Programowanie współbieżne
Laboratorium nr 13

Kolekcje a współbieżność

"Iteratory szybko zgłaszające problem [1] nie zostały zaprojektowane do zapewniania poprawności w każdej sytuacji - mają za zadanie szybko wykryć błędy współbieżności i je zgłosić. Implementuje się je w taki sposób by sprawdzały licznik modyfikacji kolekcji: jeżeli licznik ulegnie zmianie w trakcie iteracji, metody hasNext() lub next() zgłoszą wyjątek ConcurrentModificationException. Niestety sprawdzanie wartości nie jest synchronizowane, więc istnieje ryzyko widzenia nieświeżej wartości modyfikacji licznika, więc występuje pewne prawdopodobieństwo niezauważenia modyfikacji."

Przykład użycia iteratora dla zbioru. Który z poniższych przykładów jest poprawny i dlaczego?

```
public class TestCollections {
	public static void main(String[] args) {
		Set<Integer> set = new HashSet<Integer>(new Integer(10));
		set.addAll(Arrays.asList(new Integer[]{9,8,7,6,5,4,3,2,1,0}));

		for (Integer i : set)
			if(i % 2 == 1)
				set.remove(i);

		for (Integer i : set)
			System.out.print(i + ", ");

	}
}
```

```
public class TestCollections {
	public static void main(String[] args) {
		Set<Integer> set = new HashSet<Integer>(new Integer(10));
		set.addAll(Arrays.asList(new Integer[] {9,8,7,6,5,4,3,2,1,0}));

		for (Iterator<Integer> it = set.iterator(); it.hasNext();)
			if (it.next() % 2 == 1)
				it.remove();

		for (Integer i : set)
			System.out.print(i + ", ");

	}
}
```

Klasa `java.util.Collections`:

- `public static final List EMPTY_LIST,`
- `public static final Map EMPTY_MAP,`
- `public static final Set EMPTY_SET,`

niezmienna pusta lista, mapa, zbiór,

- `public static final <T> List<T> emptyList()`
- `public static final <K,V> Map<K,V> emptyMap(),`
- `public static final <T> Set<T> emptySet(),`

zwraca niezmienną pustą: listę, mapę, zbiór,

- `public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c),`
- `public static <T> List<T> unmodifiableList(List<? extends T> list),`
- `public static <K,V> Map<K,V> unmodifiableMap(Map<? extends K,? extends V> m),`
- `public static <T> Set<T> unmodifiableSet(Set<? extends T> s),`
- `public static <K,V> SortedMap<K,V> unmodifiableSortedMap(SortedMap<K,? extends V> m),`
- `public static <T> SortedSet<T> unmodifiableSortedSet(SortedSet<T> s)`

zwraca niemodyfikowalny widok na podaną: kolekcję, listę, mapę, zbiór, posortowaną mapę, posortowany zbiór,

- `public static <T> Collection<T> synchronizedCollection(Collection<T> c),`
- `public static <T> List<T> synchronizedList(List<T> list),`
- `public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m),`
- `public static <T> Set<T> synchronizedSet(Set<T> s),`
- `public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m),`
- `public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)`

zwraca synchronizowaną: kolekcję, listę, mapę, zbiór, posortowaną mapę, posortowany zbiór.

Przykład niemodyfikowalnego widoku na zbiór. Które z poniższych operacji są dozwolone?

```
public class TestCollections {

	public static void main(String[] args) {
		Set<Integer> collection = new HashSet<Integer>(
			Arrays.asList(new Integer[] { 1, 2, 3 }));

		Collection<Integer> unmodifiableCollection =
			Collections.unmodifiableCollection(collection);

		System.out.println("UnmodifiableCollection " + unmodifiableCollection);
		System.out.println("            Collection " + collection);

		try {
			Iterator<Integer> it = unmodifiableCollection.iterator();
			while (it.hasNext()) {
				it.next();
				it.remove();
			}
		} catch (Exception e) {
			System.out.println("UnmodifiableCollection: Exception");
			e.printStackTrace(System.out);
		}

		try {
			Iterator<Integer> it = collection.iterator();
			while (it.hasNext()) {
				it.next();
				it.remove();
			}
		} catch (Exception e) {
			System.out.println("            Collection: Exception");
			e.printStackTrace();
		System.out.println("UnmodifiableCollection " + unmodifiableCollection);
		System.out.println("            Collection " + collection);
	}
}
```

