Wyższa Szkoła Informatyki Stosowanej i Zarządzania

pod auspicjami Polskiej Akademii Nauk

WYDZIAŁ INFORMATYKI

Kierunek INFORMATYKA

Studia I stopnia (dyplom inżyniera)

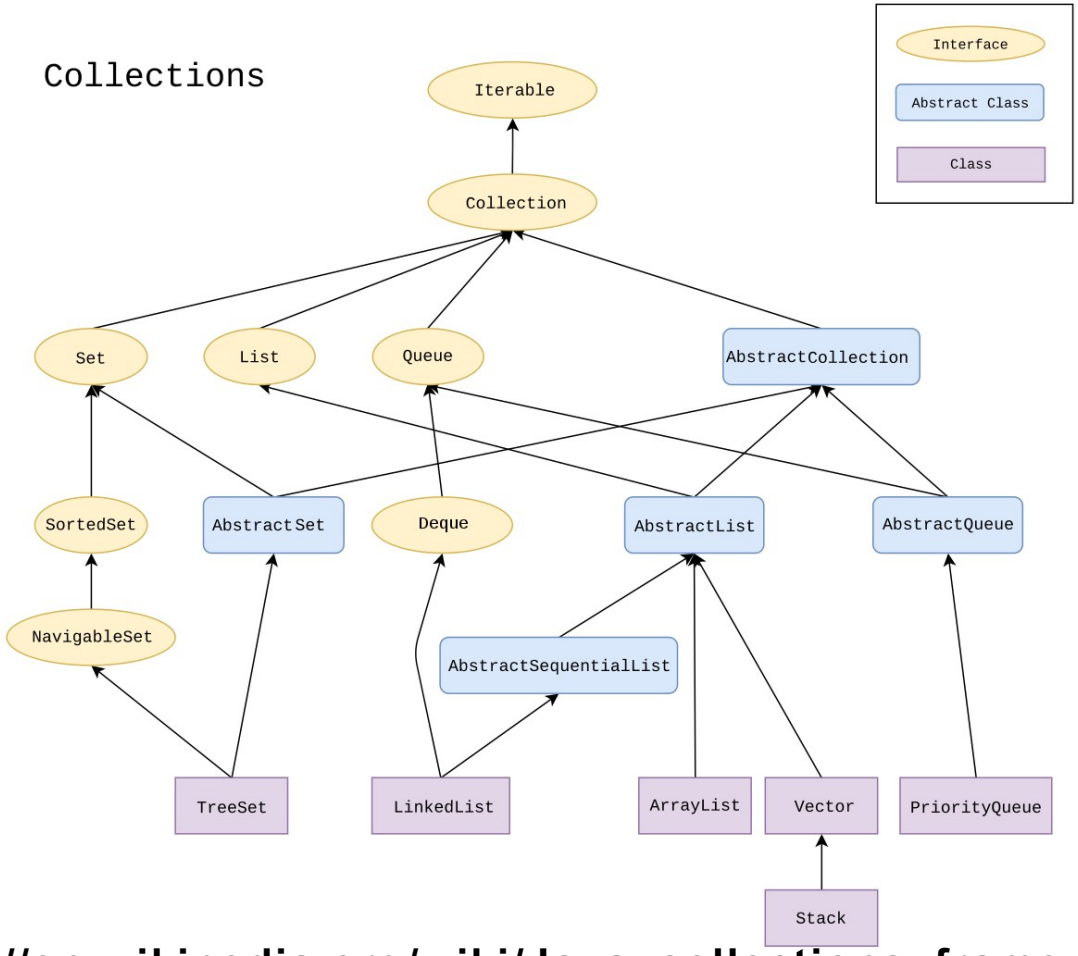
Język Java – wykład 6

dr inż. Łukasz Sosnowski
lukasz.sosnowski@wit.edu.pl
sosnowsl@ibspan.waw.pl
l.sosnowski@dituel.pl

www.lsosnowski.pl

Część 1 – typowe kolekcje w języku JAVA
cz.2 – klasy kolekcji

Hierarchia



*** - źródło w https://en.wikipedia.org/wiki/Java_collections_framework**

Klasy kolekcji

- Klasy implementujące omówione uprzednio interfejsy. Część z nich dostarcza pełnych implementacji, inne to klasy abstrakcyjne, także jedynie dostarczają szkieletowej implementacji wymagającej uzupełnienia własnym kodem.
- Wiele klas jest jednak gotowych do użycia i mocno zoptymalizowanych co powoduje częste ich użycie we własnych programach.
- Klasy te nie dostarczają implementacji synchronizowanej (co może być istotne w programach wielowątkowych). Istnieje jednak możliwość doimplementowania tej funkcjnalności lub skorzystania z pakietu współbieżności który stosowne klasy dostarcza.

