Wykład 6

Rafał Skinderowicz

Plan wykładu

- Wykonawcy (Executors)
- Kolekcje synchronizowane i współbieżne
- Programowanie współbieżne a GUI

Wykonawcy w Javie

Interfejs `java.util.concurrent.Executor` pozwala na separację zadań do wykonania od mechanizmów tworzenia i uruchamiania wątków

```
1 public interface Executor {
2     void execute(Runnable);
3 }
```

Zamiast

```
1 (new Thread(zadanie)).start();
```

mamy

```
1 e.execute(zadanie); // e - obiekt klasy Executor
```

Wykonawcy w Javie

- Nie określamy kiedy i w jaki sposób zadanie ma zostać wykonane – zależy to od konkretnej implementacji interfejsu `Executor`, w szczególności zadania mogą być uruchamiane
  - kolejno w bieżącym wątku
  - od razu, każde w osobnym wątku – wykonanie asynchroniczne
  - kolejno po jednym lub kilka jednocześnie w osobnych wątkach tworzących pulę wątków roboczych (ang. worker threads)

```
1 // Najprostsza implementacja
2 class DirectExecutor implements Executor {
3     public void execute(Runnable r) {
4         r.run();
5     }
6 }
```

Pule wątków

- W wielu aplikacjach typu serwerowego (WWW, baz danych, plików, pocztowe) istnieje konieczność wykonywania dużej liczby krótkich zadań związanych z obsługą zdalnych żądań od klientów.
- Naiwna implementacja – tworzenie osobnego wątku dla każdego żądania, problematyczne przy dużej liczbie zadań:

Pule wątków

- W wielu aplikacjach typu serwerowego (WWW, baz danych, plików, pocztowe) istnieje konieczność wykonywania dużej liczby krótkich zadań związanych z obsługą zdalnych żądań od klientów.
- Naiwna implementacja – tworzenie osobnego wątku dla każdego żądania, problematyczne przy dużej liczbie zadań:
  - narzut czasowy związany z uruchamianiem i kończeniem wątków

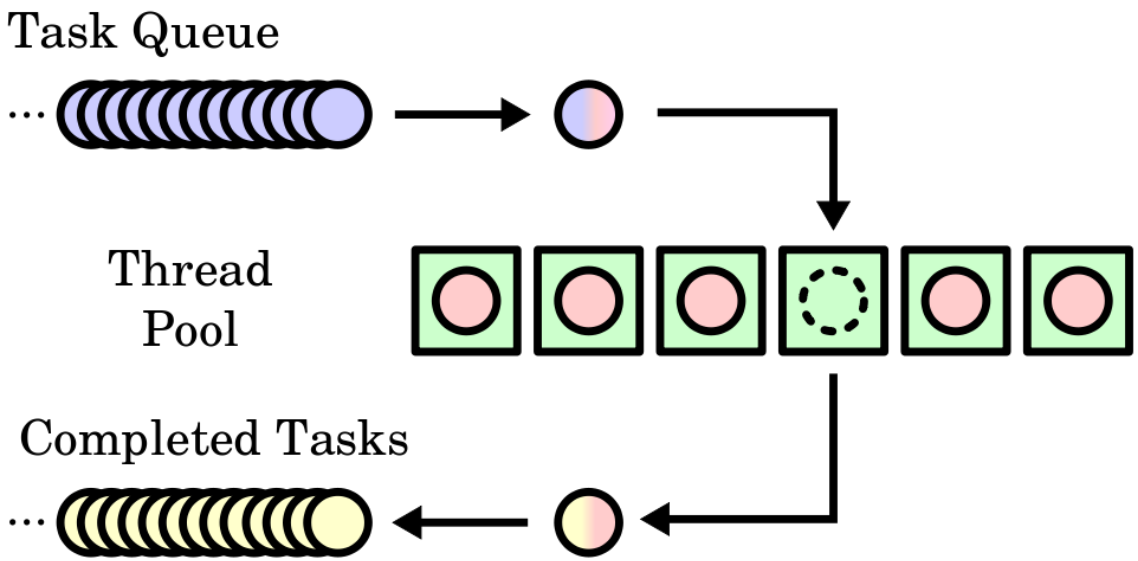
Pule wątków

- W wielu aplikacjach typu serwerowego (WWW, baz danych, plików, pocztowe) istnieje konieczność wykonywania dużej liczby krótkich zadań związanych z obsługą zdalnych żądań od klientów.
- Naiwna implementacja – tworzenie osobnego wątku dla każdego żądania, problematyczne przy dużej liczbie zadań:
  - narzut czasowy związany z uruchamianiem i kończeniem wątków
  - dodatkowe zapotrzebowanie na pamięć:
    - pamięć dla obiektu klasy Thread
    - dwa dodatkowe stosy wywołań, jeden dla javy, drugi dla metod natywnych (zazwyczaj w języku C)
    - zasoby związane z natywnymi wątkami OS

