamy możliwość uśpienia bieżącego wątku, czyli tego, który jest w danym momencie wykonywany

Używamy w tym celu metody **Thread.sleep(long milisec)**, a jako argument podajemy liczbę milisekund na którą wątek ma zostać uśpiony. Daje możliwość innym wątkom do wykonania zadań.

Usypianie wątków

Po użyciu metody sleep() zmieniany jest stan wątku z Runnable na **Waiting**, natomiast upłynięciu czasu określonego jako argument metody sleep() z powrotem zmieniany jest stan wątku z Waiting na Runnable

Metoda Thread.sleep() jest statyczna więc działa na konkretnym wątku, nie możemy uśpić innego wątku

Można też użyć Enum TimeUnit:

- **`TimeUnit.SECONDS.sleep(10)`** aby uśpić wątek na 10 sekund
- lub np. **`TimeUnit.MINUTES.sleep(2)`** aby uśpić wątek na 2 minuty

Sprawdzanie czy wątek zakończył działanie

Aby sprawdzić czy wątek zakończył swoje działanie możemy użyć metody **isAlive()**

Jeżeli wątek nie zakończy swojego działania będzie w stanie Running w przeciwnym wypadku przyjmie stan Dead, według wcześniej przedstawionego schematu cyklu życia wątku.

```
Thread t = Thread.currentThread();
System.out.println("status = " + t.isAlive());
```

Sprawdzanie czy wątek zakończył działanie

Aby sprawdzić czy wątek zakończył swoje działanie możemy użyć metody **isAlive()**

Jeżeli wątek nie zakończy swojego działania będzie w stanie Running w przeciwnym wypadku przyjmie stan Dead, według wcześniej przedstawionego schematu cyklu życia wątku.

```
Thread t = Thread.currentThread();
System.out.println("status = " + t.isAlive());
```

Wywołanie metody isAlive() na obiekcie klasy Thread

Metoda join()

Za pomocą metody **join()** można spowodować aby inne wątki oczekiwały na zakończenie działania aktualnego wątku. Metoda join() wywołuje w tle metodę isAlive() aby sprawdzić stan danego wątku, a po jego zakończeniu wysyła w tle powiadomienie za pomocą metody notify() którą poznamy w późniejszym etapie

```
Thread t1 = new Thread(new MyClass(), "t1");
Thread t2 = new Thread(new MyClass(), "t2");
t1.start();
try {
        t1.join();
} catch (InterruptedException ex) {
        ex.printStackTrace();
}
t2.start();
```

Metoda join()

Za pomocą metody **join()** można spowodować aby inne wątki oczekiwały na zakończenie działania aktualnego wątku. Metoda join() wywołuje w tle metodę isAlive() aby sprawdzić stan danego wątku, a po jego zakończeniu wysyła w tle powiadomienie za pomocą metody notify() którą poznamy w późniejszym etapie

```
Thread t1 = new Thread(new MyClass(), "t1");
Thread t2 = new Thread(new MyClass(), "t2");
t1.start();
try {
        t1.join();
} catch (InterruptedException ex) {
        ex.printStackTrace();
}
t2.start();
```

Użycie metody join() na obiekcie t1 klasy Thread

Metoda join()

Za pomocą metody **join()** można spowodować aby inne wątki oczekiwały na zakończenie działania aktualnego wątku. Metoda join() wywołuje w tle metodę isAlive() aby sprawdzić stan danego wątku, a po jego zakończeniu wysyła w tle powiadomienie za pomocą metody notify() którą poznamy w późniejszym etapie

```
Thread t1 = new Thread(new MyClass(), "t1");
Thread t2 = new Thread(new MyClass(), "t2");
t1.start();
try {
        t1.join();
} catch (InterruptedException ex) {
        ex.printStackTrace();
}
t2.start();
```