Klasa ArrayList

- Rozszerza klasę AbstractList i implementuje interfejs List.
- Klasa sparametryzowana w postaci ArrayList<T>, gdzie T określa typ obiektu przechowywanego w liście.
- Klasa obsługuje dynamiczne tablice, które mogą zmieniać wielkość w zależności od potrzeb.
- Klasa definiuje 3 konstruktory: ArrayList(), ArrayList(Collection<? extends T) c) i ArrayList(int capacity). Pierwszy tworzy pustą tablicę dynamiczną, drugi tworzy tablicę i inicjalizuje ją elementami kolekcji przekazanej w argumencie, a trzeci tworzy tablicę o przekazanym rozmiarze.
- Klasa dostarcza metodę umożliwiająca pobranie tablicy elementów T[] toArray(T array[])

Klasa LinkedList

- Rozszerza klasę AbstractSequentialList i implementuje interfejs List oraz Queue.
- Dane przechowywane są w strukturze listy.
- Klasa sparametryzowana w postaci: LinkedList<T>, gdzie T jest typem obiektu przechowywanego na liście.
- Dostarcza 2 konstruktory: LinkedList() i LinkedList(Collection<? Extends T> c). Pierwszy tworzy pustą listę, a drugi tworzy listę zawierającą elementy kolekcji przekazanej w argumencie.
- Klasa dostarcza implementacji obu interfejsów, m.in.. metod addFirst(), offerFirst() czy też addLast(), offerLast(), removeLast czy też pollLast. Ze wzlędu na interfejs List dostarczone są również implementacje metod działających po wskazanym indeksie.

Klasa HashSet

- Rozszerza klasę AbstractSet i implementuje interfejs Set.
- Do przechowywania danych używa tablicy mieszającej.
- Klasa sparametryzowana w postaci: Set<T>, gdzie T oznacza typ obiektu przechowywanego w zbiorze.
- Do określenia miejsca składowania elementu używana jest funkcja skrótu liczona na bazie elementu.
- Dostarcza 4 konstruktory: HashSet(), HashSet(Collection<? Extends T> c), HashSet(int capacity), HashSet(int capacity, float fillRatio), gdzie odpowiednio: tworzy domyślny obiekt tablicy mieszającej, inicjalizuje tablicę mieszającą elementami przekazanej kolekcji, inicjalizuje pojemność tablicy przekazaną wartością, a ostatni analogicznie lecz ze wsp. wypełnienia.
- Ten typ zbioru nie gwarantuje kolejności ani posortowania elem.

Klasa LinkedHashSet

- Rozszerza klasę HashSet i nie dodaje żadnych własnych metod.
- Klasa sparametryzowana w postaci: LinkedHashSet<T>, gdzie T oznacza typ obiektu przechowywanego w zbiorze.
- Klasa przechowuje dodatkowo listę elementów zbioru w kolejności wstawienia, dzięki czemu zachowuje kolejność elementów w zbiorze zgodną z kolejnością ich wstawiania (przy użyciu iteratora).
- Dodatkowo metoda toString() również wykorzystuje tę dodatkową listę do iterowania i raportowania elementów zbioru.

Klasa TreeSet

- Rozszerza AbstractSet i implementuje interfejs Set oraz SortedSet.
- Kolekcja przechowuje swoje elementy w strukturze drzewiastej. Umieszczane elementy sortowane są w porządku rosnącym.
- Dostęp i pobieranie elementów jest bardzo szybkie.
- Klasa sparametryzowana w postaci: TreeSet<T>, gdzie T określa typ przechowywanych obiektów.
- Definiuje 4 konstruktory: TreeSet(), TreeSet(Collection<? Extends T> c), TreeSet(Comparator<? Super T> comp), TreeSet(SortedSet<T> ss) odpowiednio: tworzy puste drzewo, tworzy drzewo na bazie elementów kolekcji c, tworzy puste drzewo które będzie wykorzystywało comp do sortowania elementów, tworzy drzewo zawierające elementy przekazane jako SortedSet.

Klasa PriorityQueue

- Rozszerza klasę AbstractQueue i implementuje interfejs Queue
- Klasa kolejki priorytetowej, w której priorytet ustalany jest przez komparator.
- Klasa sparametryzowana w postaci: PriorityQueue<T>, gdzie T oznacz typ elementów kolejki.
- Definiuje 7 konstruktorów wykorzystywanych do utworzenia obiektu kolejki priorytetowej dla różnych parametrów, np.. posiadanej kolekcji obiektów, innej kolejki priorytetowej lub sortowanego zbioru.
- Jeden z konstruktorów umożliwia przekazanie komparatora. Jeśli w momencie inicjalizacji komparator nie zostanie przekazany, zostanie użyty domyślny, układający elementy kolejny w sposób rosnący.

Klasa ArrayDeque

- Rozszerza klasę AbstractCollection i implementuje interfejs Deque.
- Klasa nie dodaje żadnych własnych metod. Dostarcza jedynie implementację już zdefiniowanych.
- Tworzy dynamiczną tablicę o nieograniczonej pojemności
- Klasa sparametryzowana z postaci: ArrayDeque<T> gdzie T oznacza typ obiektów przechowywanych.
- Definiuje 3 konstruktory: ArrayDeque(), ArrayDeque(int size), ArrayDeque(Collection<? extends E> c) odpowiednio: tworzy pustą kolekcję, tworzy kolekcję o określonej pojemności początkowej, tworzy kolekcję wypełniając ją przekazanymi elementami z kolekcji c.
- W każdym z 3 przypadków pojemność jest automatycznie zwiększana.