Podczas iterowania kolekcji [8] zwróconej z metody Collections.synchronizedXXX() należy wykonać ręczną synchronizację.
"Implementacja metody toString() [1] w standardowych kolekcjach przechodzi przez wszystkie elementy kolekcji i wywołuje ich metodę toString(), (...)."
"Iteracja [1] często zostaje pośrednio wykonana przez metody hashCode() i equals() kolekcji, które mogą zostać wywołane, gdy kolekcja znajduje się we wnętrzu innej kolekcji. Metody containsAll(), removeAll(), retainAll() i konstruktory przyjmujące kolekcje również stosują iteracje."

Przykład synchronizowanej listy:

```
public class TestCollections  {
	final private static int SIZE = 4;
	public static void main(String[] args) throws InterruptedException {
		ExecutorService e = Executors.newCachedThreadPool();
		for (int i = 0; i < SIZE; i++)
			e.execute(new Lotto());
		e.shutdown();
		e.awaitTermination(1, TimeUnit.DAYS);
		synchronized(Lotto.numbers){
			System.out.println(Lotto.numbers);
		}
	}
}
class Lotto implements Runnable {
	private static List<Integer> privateNumbers =
		new ArrayList<Integer>();
	static List<Integer> numbers =
		Collections.synchronizedList(privateNumbers);
	@Override
	public void run() {
		int i = 0;
		while (i < 10) {
			++i;
			synchronized (numbers) {
				if (numbers.contains(i) == false)
					numbers.add(i);
			}
		}
	}
}
```

Kolekcje bezpieczne wątkowo [2]:

"W pakiecie java.util.concurrent znajdują się następujące szybkie implementacje map, zbiorów uporządkowanych i kolejek:"

```
- public class ConcurrentHashMap<K,V> extends AbstractMap<K,V>
	implements ConcurrentMap<K,V>, Serializable,
- public class ConcurrentSkipListMap<K,V> extends AbstractMap<K,V>
	implements ConcurrentNavigableMap<K,V>, Cloneable, Serializable,
- public class ConcurrentSkipListSet<E> extends AbstractSet<E>
	implements NavigableSet<E>, Cloneable, Serializable,
- public class ConcurrentLinkedQueue<E> extends AbstractQueue<E>
	implements Queue<E>, Serializable.
```

"Kolekcje te [2] zwracają tak zwane słabo spójne iteratory (ang. weakly consistent iterators). Oznacza to, że mogą one (choć nie muszą) odzwierciedlać wszystkie modyfikacje dokonane po ich skonstruowaniu. Nie zwracają one jednak dwukrotnie wartości i nie zgłaszają wyjątku ConcurrentModificationException."
Klasa ConcurrentHashMap podczas blokowania wykorzystuje algorytm zwany blokowaniem paskowym [1].
"(...) implementacja ConcurrentHashMap używa tablicy 16 blokad. Każda z nich chroni $^1/_{16}$ tablicy mieszającej. Kubełek N chroni blokada N mod 16. Zakładając, że funkcja mieszająca mniej więcej po równo rozkłada obiekty w tablicy na podstawie kluczy, każda blokada jest16-krotnie mniej obciążona."

Interfejs `ConcurrentMap`:

- `V putIfAbsent(K key, V value)`

Jeżeli podany klucz nie jest odwzorowany na żadną wartość to odwzoruj go na podaną wartość. Równoważne z atomowym wykonaniem [8]:

```
if (!map.containsKey(key))
    return map.put(key, value);
else
    return map.get(key);
```

- `boolean remove(Object key, Object value)`

Usuwa wpis dla klucza jeżeli klucz jest odwzorowany na podaną wartość. Równoważne z atomowym wykonaniem [8]:

```
if (map.containsKey(key) && map.get(key).equals(value)) {
    map.remove(key);
    return true;
} else return false;
```

- `V replace(K key, V value)`

Zastępuje wpis dla klucza jeżeli klucz jest odwzorowany na jakąś wartość. Równoważne z atomowym wykonaniem [8]:

```
if (map.containsKey(key)) {
    return map.put(key, value);
} else return null;
```