Pule wątków

- W wielu aplikacjach typu serwerowego (WWW, baz danych, plików, pocztowe) istnieje konieczność wykonywania dużej liczby krótkich zadań związanych z obsługą zdalnych żądań od klientów.
- Naiwna implementacja – tworzenie osobnego wątku dla każdego żądania, problematyczne przy dużej liczbie zadań:
  - narzut czasowy związany z uruchamianiem i kończeniem wątków
  - dodatkowe zapotrzebowanie na pamięć:
    - pamięć dla obiektu klasy Thread
    - dwa dodatkowe stosy wywołań, jeden dla javy, drugi dla metod natywnych (zazwyczaj w języku C)
    - zasoby związane z natywnymi wątkami OS
- Rozwiązanie – wykonywanie zadań za pomocą puli wątków roboczych o ustalonym rozmiarze

Pule wątków



Rysunek 1: Ilustracja działania puli wątków (źródło: Cburnett CC SA 3.0)

Pule wątków – wady

- Dodatkowe ryzyko *zakleszczenia*, jeżeli wykonywane zadania oczekują na wyniki innych zadań, które nie mogą się wykonać ze względu na ograniczony rozmiar puli
- Nieefektywne wykorzystanie zasobów w przypadku puli o zbyt dużych rozmiarach w stosunku do potrzeb
- Możliwe przeładowanie puli w przypadku zbyt wielu żądań
- „Wyciek" wątków roboczych z puli przy implementacji nieobsługującej awaryjnego kończenia zadań

Pule wątków – wskazówki

- Ostrożnie z zadaniami, które czekają na wyniki innych zadań
- Ostrożnie z zadaniami o długim czasie wykonania – np. zadania oczekujące na zakończenie operacji I/O zmniejszają niepotrzebnie wykorzystanie CPU – anulowanie zadania i ponowne wstawienie do kolejki
- Jak właściwie dobrać rozmiar puli?
  - Najlepsze wykorzystanie CPU, gdy rozmiar puli $N = l\_proc$
  - Dla zadań wykonujących operacje I/O $N \approx l\_proc(1 + w/s)$, gdzie w to czas oczekiwania na zakończenie operacji I/O, s to całkowity czas wykonania zadania
  - Profilowanie

Wykonawcy w Javie

Za pomocą klasy `Executors` można tworzyć nowe obiekty wykonawców

- `newSingleThreadExecutor()` – obliczenia wykonywane za pomocą dodatkowego wątku roboczego – pozostałe zadania wstawiane do kolejki
- `newFixedThreadPool(int n)` – obliczenia wykonywane za pomocą puli *n* wątków
- `newCachedThreadPool()` – pula wątków o zmiennej liczebności – w miarę zapotrzebowania rozszerzana o nowe wątki robocze, a w przypadku bezczynności zmniejszana
- `newScheduledThreadPool(int n)` – do wykonywania zadań po upłynięciu określonego czasu lub okresowo

Wykonawcy w Javie

- Wszystkie wymienione implementacje wykonawców implementują również interfejs `ExecutorService`
- Interfejs `ExecutorService` rozszerza Executor o dodatkowe metody umożliwiające m.in. kończenie pracy wykonawców:
  - `shutdown()` – dezaktywuje wykonawcę – czeka na zakończenie zleconych zadań, ale nie przyjmuje nowych
  - `shutdownNow()` – zatrzymuje wszystkie wykonywane zadania i anuluje wykonanie zadań oczekujących
  - `awaitTermination(long timeout, TimeUnit unit)` – dezaktywuje wykonawcę i oczekuje na zakończenie wykonania zleconych zadań, ale nie dłużej, niż podany czas
  - `isTerminated()` – zwraca true, jeżeli wykonawca został dezaktywowany

ExecutorService – przykład

```
1  class NetworkService {
2     private final ServerSocket serverSocket;
3     private final ExecutorService pool; // pula wątków roboczych
4     public NetworkService(int port, int poolSize) throws IOException {
5       serverSocket = new ServerSocket(port);
6       pool = Executors.newFixedThreadPool(poolSize);
7     }
8     public void serve() {
9       try {
10        for (;;) {
11          pool.execute(new Handler(serverSocket.accept()));
12        }
13      } catch (IOException ex) {
14        pool.shutdown();
15      }
16    }
17  }
```

ExecutorService – przykład c.d.

Szkielet implementacji klasy obsługującej pojedyncze połączenie

```
1 class Handler implements Runnable {
2    private final Socket socket;
3    Handler(Socket socket) { this.socket = socket; }
4    public void run() {
5      // tutaj obsługa żądania
6    }
7 }
```

Interfejs Callable

- Interfejs `Runnable` nie umożliwia bezpośredniego zwracania wyników obliczeń w metodzie `run()`
- Interfejs `Callable<V>` umożliwia zwracanie wyniku bezpośrednio, a dodatkowo metoda ta może rzucić wyjątek

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

Interfejs Callable

- Interfejs `ExecutorService` pozwala na wykonywanie zadań implementujących `Callable` za pomocą metody
  `<V> Future<V> submit(Callable<V> task)`
- Wynik `Callable` zwracany jest przez obiekt implementujący interfejs `Future`