Iterowanie kolekcji

- Dla przypadku braku modyfikacji kolekcji stosujemy pętlę typu for-each, aby przejść przez wszystkie elementy kolekcji.
- Pętla rozszerzona w postaci:
  for(TypElementuKolekcji elem:ObiektKolekcji){
  // przetwarzanie
  }
- W przypadku konieczności dokonywania modyfikacji typu usunięcie lub wstawienie (dla listy) stosujemy Iterator lub ListIterator:
  Iterator<TypElemetuKolekcji> it = col.iterator();
  while(it.hasNext()){
  it.next();}

Przykład:

```
private List<String> list = null;
      private Queue<String> queue = null;
      private Set<String> set = null;
      private SortedSet<String> sortedSet = null;
      private Collection<String> collection = null;

public MyCollections(String[] items) {
list = new ArrayList<String>(Arrays.asList(items));
queue = new PriorityQueue<String>(Arrays.asList(items));
set = new HashSet<String>(Arrays.asList(items));
sortedSet = new TreeSet<String>(Arrays.asList(items));
collection = new LinkedHashSet<String>(Arrays.asList(items));
}

public void iterateWithForEach() {
            System.out.println("Iterowanie listy:");
            for(String item:list)
                  System.out.print(item+",");
            System.out.println("");
            System.out.println("Iterowanie kolejki:");
            for(String item:queue)
                  System.out.print(item+",");
            System.out.println("");
            System.out.println("Iterowanie zbioru:");
            for(String item:set)
                  System.out.print(item+",");
            System.out.println("");
            System.out.println("Iterowanie zbioru sortowanego:");
            for(String item:sortedSet)
                  System.out.print(item+",");
            System.out.println("");
            System.out.println("Iterowanie kolekcji:");
            for(String item:collection)
                  System.out.print(item+",");
      }
```

```
public void iterateWithItAll() {
            Iterator<String> it = list.iterator();
            String item;
            System.out.println("Iterowanie listy:");
            while(it.hasNext()) {
                  item=it.next();
                  System.out.print(item+",");
            }
            System.out.println("");
            Iterator<String> it2 = queue.iterator();
            System.out.println("Iterowanie kolejki:");
            while(it2.hasNext()) {
                  item=it2.next();
                  System.out.print(item+",");
            }
            System.out.println("");
            Iterator<String> it3 = set.iterator();
            System.out.println("Iterowanie zbioru:");
            while(it3.hasNext()) {
                  item=it3.next();
                  System.out.print(item+",");
            }
            System.out.println("");
            Iterator<String> it4 = sortedSet.iterator();
            System.out.println("Iterowanie zbioru sortowanego:");
            while(it4.hasNext()) {
                  item=it4.next();
                  System.out.print(item+",");
            }
            System.out.println("");
            Iterator<String> it5 = collection.iterator();
            System.out.println("Iterowanie kolekcji:");
            while(it5.hasNext()) {
                  item=it5.next();
                  System.out.print(item+",");
            }
      }
```

Część 2 – Programowanie wielowątkowe

Informacje podstawowe

- Wielozadaniowość w ujęciu danego programu realizowana jest na poziomie wątków.
- Dany program może jednocześnie wykonywać dwa lub więcej wątków, czyli wykonywać wiele zadań w ramach swego działania.
- Zaletą programów wielowątkowych jest optymalizacja użycia procesora poprzez wykorzystanie okresów bezczynności procesora.
- Programy wielowątkowe mogą działać zarówno w systemach jednoprocesorowych jak i wielordzeniowych. W tym pierwszym przypadku proces jest współdzielony a w drugim występuje realne zrównoleglenie zadań.
- Wątek może znajdować się w jednym z określonych stanów: *wykonywany, gotowy do wykonania, zawieszony, zablokowany,* etc.

Interfejs Runnable i klasa Thread

- Wielowątkowość w JAVA realizowana jest poprzez dziedziczenie z klasy *Thread* lub implementację interfejsu *Runnable*.
- Klasa *Thread* implementuje metody do zarządzania wątkiem:

Metoda	Opis
final String getName()	Zwraca nazwę wątku.
final int getPriority()	Zwraca priorytet wątku.
final boolean isAlive()	Sprawdza czy wątek jest nadal wykonywany.
final void join()	Metoda uruchamiająca mechanizm oczekiwania na zakończenie wątku dla którego została wykonana.
void run()	Punkt wejścia wątku, główna metoda wykonująca wątek,
static void sleep(long millis)	Zawieszenie wątku na określony w parametrze czas.
void start()	Metoda rozpoczynająca wykonywanie wątku poprzez metodę run().

Tworzenie wątków

- Wątek tworzony jest w momencie utworzenia obiektu klasy *Thread*.
- Java umożliwia tworzenie obiektów wątków poprzez klasę implementującą interfejs *Runnable* lub klasę pochodną klasy Thread. Jednakże oba te podejścia używają klasy *Thread* do utworzenia obiektu wątku.
- Interfejs *Runnable* definiuje jedną metodę: *public void run(),* które definiuje główne działanie wątku.
- W celu utworzenia wątku definiujemy obiekt klasy implementującej interfejs *Runnable*, a następnie przekazujemy go w konstruktorze klasy Thread, np. *Thread(Runnable objRunnable)*
- Następnie w celu rozpoczęcia działania wątku, należy wywołać na nim metodę *start()*.

Tworzenie wątków c.d.

- Drugim sposobem stworzenia wątku jest zdefiniowanie klasy pochodnej klasy *Thread*.
- W klasie pochodnej należy przesłonić metodę *run().*
- Możliwe jest również przesłonięcie innych metod, choć nie jest to wymagane.
- W klasie pochodnej definiujemy konstruktor lub konstruktory, które zamierzamy używać. Konstruktory te będą wywoływać odpowiednie konstruktory klasy bazowej.
- W celu stworzenia wątku tworzymy obiekt klasy pochodnej.
- W celu uruchomienia wątku wywołujemy na zmiennej klasy pochodnej metodę start();
- UWAGA: wywołanie metody run() nie powoduje wykonania jej w oddzielnym wątku, a jedynie w wątku głównym aplikacji!