Zanim swoje działanie rozpoczną inne wątki działanie musi zakończyć wątek t1

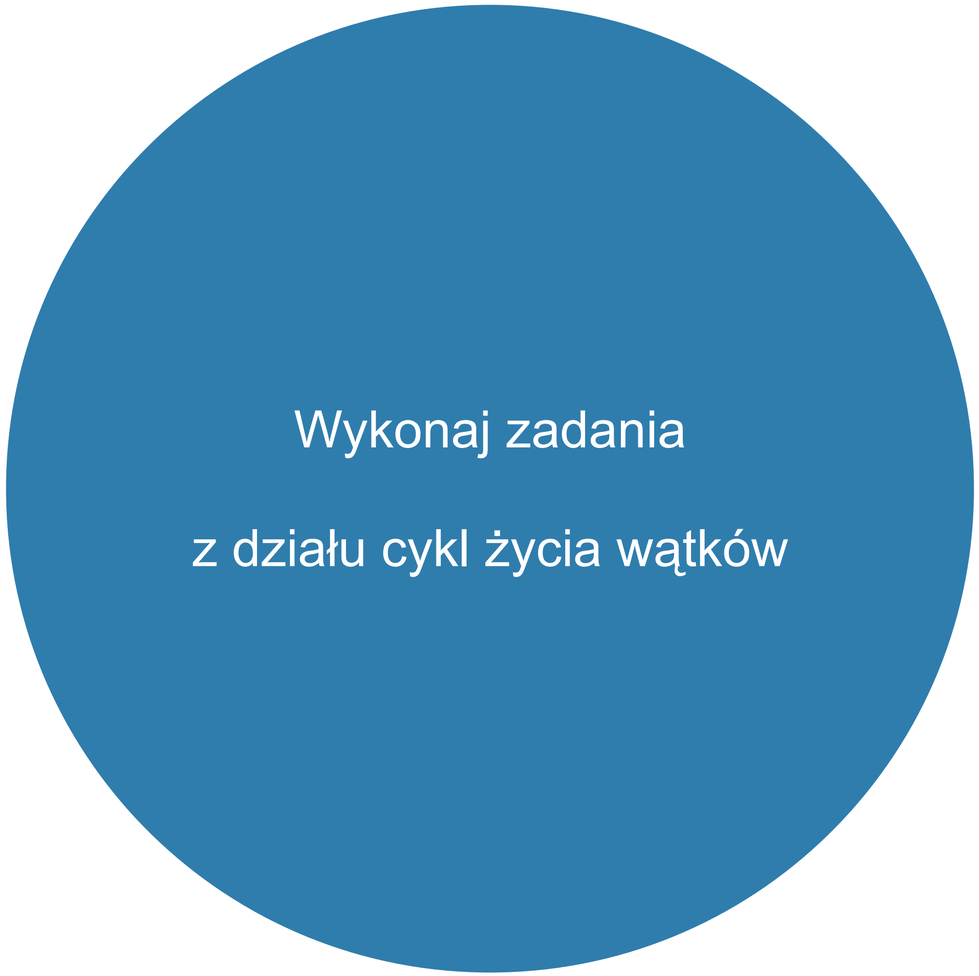
Metoda join()

Za pomocą metody **join()** można spowodować aby inne wątki oczekiwały na zakończenie działania aktualnego wątku. Metoda join() wywołuje w tle metodę isAlive() aby sprawdzić stan danego wątku, a po jego zakończeniu wysyła w tle powiadomienie za pomocą metody notify() którą poznamy w późniejszym etapie

```
Thread t1 = new Thread(new MyClass(), "t1");
Thread t2 = new Thread(new MyClass(), "t2");
t1.start();
try {
        t1.join();
} catch (InterruptedException ex) {
        ex.printStackTrace();
}
t2.start();
```

Wątek t2 zostanie uruchomiony dopiero po zakończeniu działania wątku t1

Zadania



Wartości lokalne i synchronizacja

Bezpieczeństwo kodu wielowątkowego

Bezpieczeństwo kodu wielowątkowego - pod tym pojęciem rozumiemy że kod poprawnie wykonuje się w wielowątkowym środowisku

Co powoduje że kod może być niebezpieczny w pracy z wieloma wątkami:

- każdy z wątków ma swój stan i wątki podczas wykonywania zadań zmieniają ten stan
- pracujemy z danymi które inne wątki modyfikowały lub mogą modyfikować

Zasoby jakie są dzielone:

- nie statyczne zmienne
- referencje do obiektów
- zawartość kolekcji/tablic
- gdzie dane są widoczne dla więcej niż jednego wątku

Synchronizacja danych

Java Memory Model - jest to model, który opisuje, jak wątki w języku programowania Java działają na siebie poprzez pamięć

Kolejność działania wątków:

- w kodzie jednowątkowym tylko jedna możliwość
- w wielowątkowym kodzie zależy od:
  - JVM scheduler-a
  - procesora
  - interakcji pomiędzy wątkami

Synchronizacja danych

Podczas działania programu w środowisku wielowątkowym siłą rzeczy następuje współdzielenie danych, które mogą być zmodyfikowane np. zmienne, tablice itp.

Jest to tzw. **"wyścig danych"**, aby temu zapobiec stosuje się synchronizację danych.

Trzeba więc wziąć pod uwagę, że kod który działał poprawnie w środowisku jednowątkowym może nie działać poprawnie w środowisku wielowątkowym

ThreadLocal

ThreadLocal to konstrukcja za pomocą której możemy operować zmiennymi i ich wartościami które dotyczą poszczególnych wątków.

Wartości przechowywane w ThreadLocal są globalne dla danego wątku, oznacza to że są osiągalne z każdego miejsca w danym wątku.

Wartości które przechowuje ThreadLocal są unikalnymi wartościami dla danego wątku. Tak jak w przypadku zwykłych obiektów i ich atrybutów gdzie każdy obiekt może posiadać różne wartości danych atrybutów, tak samo w różnych wątkach zmienne mogą mieć różne wartości.

ThreadLocal

Wartości dla konkretnej ThreadLocal w wątku możemy ustawiać za pomocą metody **`set()`** oraz pobierać za pomocą metody **`get()`**. Aby usunąć wartość ze zmiennej ThreadLocal używamy metody **`remove()`**

```
private ThreadLocal<String> myThreadLocal = new ThreadLocal<String>();
myThreadLocal.set("Hello ThreadLocal");
String threadLocalValue = myThreadLocal.get();
```

ThreadLocal

Wartości dla konkretnej ThreadLocal w wątku możemy ustawiać za pomocą metody **`set()`** oraz pobierać za pomocą metody **`get()`**. Aby usunąć wartość ze zmiennej ThreadLocal używamy metody **`remove()`**

```
private ThreadLocal<String> myThreadLocal = new ThreadLocal<String>();
myThreadLocal.set("Hello ThreadLocal");
String threadLocalValue = myThreadLocal.get();
```

w klasie deklarujemy zmienną ThreadLoal o typie String

ThreadLocal

Wartości dla konkretnej ThreadLocal w wątku możemy ustawiać za pomocą metody **`set()`** oraz pobierać za pomocą metody **`get()`**. Aby usunąć wartość ze zmiennej ThreadLocal używamy metody **`remove()`**

```
private ThreadLocal<String> myThreadLocal = new ThreadLocal<String>();
myThreadLocal.set("Hello ThreadLocal");
String threadLocalValue = myThreadLocal.get();
```

w klasie deklarujemy zmienną ThreadLoal o typie String

ustawiamy wartość zmiennej za pomocą metody set()

ThreadLocal

Wartości dla konkretnej ThreadLocal w wątku możemy ustawiać za pomocą metody **`set()`** oraz pobierać za pomocą metody **`get()`**. Aby usunąć wartość ze zmiennej ThreadLocal używamy metody **`remove()`**

```
private ThreadLocal<String> myThreadLocal = new ThreadLocal<String>();
myThreadLocal.set("Hello ThreadLocal");
String threadLocalValue = myThreadLocal.get();
```

w klasie deklarujemy zmienną ThreadLoal o typie String

ustawiamy wartość zmiennej za pomocą metody set()

odczytujemy wartość zmiennej za pomocą metody get()

Volatile

W przypadku synchronizacji prostych zmiennych należy dodać słowo kluczowe **volatile** do współdzielonych zmiennych

- każdy korzystający ze zmiennej wątek będzie widział najnowszą wartość
- wartość będzie pobierana z pamięci a nie z cache
- słowo kluczowe wskazuje że dana zmienna może być modyfikowana między wątkami
- synchronizuje dane pomiędzy wątkami

Przykład:

```
private volatile int counter = 0;
```

Uwaga:

- nie eliminuje to konieczności stosowania synchronizowanych bloków lub metod(o których dowiemy się później)
- podobne w działaniu są też tzw. Atomic Variables

Atomic Variables

Atomic Variables to zmienne które działają na zasadzie porównania i aktualizacji danych. Pozwalają na zmianę wartości zmiennych przez wiele wątków nie blokując tych zmiennych. Operacje na konkretnych zmiennych wykonywane są bez użycia synchronizacji. Korzystając z Atomic Variables można uniknąć wzajemnego zablokowania wątków(o którym powiemy w dalszej części kursu).

Najważniejsze metody klasy Atomic Variables to:

- **`get()`** – pobiera wartość z pamięci tak że zmiany dokonywane przez inne wątki są widoczne, odpowiada to czytaniu wartości zmiennej volatile
- **`set()`** – zapisuje wartość w pamięci tak że zmiana jest widoczna dla innych wątków, odpowiada zapisywaniu wartości w zmiennej volatile
- **`compareAndSet()`** – zwraca true jeśli operacja się powiedzie, zwraca false w przypadku niepowodzenia(przykład w dalszej części)

Atomic Variables

Mechanizm "porównaj i zmień" zmieni wartość A dla zmiennej X na B tylko wtedy, gdy aktualna wartość będzie wynosiła A

Dzięki używaniu tej metody jeżeli kilka wątków chce zaktualizować dany element tylko jednemu się to uda, inne wątki są informowane że ta operacja się nie udała

Minusem tego rozwiązania jest to, że musimy obsłużyć mechanizm ewentualnego niepowodzenia aktualizacji danych.