Interfejs Future

- Umożliwia asynchroniczne wykonywanie zadań (obliczeń)
- Czeka na zakończenie wykonania zadania i zapamiętuje jego wynik
- Typowy scenariusz zastosowania:
  - klient tworzy zadanie do wykonania i przekazuje je wykonawcy
  - klient otrzymuje referencję do obiektu implementującego `Future`
  - klient wykonuje dodatkowe obliczenia, które są niezależne od wyniku zadania
  - klient pobiera rezultat zadania z `Future`, jeżeli trzeba, to czeka na jego zakończenie
  - klient wykorzystuje wynik zadania

Interfejs Future

Metody:

- `V get()`
  - zwraca wynik zadania **czekając** na jego zakończenie
  - dostępna jest wersja z parametrem określającym maksymalny czas oczekiwania, która rzuca wyjątek `TimeoutException`
  - w przypadku **anulowania** rzuca wyjątek `CancelledException`
  - w przypadku błędu wykonania zadania rzuca wyjątek `ExecutionException`
- `boolean isDone()` – zwraca `true`, jeżeli wykonanie zadania zostało zakończone lub anulowane
- `boolean isCancelled()` – zwraca `true`, jeżeli wykonanie zadania zostało anulowane

Future – przykład

```
1  interface Image {
2      byte[] getImageData();
3  }
4  interface Renderer {
5      Image render(byte[] imgData);
6  }
7  class App { // przykładowe zastosowanie Future
8      void app(final byte[] imgData) throws ... {
9          final Renderer r = ...;
10         FutureTask<Image> p = new FutureTask<Image>(
11             new Callable<Image>() { // Wynikiem wykonania będzie
12                 Image call() {       // ob. typu Callable
13                     return r.render(imgData);
14                 }
15             });
16         new Thread(p).start();
17         doSomethingElse();
18         display(p.get()); // czekaj jeżeli nie gotowy
19     // ...
20 }
```

Future – przykład 2 (J. Ponge)

```
1  import java.util.*;
2  import java.util.concurrent.*;
3  import static java.util.Arrays.asList;
4  public class Sums {
5      static class Sum implements Callable<Long> {
6          private final long from;
7          private final long to;
8          Sum(long from, long to) {
9              this.from = from;
10             this.to = to;
11         }
12         @Override
13         public Long call() {
14             long acc = 0;
15             for (long i = from; i <= to; i++) {
16                 acc = acc + i;
17             }
18             return acc;
19         }
20     }
```

Future – przykład 2 c.d.

```
1  public static void main(String[] args) throws Exception {
2      ExecutorService executor = Executors.newFixedThreadPool(2);
3      List <Future<Long>> results = executor.invokeAll(asList(
4          new Sum(0, 10),
5          new Sum(100, 1000),
6          new Sum(10000, 1000000)
7      ));
8      executor.shutdown();
9      for (Future<Long> result : results) {
10         System.out.println(result.get());
11     }
12 }
```

Fork / Join

W Javie 7 dodano wykonawcę `ForkJoinPool` implementującego `ExecutorService` dla ułatwienia efektywnej implementacji algorytmów *równoległych*.

- automatyczna dystrybucja zadań między wątkami w puli
- jeżeli jakieś zadanie czeka na wyniki innego, to uruchamiane jest kolejne z zaplanowanych zadań (ang. work stealing)
- zadania powinny dziedziczyć z klasy `RecursiveTask<V>`, gdy zwracają wynik lub `RecursiveAction`, gdy nie zwracają wyniku
- `ForkJoinPool` domyślnie korzysta z tylu wątków ile jest dostępnych fizycznych rdzeni / procesorów

Fork / Join

Schemat algorytmu nadającego się do zastosowania modelu Fork / Join

```
1 if (wielkość zadania jest mała)
2     policz bezpośrednio
3 else {
4     podziel zadanie na dwa podzadania
5     uruchom (invoke) podzadania i czekaj na wynik
6     połącz wyniki podzadań w wynik całego zadania
7 }
```

Jest to przykład zastosowania strategii *dziel-i-zwyciężaj*

Fork / Join przykład

```
1  class Sumowanie extends RecursiveTask<Long> {
2      private final long start, koniec;
3      Sumowanie(long start, long koniec) {
4          this.start = start;
5          this.koniec = koniec;
6      }
7      @Override
8      public Long compute() {
9          long suma = 0;
10         final long Porcja = 1024;
11         if (koniec - start > Porcja) {
12             long polowa = (koniec - start) / 2;
13             Sumowanie s1 = new Sumowanie(start, start + polowa);
14             s1.fork(); // uruchom równolegle
15             Sumowanie s2 = new Sumowanie(start + polowa + 1, koniec);
16             suma = s2.compute() // oblicz zadanie s2
17                  + s1.join();   // poczekaj na wynik s1
18         } else { suma = sumujBezposrednio(); }
19         return suma;
20 }
```

Fork / Join przykład c.d.

```
1 // Małe sumy obliczaj sekwencyjnie
2     public long sumujBezposrednio() {
3         long suma = 0;
4         for (long i = start; i <= koniec; ++i) {
5             suma += i;
6         }
7         return suma;
8     }
9 }
```