Przykład:

```
public class SimpleThread implements Runnable {
String threadName;
public SimpleThread(String threadName) {
      this.threadName=threadName;
}
public void run() {
System.out.println("Wątek "+threadName+" rozpoczyna działanie.");
try {
      for (int i = 0; i < 5; i++) {
            Thread.sleep(500);
            System.out.println("Wątek "+threadName+" działa, i="+(i+1));
      }
} catch (InterruptedException e) {
      System.out.println(e);
}
System.out.println("Wątek "+threadName+" kończy działanie.");
}}
@Test
public void simpleThreadTest() {
      System.out.println("Wątek główny rozpoczyna działanie");
      SimpleThread st = new SimpleThread("Nr1");
      Thread newTh = new Thread(st);
      newTh.start();
      for(int i=0;i<30;i++) {
            System.out.print("_");
            try {
                  Thread.sleep(100);
            } catch (InterruptedException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
      }
      System.out.println("Wątek główny kończy działanie");
}
```

Przykład c.d.:

```
public class SimpleThread2 implements Runnable{
	private Thread thread;
	private SimpleThread2(String threadName) { thread = new Thread(this,threadName);}
	public static SimpleThread2 createAndRun(String threadName) {
		SimpleThread2 st = new SimpleThread2(threadName);
		st.thread.start();
		return st;
	}
	public void run() {
		System.out.println("Wątek "+thread.getName()+" rozpoczyna działanie.");
		try {
			for (int i = 0; i < 5; i++) {
				Thread.sleep(500);
				System.out.println("Wątek "+thread.getName()+" działa, i="+(i+1));
			}
		} catch (InterruptedException e) {
			System.out.println(e);
		}
		System.out.println("Wątek "+thread.getName()+" kończy działanie.");
	}}
@Test
	public void simpleThread2Test() {
		System.out.println("Wątek główny rozpoczyna działanie");
		SimpleThread2 st = SimpleThread2.createAndRun("Nr1");
		for(int i=0;i<30;i++) {
			System.out.print("_");
			try {
				Thread.sleep(100);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
		}
		System.out.println("Wątek główny kończy działanie");
	}
```

Przykład c.d.:

```
public class FirstThread extends Thread {
	public FirstThread(String threadName) {
		super(threadName);
	}
	public void run() {
		System.out.println("Wątek "+getName()+" rozpoczyna działanie.");
		try {
			for (int i = 0; i < 5; i++) {
				Thread.sleep(500);
				System.out.println("Wątek "+getName()+" działa, i="+(i+1));
			}
		} catch (InterruptedException e) {
			System.out.println(e);
		}
		System.out.println("Wątek "+getName()+" kończy działanie.");
	}
}
	@Test
	public void firstThreadTest() {
		System.out.println("Wątek główny rozpoczyna działanie");
		FirstThread ft = new FirstThread("Nr1");
		ft.start();
		for(int i=0;i<30;i++) {
			System.out.print("_");
			try {
				Thread.sleep(100);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
		System.out.println("Wątek główny kończy działanie");
	}
```

Metoda isAlive() i join()

- W celu sprawdzenia czy wątek nadal działa możemy wywołać na obiekcie metodę isAlive().
- Umożliwia to ustalenie stanu wątku oraz wykonanie ewentualnej obsługi oczekiwania wątku głównego na zakończenie prac wątków potomnych.
- Metoda join() powoduje włączenie oczekiwania na wątek dla którego została wywołana, aż ten wątek zakończy swoje działanie.
- Inne wersje metody *join* pozwalają na określenie maksymalnego czasu oczekiwania na wątek.

Priorytety wątków

- Priorytet wątku potencjalnie decyduje jaki przydział czasu procesora otrzyma dany wątek.
- Na przydział czasu mają wpływ również inne elementy, np.. oczekiwanie na zasoby. Wtedy wątki z niższym priorytetem mogą być wykonane przed wątkami z priorytetami wyższymi.
- Priorytet ustawiany domyślnie jest na taki sam jak priorytet wątku głównego. W celu zmiany ustawienia priorytetu używamy metody *setPriority(int priorytet),* gdzie parametr metody przyjmuje wartość całkowitą z przedziału <MIN_PRIORITY, MAX_PRIORITY> gdzie pierwszy posiada wartość 1 a drugi 10.
- Przywrócenie wątkowi zwykłego priorytetu następuje poprzez ustawienie NORM_PRIORITY (wartość 5).

Przykład:

```
@Test
	public void isAliveThreadTest() {
		System.out.println("Wątek główny rozpoczyna działanie");
		FirstThread ft1 = new FirstThread("Nr1");
		FirstThread ft2 = new FirstThread("Nr2");
		FirstThread ft3 = new FirstThread("Nr3");
		ft1.start(); ft2.start();  ft3.start();
		do {
			System.out.print("_");
			try {
				Thread.sleep(100);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}while(ft1.isAlive() || ft2.isAlive() || ft3.isAlive());
		System.out.println("Wątek główny kończy działanie");
	}
@Test
	public void joinThreadTest() {
		System.out.println("Wątek główny rozpoczyna działanie");
		FirstThread ft1 = new FirstThread("Nr1");
		FirstThread ft2 = new FirstThread("Nr2");
		FirstThread ft3 = new FirstThread("Nr3");
		ft1.start(); ft2.start();  ft3.start();
		try {
			ft1.join();
			ft2.join();
			ft3.join();
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		System.out.println("Wątek główny kończy działanie");
	}
```

