

1 Iteratory

”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.”

Który z poniższych przykładów jest poprawny i dlaczego?

```

1 import java.util.Arrays;
2 import java.util.HashSet;
3 import java.util.Set;
4
5 public class TestCollections1 {
6     public static void main(String[] args) {
7         Set<Integer> set = new HashSet<Integer>(new Integer(10));
8         set.addAll(Arrays.asList(new Integer[] { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 }));
9
10        for (Integer i : set)
11            if (i % 2 == 1)
12                set.remove(i);
13
14        for (Integer i : set)
15            System.out.print(i + " ");
16
17    }
18 }

```

Listing 1: Pierwszy przykład użycia iteratora {src/TestCollections1.java}

```

1 import java.util.Arrays;
2 import java.util.HashSet;
3 import java.util.Iterator;
4 import java.util.Set;
5
6 public class TestCollections2 {
7     public static void main(String[] args) {
8         Set<Integer> set = new HashSet<Integer>(new Integer(10));
9         set.addAll(Arrays.asList(new Integer[] { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 }));
10
11        for (Iterator<Integer> it = set.iterator(); it.hasNext(); )
12            if (it.next() % 2 == 1)
13                it.remove();
14
15        for (Integer i : set)
16            System.out.print(i + " ");
17
18    }
19 }

```

Listing 2: Drugi przykład użycia iteratora {src/TestCollections2.java}

2 Klasa Collections

Klasa `java.util.Collections`:

- puste, niezmiennie kolekcje:
 - `public static final List EMPTY_LIST`,
 - `public static final Map EMPTY_MAP`,
 - `public static final Set EMPTY_SET`,
- metody zwracające puste, niezmiennie kolekcje:

- `public static final <T>List<T> emptyList()`
- `public static final <K,V>Map<K,V> emptyMap()`,
- `public static final <T>Set<T> emptySet()`,
- `public static final <K,V>NavigableMap<K,V> emptyNavigableMap()`,
- `public static final <E>NavigableSet<E> emptyNavigableSet()`,
- `public static final <K,V>SortedMap<K,V> emptySortedMap()`,
- `public static final <E>SortedSet<E> emptySortedSet()`,
- metody zwracające niemodyfikowalny widok na kolekcje:
- `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>NavigableMap<K,V> unmodifiableNavigableMap(NavigableMap<K,? extends V> m)`
- `public static <T>NavigableSet<T> unmodifiableNavigableSet(NavigableSet<T> s)`
- `public static <K,V>SortedMap<K,V> unmodifiableSortedMap(SortedMap<K,? extends V> m)`,
- `public static <T>SortedSet<T> unmodifiableSortedSet(SortedSet<T> s)`,
- metody zwracające synchronizowaną kolekcje:
- `public static <T>Collection<T> synchronizedCollection(Collection<T> c)`,
- `public static <T>List<T> synchronizedList(List<T> list)`,
- `public static <K,V>NavigableMap<K,V> synchronizedNavigableMap(NavigableMap<K,V> m)`,
- `public static <T>NavigableSet<T> synchronizedNavigableSet(NavigableSet<T> s)`,
- `public static <K,V>Map<K,V> synchronizedMap(Map<K,V> m)`,
- `public static <T>Set<T> synchronizedSet(Set<T> s)`.

Jakie może być zastosowanie dla pustych kolekcji?

```

1 import java.util.ArrayList;
2 import java.util.Collections;
3 import java.util.List;
4
5 public class TestEmptyCollections {
6     public static void main(String[] args) {
7         /*
8          * Type safety: The expression of type List needs unchecked conversion to
9          * conform to List<String>
10         */
11         List<String> emptyList1 = Collections.EMPTY_LIST;
12         List<String> emptyList2 = Collections.emptyList();
13         List<String> myEmptyList = new ArrayList<String>();
14         System.out.println(emptyList1.equals(myEmptyList));
15         System.out.println(emptyList2.equals(myEmptyList));
16     }
17 }

```

Które z poniższych operacji są dozwolone?

```

1 import java.util.Arrays;
2 import java.util.Collection;
3 import java.util.Collections;
4 import java.util.HashSet;
5 import java.util.Iterator;
6 import java.util.Set;
7
8 public class TestCollections3 {
9
10     public static void main(String[] args) {
11         Set<Integer> collection = new HashSet<Integer>(Arrays.asList(new Integer[] { 1, 2, 3 }));
12
13         Collection<Integer> unmodifiableCollection = Collections.unmodifiableCollection(collection);
14
15         System.out.println("UnmodifiableCollection " + unmodifiableCollection);
16         System.out.println("Collection " + collection);
17     }
18 }