Fork / Join przykład c.d.

```
1 // Przykład uruchomienia zadań
2 public class ForkJoin {
3     public static void main(String [] args) {
4         ForkJoinPool forkJoinPool = new ForkJoinPool();
5         long suma = forkJoinPool.invoke(new Sumowanie(1, 1000000));
6         System.out.println("Suma = " + suma);
7     }
8 }
```

Strumienie

- Języki wysokiego poziomu mają wbudowane mechanizmy, które ułatwiają wykonywanie obliczeń w sposób równoległy bez konieczności tworzenia puli wątków w sposób jawny
- Java 8 zawiera API dla tzw. *strumieni* (ang. streams), które ułatwiają wykonywanie obliczeń na sekwencjach danych w sposób typowy dla programowania funkcyjnego

Przykład – zliczanie liczb pierwszych

```
1  int n = 10_000_000;
2  int [] numbers = new int[ n - 2 ];
3  for (int i = 0; i < numbers.length; ++i) {
4      numbers[i] = i+2;
5  }
6  long start = System.currentTimeMillis();
7  long count = Arrays.stream(numbers).filter(v -> isPrime(v)).count();
8  long stop = System.currentTimeMillis();
9  System.out.printf("Number of prime numbers <= %d equals %d\n", n,count);
10 System.out.printf("Calc. time was: %d [ms]\n", stop - start);
```

Strumienie a równoległość

- API strumieni pozwala na łatwe zrównoleglenie obliczeń

Przykład – wersja sekwencyjna

```
1 long count = Arrays.stream(numbers).filter(v -> isPrime(v)).count();
```

Przykład – wersja równoległa

```
1 long count = Arrays.stream(numbers).parallel().filter(v -> isPrime(v)).
      count();
```

Wysokopoziomowe mechanizmy współbieżności

- Klasa `Arrays` w Javie od wersji 8 zawiera metody umożliwiające wykonywanie obliczeń równoległych na elementach tablicy, m.in.
  - `void parallelSort(int[] a, int fromIndex, int toIndex)`
  - `<T> void parallelPrefix(T[] array, BinaryOperator< T> op)`
- Język C# udostępnia szereg wysokopoziomowych mechanizmów w bibliotece Task Paralel Library (TPL) oraz Parallel LINQ (PLINQ)

Kolekcje współbieżne

- Przed wersją 5.0 w Javie można było bezpiecznie korzystać z kolekcji synchronizowanych:
  - `Hashtable`, `Vector`, `Collections.synchronizedMap`
  - były **bezpieczne**, operacje wykonywane z zachowaniem **wzajemnego wykluczania** – metody synchronizowane
  - potencjalnie niższa wydajność niż w przypadku implementacji **niewstrzymywanych** (ang. lock free) oraz **nieczekających** (ang. wait free)
- Java 5.0 wprowadziła zbiór klas implementujących współbieżne kolekcje (ang. concurrent collections)
  - większość operacji może być wykonywana **jednocześnie**
  - potencjalnie dużo większa wydajność
  - nakładanie, dowolny przeplot

Zastosowania kolejek w programach współbieżnych

- Kolejki mogą ułatwić *bezpieczną* komunikację między wątkami

    1. Nadawca wstawia komunikat(y) do współdzielonej kolejki
    2. Odbiorca w dogodnym czasie odbiera kolejny komunikat z kolejki
    3. Odbiorca może wstawić do kolejki komunikat dla nadawcy

Zastosowania kolejek w programach współbieżnych

- Nie ma potrzeby dodatkowej synchronizacji – cały ciężar spoczywa na metodach kolejki współbieżnej
- Jedna kolejka może być wykorzystywana przez wiele wątków
- Można utworzyć osobne kolejki dla każdej grupy / kategorii wątków / zadań
- Podobny schemat można zastosować dla współdzielonych zasobów

Kolekcje współbieżne – BlockingQueue

- Interfejs `BlockingQueue` rozszerza interfejs kolejki `Queue` o dodatkowe metody
- Metoda `put(e)` wstawia do kolejki, wstrzymuje wątek, jeżeli w kolejce nie ma miejsca
- Metoda `take()` pobiera element z kolejki, wstrzymuje wątek, jeżeli kolejka jest pusta – czeka na wstawienie nowego elementu
- Metoda `poll(long timeout, TimeUnit u)` pobiera element z kolejki, oczekując nie dłużej, niż podany czas na jego dodanie
- Nie ma metody, która „zamyka” kolejkę

BlockingQueue – przykład

```
1  public class KolejkaBlokujaca {
2      public static void main(String [] args) {
3          BlockingQueue<Integer> q = new ArrayBlockingQueue<Integer>(3);
4          Producent p = new Producent(q);
5          Konsument c1 = new Konsument("1", q);
6          Konsument c2 = new Konsument("2", q);
7          new Thread(p).start();
8          new Thread(c1).start();
9          new Thread(c2).start();
10     }
11 }
```

BlockingQueue – przykład

```
1  class Konsument implements Runnable {
2      private final String id;
3      private final BlockingQueue<Integer> queue;
4      Konsument(String id, BlockingQueue<Integer> q) {
5          this.id = id; queue = q;
6      }
7      public void run() {
8          try {
9              int x = queue.take();
10             while (x != Integer.MAX_VALUE) {
11                 konsumuj(x);
12                 x = queue.take();
13             }
14             q.put(x); // Zwróć znacznik stopu dla innych wątków
15         } catch (InterruptedException ex) { }
16     }
17     void konsumuj(Integer x) {
18         System.out.printf("Konsument [%s], otrzymałem: %d\n", id, x);
19     }
20 }
```

BlockingQueue – przykład

```
1  class Producent implements Runnable {
2      private final BlockingQueue<Integer> queue;
3      private int licznik = 0;
4      Producent(BlockingQueue<Integer> q) { queue = q; }
5      public void run() {
6          try {
7              for (int i = 0; i < 10; ++i) { queue.put(produkuj()); }
8              queue.put(Integer.MAX_ VALUE); // Znacznik stopu
9          } catch (InterruptedException ex) { }
10     }
11     Integer produkuj() { return licznik++; }
12 }
```

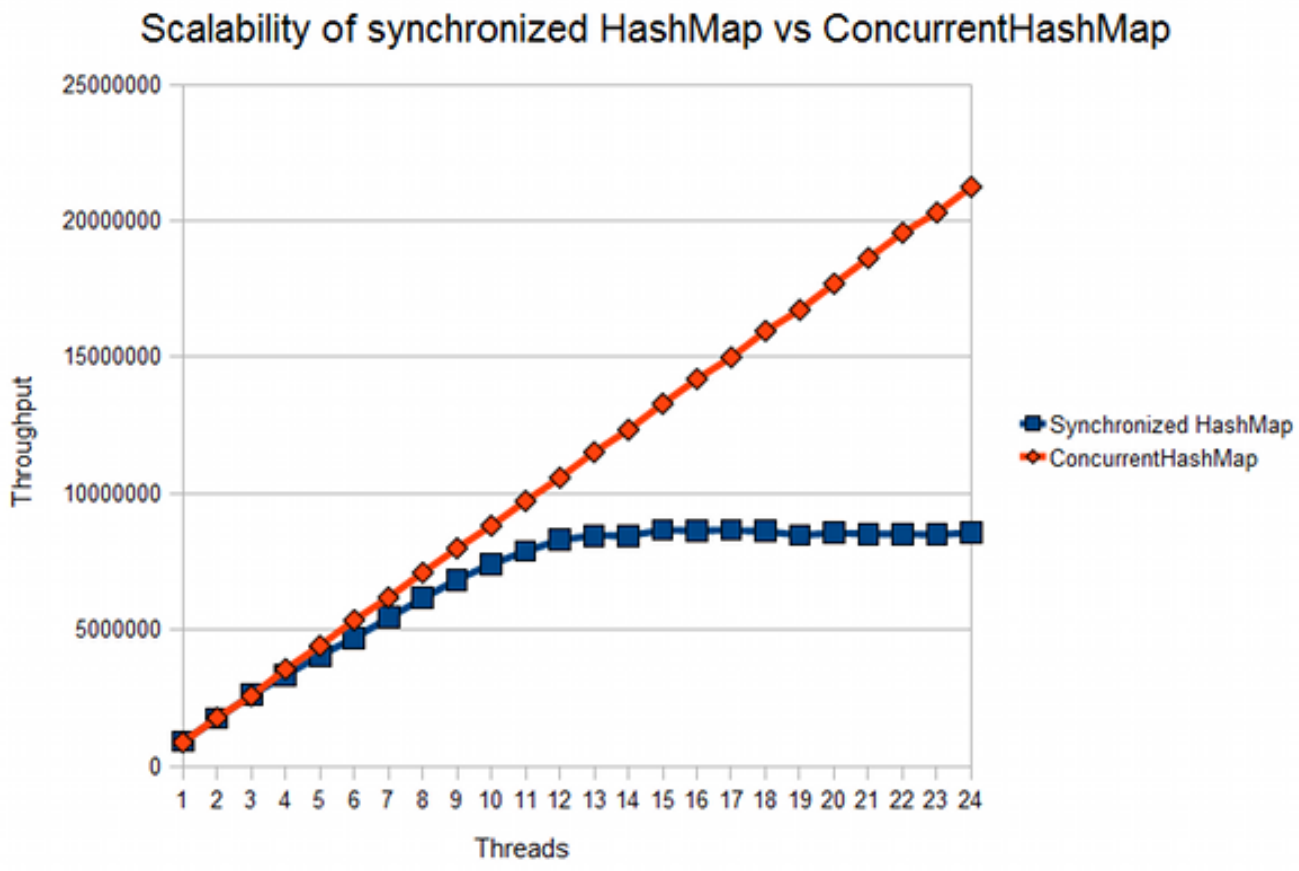
Kolekcje współbieżne – ConcurrentLinkedQueue