Przykład c.d.:

```
@Test
	public void priorityAndJoinThreadTest() {
		System.out.println("Wątek główny rozpoczyna działanie");
		FirstThread ft1 = new FirstThread("Nr1");
		FirstThread ft2 = new FirstThread("Nr2");
		FirstThread ft3 = new FirstThread("Nr3");
		ft3.setPriority(Thread.MAX_PRIORITY);
		ft2.setPriority(Thread.MIN_PRIORITY);
		ft1.setPriority(Thread.NORM_PRIORITY-1);
		ft1.start();
		ft2.start();
		ft3.start();
		try {
			ft1.join();
			ft2.join();
			ft3.join();
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
		System.out.println("Wątek główny kończy działanie");
	}
```

Synchronizacja

- W programach wielowątkowych może wystąpić konieczność współdzielenia pewnych zasobów, do których dostęp może być udzielony w danym momencie tylko jednemu wątkowi. Mechanizm zapewnienia dostępu w takim trybie nazywany jest **synchronizacją.**
- Realizacja synchronizacji w JAVA odbywa się poprzez zastosowanie koncepcji **monitora** oraz **blokad**. Monitor kontroluje dostęp do obiektu poprzez ustawianie blokad. Po ustawieniu blokady obiekt nie jest dostępny dla innych wątków. Po zakończeniu używania obiektu blokada jest zwalniana.
- Słowo kluczowe *synchronized* umożliwia użycie mechanizmu synchronizacji w kodzie programu. Możliwe jest użycie na poziomie metod oraz instrukcji.

Synchronizacja metod

- Synchronizacja dostępu do metody zapewniana jest w momencie zadeklarowania metody z użyciem słowa kluczowego *synchronized.*
- W momencie wywołania metody synchronizowanej monitor zakłada blokadę na dany obiekt, co powoduje brak możliwości wywołania tej metody oraz żadnej innej metody synchronizowanej tego obiektu przez inne wątki.
- W momencie blokady inne waki które chcą wykonać metodę synchronizowaną przechodzą w stan oczekiwania, aż do momentu zdjęcia blokady.
- Po zakończeniu wykonywania metody synchronizowanej blokada zostaje automatycznie zdjęta.

Synchronizacja instrukcji

- Istnieją sytuacje, w których nie ma możliwości zmiany deklaracji metody na synchronizowaną, lecz trzeba zapewnić zastosowanie mechanizmu synchronizacji. W takiej sytuacji można skorzystać z synchronizowanego bloku instrukcji:

```
synchronized(obRef){
  //Instrukcje wymagające synchronizacji
}
```

- *obRef* stanowi zmienną referencyjną obiektu do którego dostęp wymaga synchronizacji. W momencie rozpoczęcia wykonywania bloku synchronizowanego, żaden inny wątek nie może wykonywać metody dla wskazanego w nim obiektu. Ograniczenie to zostaje zdjęte w momencie opuszczenia bloku synchronizowanego.

Przykład:

```
public class MultiplyArr {
private long product;
public synchronized long multiplySynchArr(int num[]) {
  product = (num.length==0)?0:1;
  for(int el:num) {
      product*=el;
      System.out.println(Thread.currentThread().getName()
      +" obliczył iloczyn częściowy="+product);
      try {
            Thread.sleep(10);
      } catch (InterruptedException e) {
            e.printStackTrace();
      }
 }
      return product;
}
public long multiplyArr(int num[]) {
  product = (num.length==0)?0:1;
  for(int el:num) {
      product*=el;
      System.out.println(Thread.currentThread().getName()
      +" obliczył iloczyn częściowy="+product);
      try {
            Thread.sleep(10);
      } catch (InterruptedException e) {
            e.printStackTrace();
      }
 }
      return product;
}
}
```

```
public class SecondThread extends Thread {
      static MultiplyArr ma = new MultiplyArr();
      int arr[];
      long result;
      public SecondThread(String threadName,int arr[]) {
            super(threadName);
            this.arr=arr;
      }
      public void run() {
            System.out.println("Wątek "+getName()
            +" rozpoczyna działanie.");
            result = ma.multiplySynchArr(arr);
            System.out.println("Wątek "+getName()
            +" obliczył wynik końcowy="+result);
            System.out.println("Wątek "+getName()
            +" kończy działanie.");
      }
}
@Test
public void synchMethodThreadTest() {
      System.out.println("Wątek główny rozpoczyna działanie");
            int arr[] =new int[]{1,2,3,4,5,6,7,8,9,10};
            SecondThread st1 = new SecondThread("W1",arr);
            SecondThread st2 = new SecondThread("W2",arr);
            SecondThread st3 = new SecondThread("W3",arr);
            st1.start();st2.start();st3.start();
            try {
                  st1.join();st2.join();st3.join();
            } catch (InterruptedException e) {
                  // TODO Auto-generated catch block
                  e.printStackTrace();
            }
      System.out.println("Wątek główny kończy działanie");
      }
```

Przykład c.d:

```
public class MultiplyArr {
private long product;
public synchronized long multiplySynchArr(int num[]) {
  product = (num.length==0)?0:1;
  for(int el:num) {
      product*=el;
      System.out.println(Thread.currentThread().getName()
      +" obliczył iloczyn częściowy="+product);
      try {
            Thread.sleep(10);
      } catch (InterruptedException e) {
            e.printStackTrace();
      }
 }
      return product;
}
public long multiplyArr(int num[]) {
  product = (num.length==0)?0:1;
  for(int el:num) {
      product*=el;
      System.out.println(Thread.currentThread().getName()
      +" obliczył iloczyn częściowy="+product);
      try {
            Thread.sleep(10);
      } catch (InterruptedException e) {
            e.printStackTrace();
      }
 }
      return product;
}
}
```

```
public class ThirdThread extends Thread {
      static MultiplyArr ma = new MultiplyArr();
      int arr[];
      long result;
      public ThirdThread(String threadName,int arr[]) {
            super(threadName);
            this.arr=arr;
      }
      public void run() {
            System.out.println("Wątek "+getName()
            +" rozpoczyna działanie.");
            synchronized(ma) {
                  result = ma.multiplyArr(arr);
            }
            System.out.println("Wątek "+getName()
            +" obliczył wynik końcowy="+result);
            System.out.println("Wątek "+getName()
            +" kończy działanie.");
      }
}
@Test
public void synchBlockThreadTest() {
      System.out.println("Wątek główny rozpoczyna działanie");
      int arr[] =new int[]{1,2,3,4,5,6,7,8,9,10};
      ThirdThread tt1 = new ThirdThread("W1",arr);
      ThirdThread tt2 = new ThirdThread("W2",arr);
      ThirdThread tt3 = new ThirdThread("W3",arr);
      tt1.start();tt2.start();tt3.start();
      try {
            tt1.join();tt2.join();tt3.join();
      } catch (InterruptedException e) {
            e.printStackTrace();
      }
            System.out.println("Wątek główny kończy działanie");
      }
```

Komunikacja międzywątkowa

- Na poziomie wątku mamy do dyspozycji metody związane z komunikacją. Metoda *wait*() informująca monitor o oczekiwaniu i czasowym zawieszeniu wątku oraz *notify*() i *notifyAll*(), które wywołane wznawiają oczekujące wątki.
- *notify*() powiadamia oczekujący wątek, a *notifyAll*() wszystkie wątki, a następnie monitor okresla który wątek otrzyma dostęp.
- Metody *wait*(), *notify*() i *notifyAll*() dostępne są dla każdego obiektu, gdyż zostały zdefiniowane w klasie *Object*. Natomiast należy ich używać tylko w kontekście użycia w mechanizmie synchronizacji.
- Metoda wait istnieje w kilku wariantach: *final void wait(), final void wait(long milis), final void wait(long milis, int nanos)*

Problem „zakleszczenia” i „wyścigu”

- Programowanie wielowątkowe niesie ze sobą nowe sytuacje i zagrożenia. Jedną z nich jest **zakleszczenie**, które wstępuje wtedy gdy dwa lub więcej wątków wzajemnie oczekują na siebie i program w wyniku tego zostaje „zawieszony”.
- Inną sytuacją równie groźną jest tzw. **wyścig**, polegający na niekontrolowanym (nie synchronizowanym) dostępie do wspólnych zasobów, w wyniku czego wynik określonej operacji zależny jest od kolejności wątków, które uzyskały do tego zasobu dostęp. Sytuacja ta jest mocno niekomfortowa, ze względu na pojawiające się błędne wyniki, lecz nie w sposób powtarzalny.
- Programowanie wielowątkowe wymaga dodatkowej uwagi przy testowaniu, ale jeszcze bardziej wymaga tworzenia testów jednostkowych oraz testów integracyjnych.

Operacje na wątkach

- Do momentu opublikowania JAVA2 istniały 3 metody w klasie *Thread*, które służyły do wykonywania wstrzymywania, wznawiania oraz kończenia działania wątku. Metody te to:
  - *final void resume()*
  - *final void suspend()*
  - *final void stop()*
- Jednakże od Java 2 metody te zostały uznane za przestarzałe (ang. deprecated) ze względu na pewne problemy które mogły kończy się zakleszczeniem. W momencie uznania tych metod za przestarzałe nie zostały zarekomendowane nowe metody w ich miejsce. Dlatego w przypadku konieczności wykonywania tego rodzaju operacji należy we własnych klasach wprowadzać obsługę stanu zawieszenia oraz kończenia pracy wątku.

Przykład:

```
public class Goods {
private String product;
// True jeśli konsument czeka, False jesli produc. czeka
private boolean production = true;
public synchronized void produce(String product) {
   while (!production) {
   try {
      System.out.println("Producer waits");
       wait();
   } catch (InterruptedException e)  {

Thread.currentThread().interrupt();
              System.out.println("Thread interrupted");
   }
  }
  production = false;
  System.out.println("Produced product:"+product);
  this.product = product;
      notify();
  }
public synchronized String consume() {
    while (production) {
       try {
           System.out.println("Consumer waits");
           wait();
       } catch (InterruptedException e)  {
Thread.currentThread().interrupt();
         System.out.println("Thread interrupted");
         }
       }
     production = true;
     notify();
     return product;
   }
}
```

```
public class Producer  implements Runnable {
    private Goods goods;
   public Producer(Goods goods) {
        this.goods=goods;
   }
    public void run() {
        String products[] = {"Product1","Product2",
"Product3","Product4","Product5","Product6","Product7",
"Product8","Product9","Product10","STOP"};
         for (String product : products) {
            goods.produce(product);
            try {
                  Thread.sleep(500);
            } catch (InterruptedException e)  {
                Thread.currentThread().interrupt();
                System.out.println("Thread interrupted");
            }
        }    }}}
public class Consumer implements Runnable {
    private Goods goods;
    public Consumer(Goods goods) {
        this.goods=goods;
    }
public void run() {
 for(String consumedProduct = goods.consume();
   !"STOP".equals(consumedProduct);
 consumedProduct = goods.consume()) {
System.out.println("Consumed Product:"+consumedProduct);
 try {Thread.sleep(500);} catch (InterruptedException e) {
          Thread.currentThread().interrupt();
          System.out.println("Thread interrupted");
      }
        }
    }
}
```