- `boolean replace(K key, V oldValue, V newValue)`

Zastępuje wpis dla klucza jeżeli jest odwzorowany na podaną wartość. Równoważne z atomowym wykonaniem [8]:

```
if (map.containsKey(key) && map.get(key).equals(oldValue)) {
    map.put(key, newValue);
    return true;
} else return false;
```

Tablice kopiowane przy zapisie [2]

"CopyOnWriteArrayList i CopyOnWriteArraySet to bezpieczne wątkowo kolekcje, których mutatory tworzą kopie tablic. Taki sposób działania sprawdza się w sytuacjach, w których liczba wątków iterujących po kolekcji znacznie przewyższa liczbę wątków ją modyfikujących. Utworzony iterator zawiera referencję do aktualnej tablicy. Jeżeli tablica ta zostanie później zmodyfikowana, iterator ten nadal będzie miał starą tablicę, mimo że tablica kolekcji jest zamieniona. Dzięki temu starszy iterator dysponuje spójnym (choć potencjalnie przestarzałym) widokiem, do którego ma dostęp nieobciążony żadnym narzutem synchronizacji."

Który z poniższych przykładów wykona się poprawnie i dlaczego?

```
public class TestCollections {
	public static void main(String[] args) {
		List<Integer> list = new ArrayList<Integer>();
		list.add(1);
		list.add(2);

		try {
			Iterator<Integer> it = list.iterator();
			while(it.hasNext()){
				System.out.print(it.next() + ", ");
				list.add(3);
			}
		} catch (Exception e) {
			e.printStackTrace(System.out);
		}
		System.out.println();

		for (int i : list)
			System.out.print(i + ", ");
	}
}
```

```
public class TestCollections {
	public static void main(String[] args) {
	List<Integer> list = new CopyOnWriteArrayList<Integer>();
		list.add(1);
		list.add(2);

		try {
			Iterator<Integer> it = list.iterator();
			while(it.hasNext()){
				System.out.print(it.next() + ", ");
				list.add(3);
			}
		} catch (Exception e) {
			e.printStackTrace(System.out);
		}
		System.out.println();

		for (int i : list)
			System.out.print(i + ", ");
	}
}
```

Kolejki blokujące

```
public interface BlockingQueue<E> extends Queue<E>
```

Sposoby postępowania w przypadku wywołania operacji, która nie może być wykonana natychmiast.

Na podstawie: Dokumentacja JavaDoc 1.6

	Zgłasza wyjątek	Zwraca specjalną wartość	Blokuje się	Blokuje się na określony czas
Wprowadź (Insert)	add(e)	offer(e)	put(e)	offer(e, time, unit)
Usuń (Remove)	remove()	poll()	take()	poll(time, unit)
Sprawdź (Examine)	element()	peek()	-	-

Implementacje `BlockingQueue`:

```
- public class ArrayBlockingQueue<E> extends AbstractQueue<E>
      implements BlockingQueue<E>, Serializable,
- public class DelayQueue<E extends Delayed> extends AbstractQueue<E>
      implements BlockingQueue<E>,
- public class LinkedBlockingDeque<E> extends AbstractQueue<E>
      implements BlockingDeque<E>, Serializable,
- public class PriorityBlockingQueue<E> extends AbstractQueue<E>
      implements BlockingQueue<E>, Serializable,
- public class SynchronousQueue<E> extends AbstractQueue<E>
      implements BlockingQueue<E>, Serializable.
```

```
public interface BlockingDeque<E> extends BlockingQueue<E>, Deque<E>
```

Sposoby postępowania w przypadku wywołania operacji, która nie może być wykonana natychmiast.

Na podstawie: Dokumentacja JavaDoc 1.6

Pierwszy element				
	Zgłasza wyjątek	Zwraca specjalną wartość	Blokuje się	Blokuje się na określony czas
Wprowadź (Insert)	addFirst(e)	offerFirst(e)	putFirst(e)	offerFirst(e, time, unit)
Usuń (Remove)	removeFirst()	pollFirst()	takeFirst()	pollFirst(time, unit)
Sprawdź (Examine)	getFirst()	peekFirst()	-	-