Atomic Variables

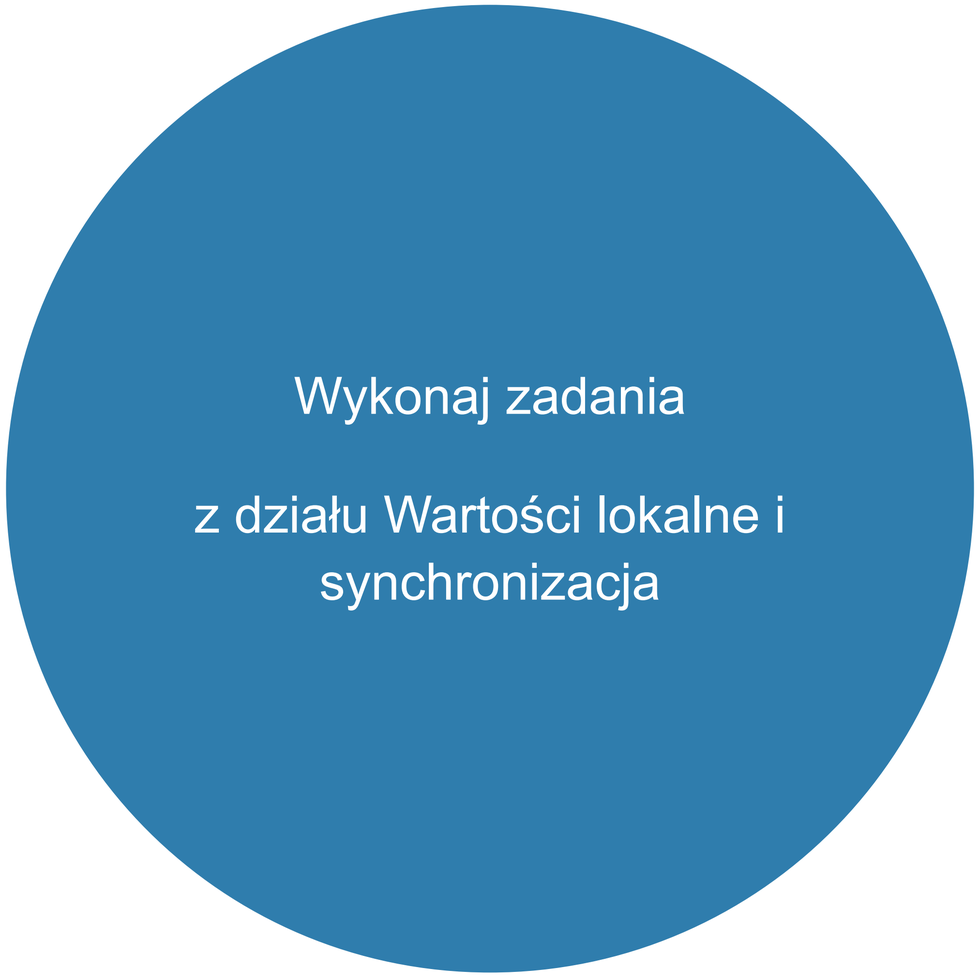
Przykład użycia AtomicInteger w ciele klasy:

```
private final AtomicInteger counter = new AtomicInteger(0);
counter.compareAndSet(existingValue, newValue);
```

dla pewności zmodyfikowania zmiennej możemy użyć konstrukcji:

```
public void increment() {
    while(true) {
        int existingValue = counter.get();
        int newValue = existingValue + 1;
        if(counter.compareAndSet(existingValue, newValue)) {
            return;
        }
    }
}
```

Zadania


Wykonaj zadania
z działu Wartości lokalne i synchronizacja

Synchronizacja

Wyścig warunków

"Wyścigu warunków"(ang. race conditions) - to niepoprawne zachowanie spowodowane przez wątki przeplatające się i wykonujące kod w niezamierzonym porządku - niezależnie od tego że dane zostały poprawnie zsynchronizowane.

Częstymi objawami wyścigu warunków w naszym kodzie może być zachowanie kod z reguły kod działa, ale czasami coś jest nie tak. Zachodzi wtedy duże prawdopodobieństwo, że kod nie jest bezpieczny pod kątem wątków(ang. **thread safe**)

W takich sytuacjach stosujemy też pojęcie sekcji krytycznych(ang. **critical sections**) - czyli jednej lub więcej części kodu które nie mogą być dostępne dla więcej niż jednego wątku w tym samym czasie, w przeciwnym wypadku może powodować to błędy niespójności danych.