API współbieżności w JAVA

- W pakiecie *java.util.concurrent* udostępniono tzw. API współbieżności do tworzenia zaawansowanych programów z użyciem programowania współbieżnego. W pakiecie tym znajdują się klasy m.in.: synchronizatorów, puli wątków, menedżerów wykonania, blokad, itp..
- W pakiecie tym znajduje się również szkielet Fork/Join umożliwiający programowanie równoległe, które umożliwia podział zadania na mniejsze oraz uruchomienie jego rozwiązania na rożnych procesorach. Podstawową zaletą tego szkieletu jest łatwość jego użycia oraz wewnętrzna optymalizacja w celu wykorzystania całej dostępnej mocy obliczeniowej w postaci dostępnych procesorów.

Egzekutory

- Interfejs *Concurrent API* udostępnia narzędzia o nazwie **egzekutor** do tworzenia i kontrolowania wątków, stanowiące alternatywę dla zarządzania wątkami z użyciem klasy *Thread*.
- Głównym elementem egzekutora jest interfejs *Executor*, definiujący metodę: *void execute(Runnable thread).*
- Dodatkowo dostępny jest interfejs *ExecutorService* rozszerzający interfejs *Executor*. Dodaje on metody do zarządzania wątkami i umożliwiające ich kontrolę (np. ***shutdown***)
- Ponadto dostępne są róznież implementacje klas egzekutorów: *ThreadPoolExecutorService*, *ScheduledThreadPoolExecutor*, *ForkJoinPool*. Pierwsza zapewnia funkcjonalności do obsługi puli wątków, druga zapewnia funkcjonalność planowania uruchamiania wątków, a trzecia wspiera szkielet *Fork/Join.*

Egzekutory c.d.

- Najczęściej egzekutory używane są poprzez następujące udostępnione metody statyczne:
  - *static ExecutorService newCachedThreadPool()*
  - *static ExecutorService newFixedThreadPool(int liczbaWatkow)*
  - *static ScheduledExecutorService newScheduledThreadPool(int numThreads)*
- Pierwsza metoda zapewnia utworzenie puli wątków do której można w miarę potrzeby dodawać kolejne wątki, a także ponownie wykorzystywać już istniejące.
- Druga metoda tworzy pulę wątków o ustalonej liczności.
- Trzecia metoda tworzy pulę wątków zapewniającą funkcjonalność planowania uruchamiania wątków.
- Każda z trzech metod zwraca referencję do egzemplarza interfejsu *ExecutorService*, którego można użyć do zarządzania pulą.

Przykład:

```
public class FirstThread extends Thread {
	public FirstThread(String threadName) {
		super(threadName);
	}
	public void run() {
		System.out.println("Wątek "+getName()+" rozpoczyna działanie.");
		try {
			for (int i = 0; i < 5; i++) {
				Thread.sleep(100);
				System.out.println("Wątek "+getName()+" działa, i="+(i+1));
			}
		} catch (InterruptedException e) {
			System.out.println(e);
		}
		System.out.println("Wątek "+getName()+" kończy działanie.");
	}
}
	@Test
	public void executorsThreadTest() {
		System.out.println("Wątek główny rozpoczyna działanie");
		ExecutorService es = Executors.newFixedThreadPool(3);
		for(int i=0;i<10;i++) {
			es.execute(new FirstThread("Nr"+(i+1)));
		}
		es.shutdown();
		try {
			es.awaitTermination(1, TimeUnit.MINUTES);
		} catch (InterruptedException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		System.out.println("Wątek główny kończy działanie");
	}
}
```

Klasa zatrzasku - CountDownLatch

- W niektórych sytuacjach istnieje potrzeba aby wątek poczekał na zajście pewnego zdarzenia. Wtedy można użyć klasy zatrzasku w trybie zliczania zaistnienia wymaganej sytuacji lub dodatkowo w nieprzekraczającym zdefiniowanym czasie.
- Klasa posiada konstruktor w postaci: *CountDownLatch(int num).*
- Aby wątek czekał na zwolnienie zatrzasku musi w nim być wywołana metoda await(), posiadająca dwie postacie:
  - `void await() throws InterruptedException`
  - `boolean await(long wait, TimeUnit tu) throws InterruptedException`
- Pierwsza metoda wymusza oczekiwanie aż licznik osiągnie wartość zero. Druga wymusza oczekiwanie na wartość zero lecz nie dłużej niż zdefiniowano w przekazanym parametrze (wartość oraz jednostka czasu). Metoda ta zwraca true jeśli licznik osiągnie 0 lub false jeśli został przekroczony czas.