Ostatni element				
	Zgłasza wyjątek	Zwraca specjalną wartość	Blokuje się	Blokuje się na określony czas
Wprowadź (Insert)	addLast(e)	offerLast(e)	putLast(e)	offerLast(e, time, unit)
Usuń (Remove)	removeLast()	pollLast()	takeLast()	pollLast(time, unit)
Sprawdź (Examine)	getlast()	peekLast()	-	-

Implementacja BlockingQueue:

```
- public class LinkedBlockingDeque<E> extends AbstractQueue<E>
      implements BlockingDeque<E>, Serializable.
```

Przykład użycia `LinkedBlockingQueue`:

```
class Producer implements Runnable {
	private boolean sleep;
	private int i;
	Producer(boolean sleep){
		this.sleep = sleep;
	}
	@Override
	public void run() {
		try {
			while (true)
				produce();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
	final void produce() throws InterruptedException {
		System.out.println("> " + ++i);
		TestCollections.queue.put(i);
		System.out.println(">> " + i);
		if(sleep)
			TimeUnit.SECONDS.sleep(2);
	}
}

class Consumer implements Runnable {
	private boolean sleep;
	private int i;
	Consumer(boolean sleep){
		this.sleep = sleep;
	}
	@Override
	public void run() {
		try {
			while (true)
				consume();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
	final void consume() throws InterruptedException {
		System.out.println("< " + ++i);
		int tmp = TestCollections.queue.take();
		System.out.println("<< " + i + ":" + tmp);
		if(sleep)
			TimeUnit.SECONDS.sleep(2);
	}
}

public class TestCollections {
	static LinkedBlockingQueue<Integer> queue = new LinkedBlockingQueue<Integer>(4);

	public static void main(String[] args) throws Exception {
		new Thread(new Producer(false)).start();
		new Thread(new Consumer(true)).start();
	}
}
```

Inne kolekcje bezpieczne wątkowo [2]:

- `java.util.Vector<E>`,

- `java.util.Hashtable<K,V>`.

Przykładowa treść laboratorium:

1. Stworzyć aplikacje składające się z kilku wątków zapisujących i odczytujący dane przechowywane w zbiorach:
- HashSet z własną synchronizacją wykorzystującą metody lub bloki synchronizowane,
- Collections.synchronizedSet(new HashSet()),
- ConcurrentSkipListSet,
- CopyOnWriteArraySet.
2. Stworzyć aplikację symulującą działanie konserwatora budynku. W budynku mieszka kilka osób, modelowanych jako wątki. Każda z nich co jakiś czas sprawdza stan swojego mieszkania (np. co 10 sekund) i wykrywa awarię z podanym prawdopodobieństwem (np. 0.1). Konserwator modelowany jest także jako wątek. Do komunikacji pomiędzy mieszkańcami, a konserwatorem należy zastosować klasę implementującą kolejkę blokującą BlockingQueue. Każdy z mieszkańców, który zauważy awarię zgłasza ją, czyli dodaje ją do kolejki blokującej. Konserwator pobiera opis awarii z kolejki i naprawia ją. Każda naprawa zajmuje czas (np. losowo wybrany od 1 do 5 sekund). Jeżeli nie ma zgłoszeń awarii konserwator czeka. Należy przetestować działanie aplikacji dla różnych wartości: liczby osób, odstępu pomiędzy sprawdzeniami czy wystąpiła awaria, prawdopodobieństwa awarii oraz zakresu czasu naprawy.

Literatura:

[1] Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D.,
Java Współbieżność dla praktyków, Helion 2007
[2] Horstmann C.S., Cornell G., Java Podstawy, Helion, Wyd. VIII, 2009
[3] Horstmann C.S., Cornell G., Java Techniki zaawansowane, Helion, Wyd. VIII, 2009
[4] Eckel B.: Thinking in Java, Wyd. IV, Helion, 2006.
[5] Bloch J.: Java Efektywne programowanie, Wyd. II, Helion, 2009.
[6] Brackeen D., B. Barker, L. Vanhelsuwe: Java Tworzenie gier, Helion, 2004.
[7] Silberschatz A., Galvin P. B., Gagne G.: Podstawy systemów operacyjnych, WNT, 2005
[8] Dokumentacja JavaDoc 1.6 htp://java.sun.com
[9] Dokumentacja JavaDoc 1.7 htp://java.sun.com
[10] The Java Language Specification, Third Edition,
http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf