# Współbieżność i równoległość w środowiskach obiektowych

# Java

- Model współbieżności Javy opiera się na realizacji szeregu omawianych dotychczas elementów:
  - zarządzanie wątkami
  - operacje atomowe (wiele standardowych operacji na referencjach, zmiennych typów wbudowanych, operacje na zmiennych deklarowanych jako volatile, a także specjalne operacje takie jak np. compareAndSet, getAndIncrement itp.)
  - mechanizmy synchronizacji (zamki, zmienne warunku i inne charakterystyczne dla funkcjonowania obiektów)
  - klasy kontenerowe przystosowane do współbieżnego używania, takie jak: BlockingQueue, ConcurrentMap itp.

# Klasy, obiekty i współbieżność

- Atrybuty obiektów w programach współbieżnych niosą te same ryzyka błędów przy wykonaniu współbieżnym co zmienne globalne w programach proceduralnych
- Metody obiektów mogą zostać wywołane przez procedury z dowolnych wątków
- Wymaganiem dla każdej projektowanej klasy, której obiekty mogą być wykorzystywane przez obiekty innych klas jest „bezpieczeństwo wątkowe” (*thread safety*)
  - Od wszystkich ogólnie dostępnych bibliotek wymaga się bezpieczeństwa wątkowego
- Pośrednim rozwiązaniem, nie zawsze możliwym jest wykorzystanie klas niezmiennych (*immutable*)

# Wątki w Javie

- Narzędzia udostępniane przez Javę programistom tworzącym programy wielowątkowe należą do typowych konstrukcji obiektowych: klasy, interfejsy, metody
  - wątek w Javie jest także obiektem (klasy Thread z pakietu java.lang)
- Wielowątkowość jest wbudowana w same podstawy systemu
  - podstawowa klasa Object zawiera metody związane z przetwarzaniem współbieżnym (wait, notify, notifyAll)
  - każdy obiekt może stać się elementem przetwarzania współbieżnego

# Wątki w Javie

- W jaki sposób sprawić żeby przetwarzanie stało się współbieżne
  - zaprojektować obiekt z metodą przeznaczoną do rozpoczęcia przetwarzania współbieżnego
  - uruchomić nowy wątek realizujący metodę
- Istnieją dwa sposoby uruchamiania wątków w Javie
  - poprzez przekazanie obiektu, którego metodę chcemy wykonać, jako argumentu dla metody `start` obiektu klasy `Thread`
    - nasz obiekt musi implementować interfejs `Runnable`
    - uruchamiana metoda musi implementować metodę `run`
  - poprzez uruchomienie metody `start` naszego obiektu
    - nasz obiekt musi być obiektem klasy pochodnej klasy `Thread`
    - uruchamiana metoda musi przeciążać metodę `run`

# Wątki w Javie - przykłady

```
public class HelloRunnable implements Runnable
    {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(
            new HelloRunnable())).start();
    }
}
```

- Klasa może dziedziczyć po innych klasach

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }
}
```

- Klasa nie może dziedziczyć po innych klasach

# Wątki w Javie

- Podstawowe metody klasy Thread (pakiet java.lang)
  - void start() – polecenie dla JVM uruchomienia metody run
  - void run() – uruchomienie metody run przekazanego obiektu implementującego interfejs Runnable (w przeciwnym przypadku natychmiastowy powrót)
    - metoda jest przeciążana w klasach pochodnych
  - static void sleep(long millis) – uśpienie bieżącego wątku na określony czas
  - void join(long millis), void join() - oczekiwanie na zakończenie wątku
  - void interrupt() – przerwanie działania wątku (dokładniej ustawienie sygnalizatora przerwania) – przerwanie następuje jeśli wątek znajduje się wewnątrz metody dającej się przerwać, np. sleep, join, wait lub jeśli w trakcie realizacji wątku wywołana jest metoda:
  - static boolean interrupted() - sprawdzenie stanu sygnalizatora przerwania (dla bieżącego wątku) - później powinna znajdować się reakcja wątku na próbę przerwania, np. zwrócenie wyjątku InterruptedException

# Wątki w Javie - przykład

```
private static class obiektRunnable implements Runnable {
   public void run() {
      try {
         Thread.sleep(4000);
         for (int i = 0; i < number; i++) {
            obliczenia();
            if (Thread.interrupted()) {
               throw new InterruptedException();
            }
         }
      } catch (InterruptedException e) {
          // reakcja na przerwanie
   }
}
```

# Wątki w Javie - przykład

```
public class TestThreads {
   public static void main(String args[]) throws InterruptedException {
      Thread t = new Thread(new obiektRunnable());
      t.start();
      t.join(1000); // oczekiwanie 1 sekundę
      if (t.isAlive()) {
         t.interrupted();
         t.join();
      }
   }
}
```

- boolean isAlive(), boolean isInterrupted() - procedury sprawdzenia czy dany wątek jest wykonywany i czy został przerwany

# Wątki w Javie - synchronizacja

- Działanie dowolnego obiektu w Javie można uczynić bezpiecznym ze względu na wykonanie wielowątkowe poprzez określanie jego metod jako synchronizowanych
- Jeżeli metoda jest określona jako synchronizowana, wtedy
  - jej realizacja przez jakiś wątek uniemożliwia realizację dowolnej synchronizowanej metody tego obiektu przez inne wątki (wątki zgłaszające chęć wywołania dowolnej synchronizowanej metody tego obiektu są w takiej sytuacji ustawiane w kolejce wątków oczekujących)
  - ten sam wątek może realizować kolejne wywołania synchronizowanych metod
  - w momencie zakończenia realizacji wszystkich synchronizowanych metod obiektu przez dany wątek JVM wznawia działanie pierwszego wątku oczekującego w kolejce
- Uwaga: konstruktory nie mogą być synchronizowane

# Wątki w Javie - synchronizacja

- Mechanizm synchronizacji w Javie opiera się na istnieniu wewnętrznych zamków monitorowych (*monitor locks*) zwanych często w skrócie monitorami
- Z każdym obiektem w Javie związany jest zamek monitorowy
  - wątek rozpoczynający wykonywanie synchronizowanej metody obiektu zajmuje zamek tego obiektu – żaden inny wątek nie otrzyma go dopóki dany wątek nie zwolni zamka
  - zamek związany z danym obiektem można zająć także rozpoczynając wykonanie bloku synchronizowanego
  - rozpoczynając wykonanie synchronizowanej statycznej funkcji klasy zajmuje się zamek związany z obiektem reprezentującym daną klasę

# Wątki w Javie – synchronizacja – przykład

```
public class Liczniki {

   private long c1 = 0; private long c2 = 0;

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

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

# Wątki w Javie - synchronizacja

- Synchronizowany blok oznacza się określeniem synchronized z argumentem będącym referencją do obiektu, którego zamek ma zostać zajęty
  - najczęściej przesyłana jest referencja do obiektu, w klasie którego blok jest zdefiniowany (referencja this)
    - dzięki temu tylko część kodu obiektu jest synchronizowana
- Można tworzyć odrębne obiekty, których jedyną funkcją będzie występowanie jako argument przy synchronizacji bloków
  - takie obiekty pełnią rolę zbliżoną do muteksów
  - jedna z różnic polega na braku realizacji metody trylock – dlatego Java w pakiecie java.util.concurrency udostępnia interfejs Lock (posiadający standardowe funkcje muteksa) i jego kilka implementacji (np. typowy ReentrantLock)

# Wątki w Javie – synchronizacja – przykład

```
public class LicznikiMuteksy {
   private long c1 = 0; private long c2 = 0;
   private Object lock1 = new Object();
   private Object lock2 = new Object();

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

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

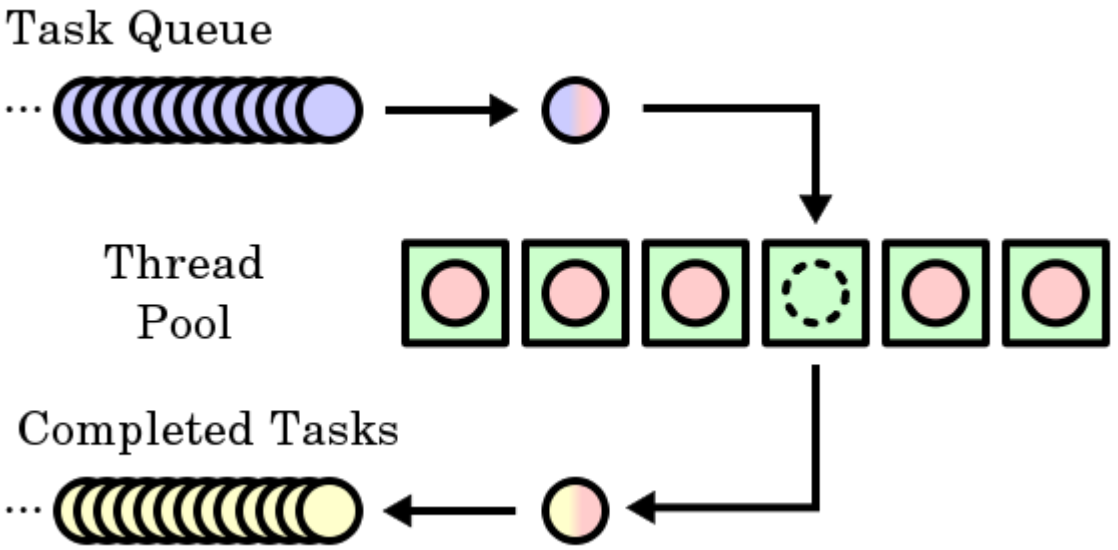
# Wątki w Javie – zmienne i operacje atomowe

```
import java.util.concurrent.atomic.AtomicInteger;

public class LicznikiAtomowe{
   private AtomicInteger c1 = new AtomicInteger(0);
   private AtomicInteger c2 = new AtomicInteger(0);

   public void inc1() {
        try {
            c1.incrementAndGet();
        } catch (InterruptedException e) {    // reakcja na przerwanie    }
   }

   public void inc2() {
        try {
            c2.incrementAndGet();
        } catch (InterruptedException e) {  // reakcja na przerwanie    }
   }
}
```

# Java – obliczenia masowo równoległe

- Dotychczas omówione mechanizmy ukierunkowane były głównie na przetwarzanie wielowątkowe z mała liczbą zadań
- Dla masowej równoległości Java posiada specjalne udogodnienia: pule wątków (*thread pool*) i interfejsy wykonawców (*executor*)
- Konkretna pula wątków (obiekt klasy ThreadPoolExecutor ) realizuje zarządzanie dostarczonymi zadaniami (obiektami implementujacymi interfejs Runnable lub Callable), które może obejmować:
  - uruchamianie, zatrzymywanie, sprawdzanie stanu itp.
- Uruchamianie zadań w ramach wykonawców posiada mniejszy narzut niż uruchamianie nowych wątków (klasy Thread)

# Pula wątków



# Prosty przykład puli wątków w Javie

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Simple_executor_test {

   private static final int NTHREADS = 10;

   public static void main(String[] args) {

      Licznik licznik = new Licznik();
      ExecutorService executor = Executors.newFixedThreadPool(NTHREADS);
      // ExecutorService jest rozszerzeniem interfejsu Executor
                                          // klasa Executors dostarcza metody wytwórcze dla pul wątków

      for (int i = 0; i < 50; i++) {
         Runnable zwiekszaczLicznikow = new ZwiekszaczLicznikow(licznik);
         executor.execute(zwiekszaczLicznikow); // funkcja execute tylko wykonuje zadanie
      }

      executor.shutdown(); // egzekutor przestaje przyjmować nowe wątki

      while (!executor.isTerminated()) {} // oczekiwanie na zakończenie pracy wątków
   }
}
```

# Pule wątków

- Najprostsze zadania (interfejs Runnable) nie zwracają wyniku
- Interfejs Callable umożliwia definiowanie zadań zwracających wynik
- Do przekazywania wyników z obiektów Callable i sprawdzania zakończenia działania obiektów Runnable służą obiekty Future
  - interfejs Future jest parametryzowany typem zwracanego wyniku
  - interfejs Future definiuje funkcje do:
    - sprawdzania czy zadanie zostało zakończone
    - oczekiwania na zakończenie zadania i pobierania wyniku realizacji zadania zapisanego w obiekcie Future
    - anulowania zadań i sprawdzania czy zostały anulowane

# Pule wątków

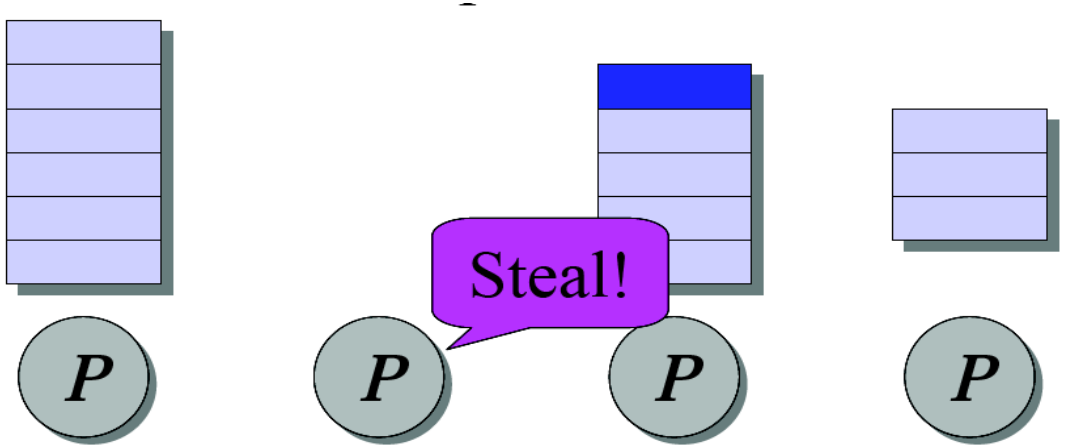
```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

   public class HelloFuture implements Callable<String> {
                        // klasa HelloFuture implementuje interfejs Callable parametryzowany
                        //  typem wyniku zwracanego przez funkcję call
   private int task_no;

   public HelloFuture(Integer Task_no) {
      this.task_no = Task_no;
   }

   public String call() throws Exception {
      return("Hello from task "+task_no+"(thread: " +Thread.currentThread().getName()+")");
   }
```

# Pule wątków

```
public static void main(String args[]){
    ExecutorService executor = Executors.newFixedThreadPool(10);
    List<Future<String>> list = new ArrayList<Future<String>>(); // interfejs Future jest
                    // parametryzowany typem obiektu zwracanego przez funkcję call powiązanego zadania
    for(int i=0; i< 100; i++){
        Callable<String> callable = new HelloFuture( i );
        Future<String> future = executor.submit(callable); // funkcja submit zwraca obiekt typu future
                                                            // powiązany z przekazanym zadaniem
        list.add(future);
    }
    for(Future<String> future_string : list){
        try {
            System.out.println(future_string.get()); // funkcja get przekazuje wyniki zwracany przez call
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
     executor.shutdown();
}
```

# Work stealing

- Technika „kradzieży pracy”(?) (*work stealing*) jest sposobem na efektywne dzielenie dużej liczby zadań pomiędzy wątki
  - decyzja o przydziale zadania jest podejmowana tylko jeśli jakiś wątek nie ma w danym momencie nic do zrobienia
- Strategia *work stealing* jest realizowana np. w klasie ForkJoinPool bedącej realizacją wzorca *fork-join* (*recursive splitting*)



# Prosty przykład interfejsu ForkJoin

```
import java.util.concurrent.ForkJoinPool;
// pula wątków klasy ForkJoinPool realizuje zadania implementujące interfejs Future
import java.util.concurrent.RecursiveAction; // zadanie nie zwracające wyniku
import java.util.concurrent.RecursiveTask; // klasa zadań umożliwiająca zwracanie wyniku
                                           // jest parametryzowana typem wyniku
class MaxTask extends RecursiveTask<Double> {

    private double[] arrayForMax;
    private int index_start;
    private int index_end;

    final private int THRESHOLD = 100;

    public MaxTask(double[] arrayForMax, int index_start, int index_end) {
        this.index_start = index_start;
        this.index_end = index_end;
        this.arrayForMax = arrayForMax;
    }

    public double maxSeq(){
        double maxLoc=arrayForMax[index_start];
        for (int i = index_start; i <= index_end; i++) {
            if(arrayForMax[i]>maxLoc) maxLoc = arrayForMax[i];
        }
        return maxLoc;
    }
```

# Prosty przykład interfejsu ForkJoin

```
protected Double compute() {  // podstawowa funkcja podklas ForkJoinTask

    System.out.println("Task: ( " + index_start + " , " + index_end + ")");

    if ((index_end - index_start) <= THRESHOLD) {
        return maxSeq();
    }

    int index_mid = (index_start + index_end) / 2;

    MaxTask left = new MaxTask(arrayForMax, index_start, index_mid-1);
    MaxTask right = new MaxTask(arrayForMax, index_mid, index_end);

    left.fork(); // dla obiektu (pod)klasy ForkJoinTask zlecenie wykonania funkcji compute
    right.fork();

    double leftMax = left.join();       // dla obiektu (pod)klasy ForkJoinTask oczekiwanie na zakończenie
    double rightMax = right.join();     // wykonania funkcji compute i zwrócenie wyniku
                                        // (dla zadań ForkJoinTask można także wywoływać funkcję get

    return Math.max(leftMax, rightMax);
  }

}
```

# Prosty przykład z interfejsem ForkJoin

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;
import java.util.concurrent.RecursiveTask;

public class ForkJoinTest {

  public static void main(String[] args) {

    Scanner scanner = new Scanner(System.in);
    System.out.println("Set array size: ");    int totalNumbers = scanner.nextInt();
    double[] numbers = new double[totalNumbers]; // ...nadanie wartości w tablicy

    int i_p = 0;    int i_k = totalNumbers-1;
    MaxTask task = new MaxTask(numbers, i_p, i_k);
    ForkJoinPool forkJoinPool = new ForkJoinPool();

    //double wynik = forkJoinPool.invoke(task); // invoke umożliwia oczekiwanie na wynik
    forkJoinPool.execute(task); // execute wykonuje zadanie bez pobrania wyniku
    //forkJoinPool.submit(task); // submit wykonuje zadanie i zwraca obiekt Future

    System.out.println("Max el in array: " + task.join());

  }
}
```