Przykład:

```
public class CountDownLatchThread extends Thread {
	private CountDownLatch latch=null;
	private long limit;
	public CountDownLatchThread(CountDownLatch latch) {
		this.latch=latch;
		this.limit = latch.getCount();
	}
	public void run() {
		for(int i=0;i<limit;i++) {
			System.out.println(""+i);
			latch.countDown();
			try {
				Thread.sleep(200);
			} catch (InterruptedException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
		}
	}
}
@Test
public void latchThreadTest() {
	System.out.println("Wątek główny rozpoczyna działanie");
	CountDownLatch cdl = new CountDownLatch(4);
	ExecutorService es = Executors.newFixedThreadPool(1);
	es.execute(new CountDownLatchThread(cdl));
	try {
		cdl.await();
		es.shutdown();
	} catch (InterruptedException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}
	System.out.println("Wątek główny kończy działanie");
}
```

Konsola:
Wątek główny rozpoczyna działanie
0
1
2
3
Wątek główny kończy działanie

Typ wyliczeniowy – TimeUnit

- Wiele metod klas i interfejsów Concurrent API w parametrze wywołania przyjmuje obiekt TimeUnit reprezentujący granulację czasu.
- Typ wyliczeniowy udostępnia następujące stałe: *DAYS, HOURS, MINUTES, SECONDS, MICROSECONDS, MILLISECONDS. NANOSECONDS.*
- Dodatkowo klasa zawiera szereg metod konwersji. Wybrane z nich to:
  - *long convert(long tval, TimeUnit tu);*
  - *long toMillis(long tval);*
  - *long toSeconds(long tval);*
  - *long toMinutes(long tval);*

Interfejsy Callable i Future

- Interfejs Cuncurrent API udostępnia dwa bardzo użyteczne interfejsy, pozwalające jeszcze prościej rozdzielić wymagane obliczenia do wykonywania w wielu wątkach. Interfejs Callable reprezentuje wątek, który zwraca określoną wartość.
- Interfejs Callable jest sparametryzowany: *interface Callable<V>*
- V określa typ danych zwracany przez zadanie
- Interfejs definiuje tylko jedną metodę: *V call() throws Exception*
- Metoda call określa zadanie do wykonania. Po jego wykonaniu zwracany jest wynik lub wyjątek w przypadku jego braku.
- Do wykonania zadania z typu Callable używamy ExecutorService za pomocą metody: *Future<T> submit(Callable<T> task).*
- Future jest interfejsem sparametryzowanym, który reprezentuje wartość zwróconą w przyszłości przez wątek.

Przykład:

```
public class ArrayIntSum implements Callable<Integer>{
	private int arr[];
	public ArrayIntSum(int arr[]) {
		this.arr = arr;
		System.out.println(Arrays.toString(arr));
	}
	public Integer call() throws Exception {
		int sum = Arrays.stream(arr).reduce(0, (a, b) -> a + b);
		return Integer.valueOf(sum);
	}
}
@Test
	public void callableFutureTest() {
		int arr[] = new int[21];
		Random random = new Random();
		arr = random.ints(21, 10,100).toArray();
		ExecutorService es = Executors.newFixedThreadPool(3);
		Future<Integer> sum1, sum2,sum3,sumAll;
		System.out.println("Start");
		System.out.println("arr="+Arrays.toString(arr));
		sum1 = es.submit(new ArrayIntSum(Arrays.copyOf(arr, 7)));
		sum2 = es.submit(new ArrayIntSum(Arrays.copyOfRange(arr, 7,14)));
		sum3 = es.submit(new ArrayIntSum(Arrays.copyOfRange(arr, 14,21)));
		sumAll = es.submit(new ArrayIntSum(arr));
		try {
			System.out.println("sum1="+sum1.get());
			System.out.println("sum2="+sum2.get());
			System.out.println("sum3="+sum3.get());
			System.out.println("sumAll="+sumAll.get());
		}catch(InterruptedException | ExecutionException e) {
			System.out.println(e);
		}
		es.shutdown();
		System.out.println("Koniec");
	}
```

```
Start
arr=[16, 79, 90, 63, 29, 43, 74, 13, 72, 57,
34, 17, 48, 45, 22, 39, 29, 50, 59, 54, 70]
[16, 79, 90, 63, 29, 43, 74]
[13, 72, 57, 34, 17, 48, 45]
[22, 39, 29, 50, 59, 54, 70]
[16, 79, 90, 63, 29, 43, 74, 13, 72, 57, 34,
17, 48, 45, 22, 39, 29, 50, 59, 54, 70]
sum1=394
sum2=286
sum3=323
sumAll=1003
Koniec
```

Klasa Semaphore

- Klasa ta stanowi implementację klasycznego semafora.
- Pozwala na kontrolowanie dostępu do zasobu za pomocą licznika. Gdy jego wartość jest większa od 0 dostęp jest możliwy, jeśli równa zero dostęp jest zabroniony.
- Licznik semafora wydając zezwolenie zmniejsza swą wartość.
- Gdy zasób nie jest już używany pozwolenie zostaje zwracane i wtedy licznik zwiększa swą wartość.
- W celu uzyskania pozwolenia, watek musi wywołać metodę *acquire*(), która obsługuję wydanie pozwolenia. Można wywołać ją również z parametrem definiującym potrzebę większej liczby pozwoleń.
- W celu zwolnienia pozwolenia należy wywołać metodę release() w jednej z dwóch postaci.

Podsumowanie

- Klasy kolekcji
- Przykłady kolekcji
- Przykłady iteracji poprzez obiekty kolekcji
- Programowanie wielowątkowe:
  - Interfejs Runnable
  - Klasa Thread
  - Metody: alive(), join()
  - Synchronizacja
  - Egzekutory
- Przykłady na poszczególne elementy wielowątkowości

Wyższa Szkoła Informatyki Stosowanej i Zarządzania

pod auspicjami Polskiej Akademii Nauk

WYDZIAŁ INFORMATYKI

Kierunek INFORMATYKA

Studia I stopnia (dyplom inżyniera)

Dziękuję za uwagę!