Wyścig warunków

Przykład wyścigu warunków w inkrementacji - dwa wątki na raz chcą inkrementować zmienną:

```
public int increment(){
    this.valueToIncrement++;
}
```

Inkrementacja nie jest operacją atomową, następuje: odczyt, zmiana wartości, zapis - w tym samym czasie inny wątek może robić to samo:

- Wątek A odczytuje zmienną i zapamiętuje jej wartość 0
- Wątek A zwiększa wartość do 1 i nadpisuje 0

- Wątek B odczytuje zmienną i zapamiętuje jej wartość 0
- Wątek B zwiększa wartość do 1 i nadpisuje 1 (BŁĄD)

Synchronized

Aby zapobiec sytuacjom gdzie zmienne odczytywane są przez różne wątki i nie wiedzą one nawzajem o wprowadzanych zmianach stosujemy słowo kluczowe **synchronized**

Dzięki słowu Synchronized otrzymujemy następujące działanie:

- jeżeli jakiś wątek chce skorzystać z zasobu, który nie jest używany, ten zasób jest mu udostępniany
- jeżeli inny wątek chce skorzystać z zasobu, który jest już używany, zmieniany jest mu stan z Runnable na Blocked
- kiedy zasób zostanie zwolniony, wtedy czekający wątek zostanie odblokowany i jego stan z Blocked zostanie zmieniony na Runnable
- aby sprawdzić jaki wątek używa aktualnie zasobów można użyć metody **`boolean Thread.holdsLock(object)`**

Synchronizacja - przykład

```
class SynchDemo implements Runnable {
public void run() {
    System.out.println(Thread.holdsLock(this));
    synchronized (this) {
      System.out.println(Thread.holdsLock(this));
    }
  }
}
```

Synchronizacja - przykład

```
class SynchDemo implements Runnable {
public void run() {
    System.out.println(Thread.holdsLock(this));
    synchronized (this) {
      System.out.println(Thread.holdsLock(this));
    }
  }
}
```

sprawdzenie utrzymywania blokady za pomocą metody holdsLock() - zwróci false

Synchronizacja - przykład

```
class SynchDemo implements Runnable {
public void run() {
    System.out.println(Thread.holdsLock(this));
    synchronized (this) {
      System.out.println(Thread.holdsLock(this));
    }
  }
}
```

sprawdzenie utrzymywania blokady za pomocą metody holdsLock() - zwróci false

użycie bloku synchronized()

Synchronizacja - przykład

```
class SynchDemo implements Runnable {
public void run() {
    System.out.println(Thread.holdsLock(this));
    synchronized (this) {
      System.out.println(Thread.holdsLock(this));
    }
  }
}
```

sprawdzenie utrzymywania blokady za pomocą metody holdsLock() - zwróci false

użycie bloku synchronized()

ponowne sprawdzenie utrzymywania blokady - zwróci true

Sposoby użycia synchronized

Obiekt, który otacza słowo kluczowe synchronized może być modyfikowane tylko przez jeden wątek a wszystkie inne wątki chcące skorzystać z synchronizowanego bloku używanego przez inny wątek są zablokowane, dopóki korzystający z niego wątek opuści dany blok

Słowo kluczowe synchronized może być użyte do oznaczenia czterech różnych typów bloków, typ synchronizacji zależy od konkretnej sytuacji:

- metoda 1 - synchronizacja metod
- metoda 2 - synchronizacja statycznych metod
- metoda 3 - synchronizacja bloków w metodach
- metoda 4 - synchronizacja bloków w statycznych metodach

Synchronizacja - rodzaje

Synchronizacja metod jest to synchronizacja na poziomie instancji obiektu

```
public synchronized void add(int value) {
    this.count += value;
}
```

Synchronizacja bloków w metodach jest stosowana gdy nie ma konieczności synchronizowania całych metod lub gdy preferowane jest synchronizowanie wybranych bloków

```
public void add(int value) {
  synchronized(this) {
    this.count += value;
  }
}
```

Synchronizacja - rodzaje

Synchronizacja statycznych metod jest to synchronizacja na poziomie całego obiektu

```
public static synchronized void add(int value) {
      count += value;
}
```

Synchronizacja bloków w statycznych metodach - synchronizacja tak jak w przypadku synchronizacji statycznej metody przebiega na poziomie obiektu

```
public static void log2(String msg1, String msg2) {
    synchronized(MyClass.class) {
      log.writeln(msg1);
      log.writeln(msg2);
    }
}
```

Zadania

Problemy z synchronizacją

Problemy synchronizacji - Deadlock

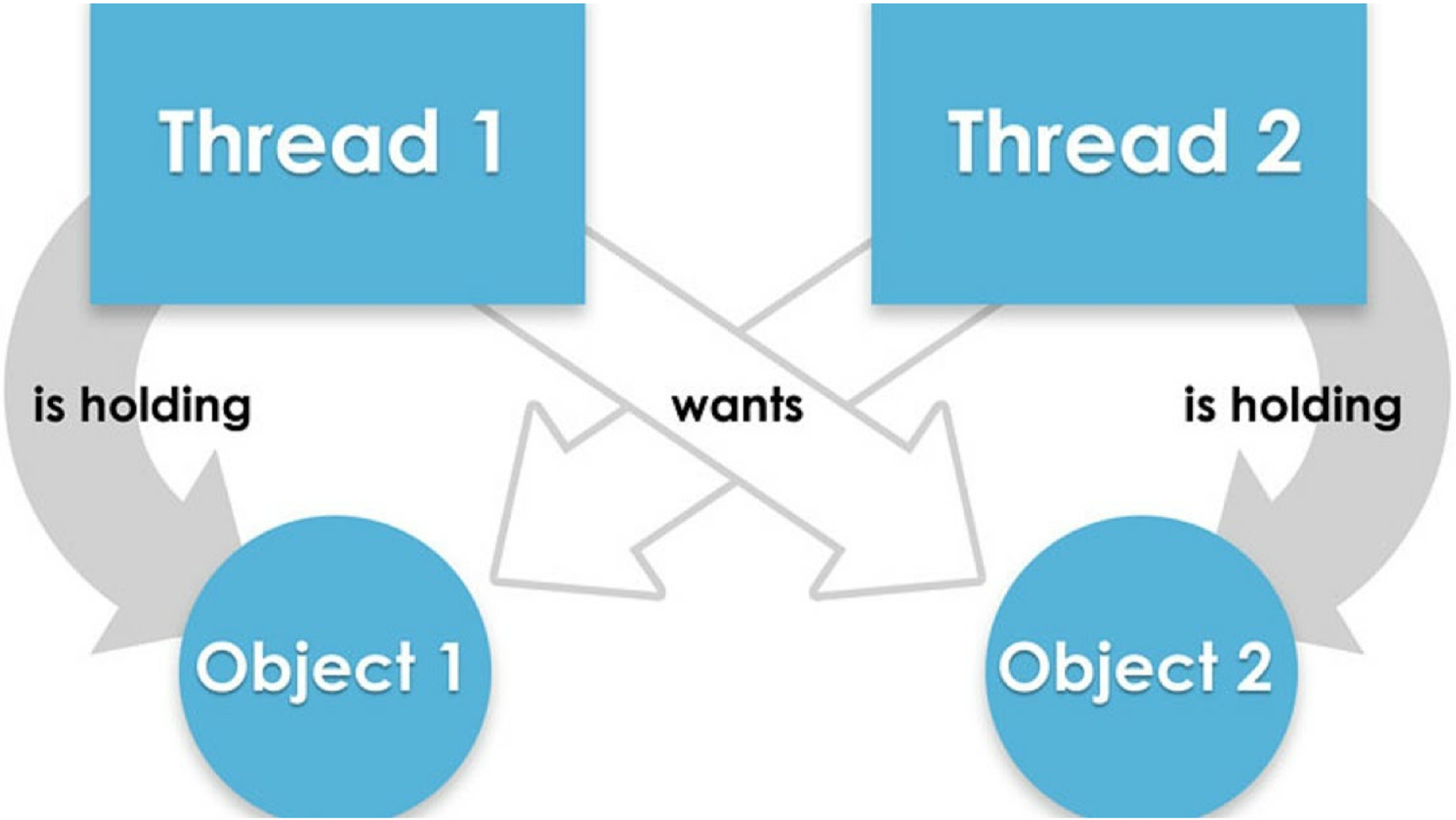
Podczas synchronizacji może pojawić się problem że np. wątki nie robią postępu w wykonywaniu swoich zadań a współdzielone zasoby są blokowane przez inne wątki.

Występuje wtedy sytuacja, że żaden z wątków nie otrzymuje zasobów których wymaga oraz żaden z wątków nie obsługuje swojej sekcji krytycznej, więc utrzymuje blokadę innych wątków jest to przypadek błędu synchronizacji typu **Dead Lock**

W tym typie błędu synchronizacji dwa lub więcej wątków są zaangażowane w proces Dead Lock i nie ma możliwości aby "przełamać" Dead Lock - wątki są całkowicie zablokowane

Dla użytkownika najczęściej aplikacja wygląda jakby się zawiesiła oraz konieczny jest restart aplikacji

Problemy synchronizacji - Deadlock



Deadlock - przykład

Mamy poniższą klasę SyncThread:

```
public class SyncThread implements Runnable {
  Object obj1;
  Object obj2;
  public SyncThread(Object obj1, Object obj2) {
    this.obj1=obj1;
    this.obj2=obj2;
  }
```

Deadlock - przykład

Poniższy przykład metody run() powoduje Deadlock:

```
  public void run() {
    synchronized (obj1) {
      Thread.sleep(2000);
      synchronized (obj2) {
        Thread.sleep(2000);
      }
    }
  }
}
```

Deadlock - przykład

Poniższy przykład metody run() powoduje Deadlock:

```
  public void run() {
    synchronized (obj1) {
      Thread.sleep(2000);
      synchronized (obj2) {
        Thread.sleep(2000);
      }
    }
  }
}
```

pierwszy blok synchronized()

Deadlock - przykład

Poniższy przykład metody run() powoduje Deadlock:

```
  public void run() {
    synchronized (obj1) {
      Thread.sleep(2000);
      synchronized (obj2) {
        Thread.sleep(2000);
      }
    }
  }
}
```

drugi zagnieżdżony blok synchronized

Deadlock - przykład

Wywołanie klasy z wcześniej przedstawioną metodą run():

```
Object obj1 = new Object();
Object obj2 = new Object();
Object obj3 = new Object();
Thread t1 = new Thread(new SyncThread(obj1, obj2), "t1");
Thread t2 = new Thread(new SyncThread(obj2, obj3), "t2");
Thread t3 = new Thread(new SyncThread(obj3, obj1), "t3");
t1.start();
Thread.sleep(5000);
t2.start();
Thread.sleep(5000);
t3.start();
```

Deadlock - przykład

Wywołanie klasy z wcześniej przedstawioną metodą run():

```
Object obj1 = new Object();
Object obj2 = new Object();
Object obj3 = new Object();
Thread t1 = new Thread(new SyncThread(obj1, obj2), "t1");
Thread t2 = new Thread(new SyncThread(obj2, obj3), "t2");
Thread t3 = new Thread(new SyncThread(obj3, obj1), "t3");
t1.start();
Thread.sleep(5000);
t2.start();
Thread.sleep(5000);
t3.start();
```

Utworzenie obiektów

Deadlock - przykład

Wywołanie klasy z wcześniej przedstawioną metodą run():

```
Object obj1 = new Object();
Object obj2 = new Object();
Object obj3 = new Object();
Thread t1 = new Thread(new SyncThread(obj1, obj2), "t1");
Thread t2 = new Thread(new SyncThread(obj2, obj3), "t2");
Thread t3 = new Thread(new SyncThread(obj3, obj1), "t3");
t1.start();
Thread.sleep(5000);
t2.start();
Thread.sleep(5000);
t3.start();
```

utworzenie wątków

Deadlock - przykład

Wywołanie klasy z wcześniej przedstawioną metodą run():

```
Object obj1 = new Object();
Object obj2 = new Object();
Object obj3 = new Object();
Thread t1 = new Thread(new SyncThread(obj1, obj2), "t1");
Thread t2 = new Thread(new SyncThread(obj2, obj3), "t2");
Thread t3 = new Thread(new SyncThread(obj3, obj1), "t3");
t1.start();
Thread.sleep(5000);
t2.start();
Thread.sleep(5000);
t3.start();
```

rozpoczęcie działania

Deadlock - przeciwdziałanie

W poprzednim przykładzie kodu poszczególne obiekty zostały zablokowane i spowodowało to wystąpienie błędu typu Dead Lock. Można temu zapobiec w następujący sposób:

- najprostszym sposobem przeciwdziałania Dead Lock jest unikanie **zagnieżdżonych** synchronizacji
- w poprzedniej metodzie run(), aby klasa działa poprawnie i wyeliminowała Dead Lock, wystarczyło nie zagnieżdżać bloków synchronized
- należy też pamiętać aby synchronizować tylko wymagane obiekty i nie robić tego "na zapas"
- należy również unikać oczekiwania na zakończenie działania wątku w nieskończoność przy użyciu metody join(), warto wtedy użyć ograniczenia czasowego

LiveLock

Kolejnym popularnym błędem synchronizacji danych jest **LiveLock** który charakteryzuje się tym że:

- wątki nie są całkowicie zablokowane jak w Dead Lock
- wątki czekając mogą wykonać inną pracę
- zasoby na które oczekują wątki są zablokowane
- można powiedzieć, że wątki działają w nieskończonej pętli i nie robią postępu
- w tym przypadku zarówno jeden jak i drugi wątek będą czekały na zmianę statusu drugiego, co biorąc pod uwagę poniższy kod nigdy nie nastąpi

LiveLock - przykład

```
while (!threadTwo.isColorSet()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isPaintMade = true;
```

```
while (!threadOne.isPaintMade()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isColorSet = true;
```

LiveLock - przykład

```
while (!threadTwo.isColorSet()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isPaintMade = true;
```

```
while (!threadOne.isPaintMade()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isColorSet = true;
```

Działanie wątku threadOne

LiveLock - przykład

```
while (!threadTwo.isColorSet()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isPaintMade = true;
```

```
while (!threadOne.isPaintMade()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isColorSet = true;
```

Działanie wątku threadTwo

LiveLock - przykład

```
while (!threadTwo.isColorSet()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isPaintMade = true;
```

```
while (!threadOne.isPaintMade()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isColorSet = true;
```

Warunek działający w wątku threadOne oczekujący na zmianę flagi w wątku threadTwo