- Nieczekająca (ang. wait-free) implementacja kolejki
- Potencjalnie większa wydajność, jednak trzeba testować
- Uwaga na metodę `size()`, której złożoność nie jest O(1), ale może być O(n) – wymaga przejścia po wszystkich elementach

Kolekcje współbieżne – ConcurrentHashMap

- `ConcurrentHashMap`
  - Zamiennik dla `Hashtable` oraz `Collections.synchronizedMap`
  - Pozwala na jednoczesny odczyt przez kilka wątków
  - Pozwala na zapis jednocześnie z odczytem
  - Iteratory nie rzucają wyjątku `ConcurrentModificationException`
  - Operacja odczytu `get()` „widzi" rezultaty wykonanych w całości (zakończonych) operacji modyfikacji takich jak `put()` i `remove()`
  - Pozwala na nakładanie się domyślnie 16 operacji modyfikacji – można podać więcej w parametrze konstruktora

Kolekcje współbieżne – wydajność



Rysunek 2: Porównanie skalowalności ConcurrentHashMap z HashMap. (Oś Y pokazuje liczbę operacji w jednostce czasu. Komputer: Intel i7-720QM Quad Core Hypterthreading, Hotspot JVM wersja 1.6.0_18, Windows 7. Źródło: www.javamex.com)

Kolekcje współbieżne

- `CopyOnWriteArrayList`
  - Współbieżny odpowiednik `ArrayList`
  - Dobra efektywność, gdy operacji odczytu jest znacznie więcej, niż operacji zapisu, ponieważ modyfikacja powoduje utworzenie *kopii listy*
  - Przykładowe zastosowanie to listy odbiorców zdarzeń (ang. event listeners)

Kolekcje współbieżne

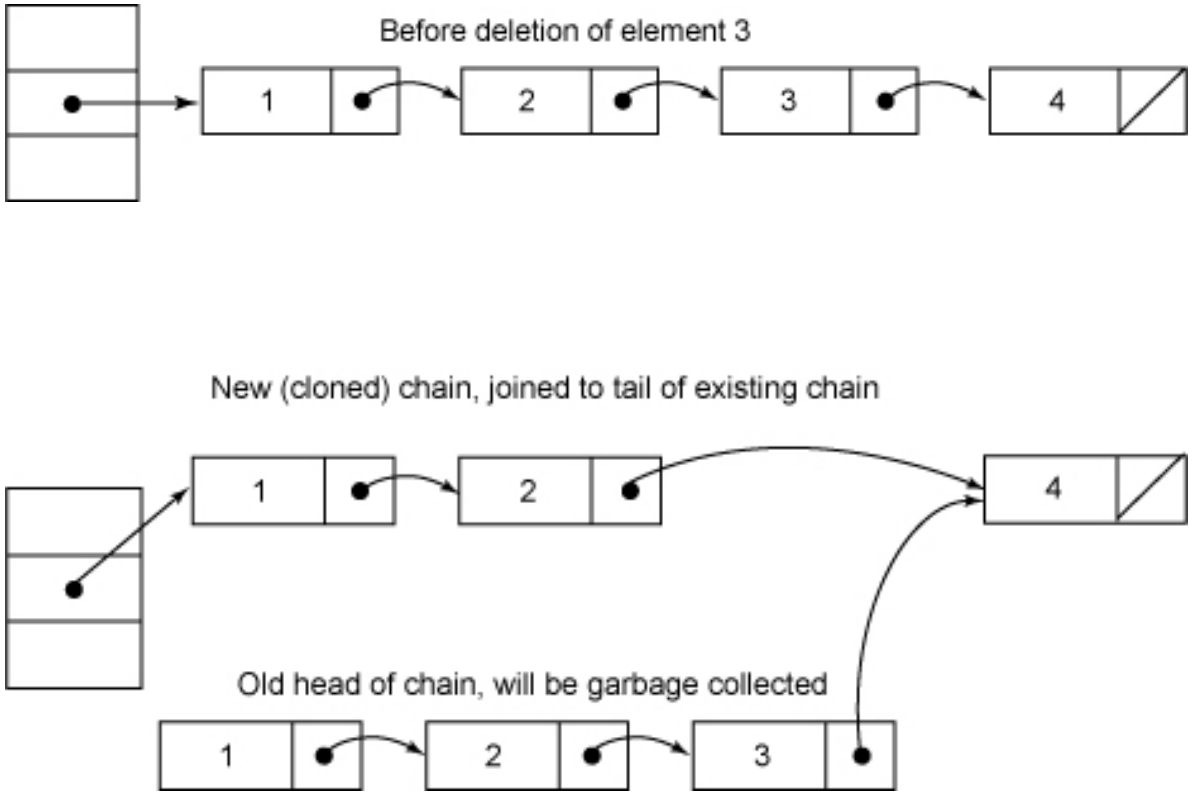
- Metody kolekcji współbieżnych wykonywane są atomowo, jednak wykonanie w sposób atomowy ciągu operacji na kolekcji wymaga synchronizacji
- Należy pamiętać, że metody takie jak `size()`, czy `isEmpty()` zwracają bieżący stan, który może stać się nieaktualny w wyniku jednoczesnej modyfikacji przez inne wątki.

Iteratory w kolekcjach współbieżnych

- Iteratory w kolekcjach synchronizowanych powodują rzucenie wyjątku `ConcurrentModificationException`, gdy zajdzie próba modyfikacji kolekcji podczas jej przeglądania za pomocą iteratora (ang. fast-fail mechanism)
- Iteratory w kolekcjach współbieżnych z `java.util.concurrent` nie rzucają wyjątku `ConcurrentModificationException` w przypadku jednoczesnej modyfikacji
  - mogą nie zobaczyć zmian w kolekcji dokonanych podczas iterowania
  - w najgorszym przypadku zobaczą nieaktualne dane (ang. stale data)

Iteratory w kolekcjach współbieżnych

Ilustracja usuwania elementu z `ConcurrentHashMap`



Współbieżność a interfejs użytkownika

- Wielowątkowość może być pomocna w zapewnieniu płynnego działania interfejsu użytkownika (GUI)
- Interakcje użytkownika z GUI aplikacji generują *zdarzenia*
- Pogodzenie programowania zdarzeniowego ze współbieżnym jest trudne – obie koncepcje charakteryzują się nieliniowym przebiegiem sterowania[1]
- Większość bibliotek GUI nie jest wielowątkowa, tj. trzeba zachować ostrożność w przypadku interakcji wątków w GUI

[1] John Ousterhout. "Why Threads Are A Bad Idea (for most purposes)". Sun Microsystems Laboratories (1995)

Współbieżność a interfejs użytkownika

- Zazwyczaj organizacją, wyświetlaniem i obsługą GUI zajmuje się *jeden, wyróżniony wątek*, który zajmuje się m. in. *dyspozycją* zdarzeń do zarejestrowanych metod obsługi
- Długotrwałe obliczenia prowadzone są w wątkach roboczych, które komunikują się z wątkiem GUI
- W celu zapewnienia poprawnego działania programu należy pamiętać o wszystkich zasadach bezpieczeństwa, jak w przypadku każdego innego typu programu współbieżnego

Współbieżność GUI na przykładzie Java Swing

Standardowo Java udostępnia trzy biblioteki GUI:

- Abstract Window Toolkit (AWT) – korzysta bezpośrednio z kontrolek GUI udostępnianych przez OS
- Swing – nowsza propozycja – implementuje własne kontrolki GUI, niezależne od OS. Oferowana lista kontrolek jest bogatsza niż w przypadku AWT
- JavaFX – zespół bibliotek i narzędzi do tworzenia aplikacji typu Rich Internet Application działających w różnych OS, przeglądarkach internetowych oraz urządzeniach mobilnych. Odpowiednik Adobe Flex, czy Microsoft Silverlight.

Dostępne są również alternatywne biblioteki GUI, m.in.: SWT, Vaadin

Współbieżność GUI na przykładzie Java Swing

Wątki w aplikacji korzystającej ze Swing można podzielić na:

- wątki początkowe, w tym wątek główny aplikacji – inicjują tworzenie interfejsu użytkownika
- wątek dyspozycji zdarzeń (ang. event dispatch) – obsługuje lub przekierowuje zdarzenia – większość kodu obsługi zdarzeń *wykonuje się w tym wątku*
- wątki robocze – szczególnie przydatne dla długotrwałych obliczeń

Java Swing – wątki początkowe

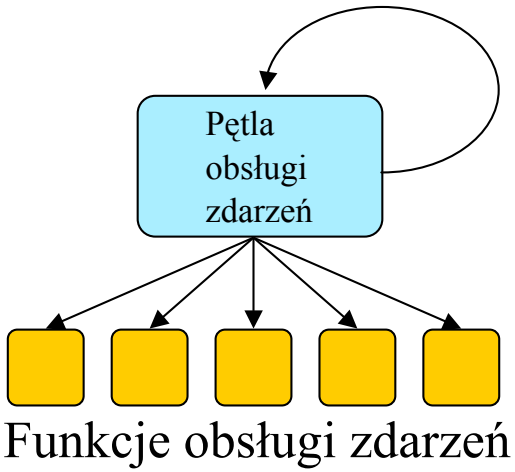
- Wątek początkowy tworzy obiekt impl. `Runnable` odpowiedzialny za utworzenie GUI, który jest przekazywany do wątku dyspozycji zdarzeń (ang. event dispatch thread) za pomocą
  - `javax.swing.SwingUtilities.invokeLater`
  - `javax.swing.SwingUtilities.invokeAndWait`

Przykład z tutoriala Swing (docs.oracle.com)

```
1 SwingUtilities.invokeLater(new Runnable() {
2     public void run() {
3         createAndShowGUI();
4     }
5 });
```

Java Swing – wątek dyspozycji zdarzeń

- Zarządzaniem GUI Swing zajmuje się **wątek dyspozycji zdarzeń** (ang. event dispatch thread)
- Większość metod klas z biblioteki Swing nie może być wywoływana bezpiecznie z innych wątków *bezpośrednio* (nie są *thread safe*)
- Metody *bezpieczne* oznaczone są odpowiednio w API, np. `JTextField.setText()`
- W celu sprawdzenia, czy bieżąca metoda wykonywana jest w wątku GUI można użyć metody `javax.swing.SwingUtilities.isEventDispatchThread()`



Java Swing – wątek obsługi zdarzeń

- Obsługa zdarzeń powinna być *jak najkrótsza*
- Długotrwałe obliczenia (operacje) powinny być wykonywane w wątkach roboczych
- Wątki robocze mogą modyfikować elementy GUI Swing przez zlecenie zadań za pomocą metod:
  - `javax.swing.SwingUtilities.invokeLater`
  - `javax.swing.SwingUtilities.invokeAndWait`
- Uproszczenie komunikacji w wątkiem obsługi zdarzeń można uzyskać dzięki dziedziczeniu z klasy `SwingWorker`

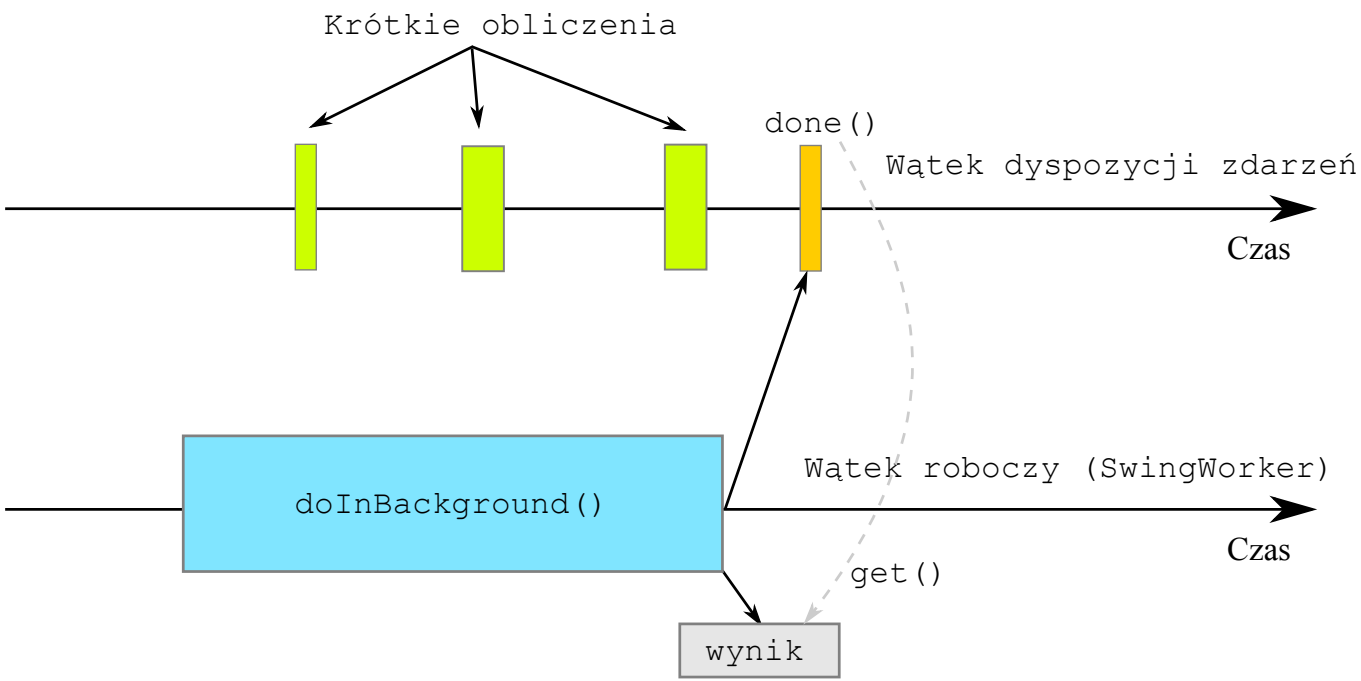
Java Swing – przykład 1

Przykład zastosowania `SwingUtilities.invokeLater` do utworzenia okna

```
1  public class Main {
2    public static void main(final String[] args) {
3      SwingUtilities.invokeLater(new Runnable() {
4        @Override
5        public void run() {
6          final JFrame frame = new JFrame();
7          frame.setTitle("Test Frame");
8          frame.setSize(600, 400);
9          frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10         frame.setVisible(true);
11       }
12     });
13   }
14 }
```

Java SwingWorker

- Klasa `SwingWorker` udostępnia łatwy interfejs dla wykonywania długotrwałych zadań w wątku roboczym i interakcji z wątkiem dyspozycji zdarzeń
- Obliczenia wykonywane są w metodzie `doInBackground()`
- Po zakończeniu obliczeń w wątku dyspozycji zdarzeń wykonywana jest metoda `done()`, która może pobrać wynik obliczeń za pomocą metody `get()`



Java SwingWorker

- W celu informowania o postępie obliczeń klasa `SwingWorker` udostępnia metody
  - `publish` w celu publikowania pośrednich wyników
  - `progress` w celu przetwarzania pośrednich wyników w wątku GUI