LiveLock - przykład

```
while (!threadTwo.isColorSet()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isPaintMade = true;
```

```
while (!threadOne.isPaintMade()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isColorSet = true;
```

kod działający w pętli dopóki powyższy warunek nie spełni się

LiveLock - przykład

```
while (!threadTwo.isColorSet()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isPaintMade = true;
```

```
while (!threadOne.isPaintMade()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isColorSet = true;
```

dopiero w tym miejscu wątek threadOne zmieniłby swoją flagę

LiveLock - przykład

```
while (!threadTwo.isColorSet()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isPaintMade = true;
```

```
while (!threadOne.isPaintMade()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isColorSet = true;
```

Warunek działający w wątku threatTwo oczekujący na zmianę flagi w wątku threadOne

LiveLock - przykład

```
while (!threadTwo.isColorSet()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isPaintMade = true;
```

```
while (!threadOne.isPaintMade()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isColorSet = true;
```

kod działający w pętli dopóki powyższy warunek nie spełni się

LiveLock - przykład

```
while (!threadTwo.isColorSet()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isPaintMade = true;
```

```
while (!threadOne.isPaintMade()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isColorSet = true;
```

dopiero w tym miejscu wątek threadTwo zmieniłby swoją flagę

LiveLock - przykład

```
while (!threadTwo.isColorSet()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isPaintMade = true;
```

```
while (!threadOne.isPaintMade()) {
  try {
    Thread.sleep(1000);
  } catch (InterruptedException ex) {
    ex.printStackTrace();
  }
}
this.isColorSet = true;
```

jak widać, warunki nigdy nie zostaną spełnione

Starvation

Ostatnim i omawianym błędem synchronizacji danych jest **Starvation** który charakteryzuje się tym że:

- zachodzi w sytuacji gdy jeden wątek utrzymuje zasoby tak długo, że inne wątki są zablokowane
- zablokowane wątki oczekują na zasoby których nie dostają
- wątki mogą czekać długo np. przez wykonywanie przez dany wątek długiej operacji I/O
- wątek nie dostaje zasobów od CPU ze względu na niski priorytet w porównaniu z innymi wątkami

Problemy z synchronizacją - podsumowanie

Poznaliśmy trzy problemy które mogą wystąpić przy wielowątkowych programach: Deadlock, LiveLock i Starvation

Livelock i starvation nie są tak powszechne jak deadlock, ale też zdarza sie że występują więc warto mieć je na uwadze przy tworzeniu własnych aplikacji

W ramach podsumowania:

- deadlock: wszystkie wątki są zablokowane, program "wisi" w nieskończoność
- livelock: wątki nie są zablokowane ale działają w nieskończonej pętli, program działa ale nie robi postępu
- starvation: tylko jeden wątek działa, inne wątki czekają w nieskończoność

Komunikacja między wątkami

Komunikacja między wątkami

Klasa Object posiada trzy finalne metody które umożliwiają wątkom komunikację na temat statusu blokady zasobów, są to:

- **`wait()`**
- **`notify()`**
- **`notifyAll()`**

Metoda wait()

Bardzo przydatna metoda która czasami mylona jest z metodą sleep() ale działa w zupełnie inny sposób. Charakteryzuje się tym że:

- działa na obiekcie i jest ściśle związana z mechanizmem synchronizacji
- informuje wątek że może zwolnić zasoby i zostać uśpiony do czasu aż inny wątek wywoła metodę notify() która ponownie "obudzi" dany wątek
- nie jesteśmy w stanie samodzielnie zaimplementować metody wait() i musimy polegać na domyślnej implementacji
- przykład użycia:

```
synchronized(object) {
  while(!condition)
  {
        object.wait();
  }
}
```

Metoda notify()

Przy stosowaniu wcześniej poznanej metody wait() która "usypia" obiekt stosuje się odwrotny mechanizm czyli metodę **`notify()`** która charakteryzuje się tym że:

- "budzi" pojedynczy wątek który wywołał metodę wait() na tym samym obiekcie
- notify() nie zwalnia zasobów, a informuje docelowy wątek że może się "wybudzić"
- zasoby są zwalniane dopiero gdy blok synchronizacji z użytą metodą notify() dochodzi do końca
- przykład użycia:

```
synchronized(object) {
    object.notify();
}
```

Metoda notifyAll()

Dopełnieniem wcześniej poznanych metod wait() oraz notify() jest metoda **`notifyAll()`** która charakteryzuje się tym że:

- ma za zadanie "wybudzić" wszystkie wątki które wcześniej wywołały metodę wait() na konkretnym obiekcie
- notifyAll() nie zwalnia zasobów, a informuje docelowy wątek że może się "wybudzić"
- przykład użycia:

```
synchronized(object) {
    object.notifyAll();
}
```

Zadania

KONIEC