```

```

18     try {
19         Iterator<Integer> it = unmodifiableCollection.iterator();
20         while (it.hasNext()) {
21             it.next();
22             it.remove();
23         }
24     } catch (Exception e) {
25         System.out.println("UnmodifiableCollection: Exception");
26         e.printStackTrace(System.out);
27     }
28
29     try {
30         Iterator<Integer> it = collection.iterator();
31         while (it.hasNext()) {
32             it.next();
33             it.remove();
34         }
35     } catch (Exception e) {
36         System.out.println(" Collection: Exception");
37         e.printStackTrace();
38         System.out.println("UnmodifiableCollection " + unmodifiableCollection);
39         System.out.println(" Collection " + collection);
40     }
41 }
42 }

```

Listing 3: Przykład niemodyfikowalnego widoku na zbiór {src/TestCollections3.java}

”Nadal trzeba stosować blokowanie [2] po stronie klienta, aby móc iterować po kolekcji, podczas gdy inny wątek może ją modyfikować (...)”

Podczas iterowania kolekcji [8] zwróconej z metody `Collections.synchronized???`() 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 iterację.”

” Iteration [1] is often done indirectly by the collection’s `hashCode ()` and `equals ()` methods, which can be called when the collection is inside another collection. The `containsAll ()`, `removeAll ()`, `retainAll ()` methods and constructors that accept collections also iterate. ”

Jak jest różnica w działaniu poniższych programów?

```

1 public class TestCollections {
2     final private static int SIZE = 4;
3     public static void main(String[] args) throws InterruptedException {
4         ExecutorService e = Executors.newCachedThreadPool();
5         for (int i = 0; i < SIZE; i++)
6             e.execute(new Lotto());
7         e.shutdown();
8         e.awaitTermination(1, TimeUnit.DAYS);
9         System.out.println(Lotto.numbers);
10    }
11 }
12 class Lotto implements Runnable {
13     private static List<Integer> privateNumbers = new ArrayList<Integer>();
14     static List<Integer> numbers = Collections.synchronizedList(privateNumbers);
15     @Override
16     public void run() {
17         int i = 0;
18         while (i < 10) {
19             ++i;
20             if (numbers.contains(i) == false)
21                 numbers.add(i);
22         }
23     }
24 }

```

```

1 public class TestCollections4 {
2     final private static int SIZE = 4;
3

```

```

4      public static void main(String[] args) throws InterruptedException {
5          ExecutorService e = Executors.newCachedThreadPool();
6          for (int i = 0; i < SIZE; i++)
7              e.execute(new Lotto());
8          e.shutdown();
9          e.awaitTermination(1, TimeUnit.DAYS);
10         synchronized (Lotto.numbers) {
11             System.out.println(Lotto.numbers);
12         }
13     }
14 }
15
16 class Lotto implements Runnable {
17     private static List<Integer> privateNumbers = new ArrayList<Integer>();
18     static List<Integer> numbers = Collections.synchronizedList(privateNumbers);
19
20     @Override
21     public void run() {
22         int i = 0;
23         while (i < 10) {
24             ++i;
25             synchronized (numbers) {
26                 if (numbers.contains(i) == false)
27                     numbers.add(i);
28             }
29         }
30     }
31 }

```

Listing 4: Przykład synchronizowanej listy {src/TestCollections4.java}

3 Kolekcje bezpieczne wątkowo

”W pakiecie `java.util.concurrent`[2] 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,`
- `public class ConcurrentLinkedDeque<E> extends AbstractCollection<E> implements Deque<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` to tabela haszowana wspierająca pełną współbieżność pobierania i wysoką współbieżność aktualizacji. Tablica jest dynamicznie rozszerzana, gdy występuje zbyt wiele kolizji, która występuje gdy klucze, które mają różne `hashCode()`, ale są umieszczane w tym samym miejscu modulo rozmiar tabeli. Wszystkie operacje są bezpieczne dla wątków. Operacje pobierania nie pociągają za sobą blokowania i nie możliwości zablokowania całej tabeli w sposób uniemożliwiający wszelki dostęp.

Operacje pobierania (w tym `get`) zazwyczaj nie blokują się, więc mogą nakładać się na operacje aktualizacji (w tym operacje wstawiania i usuwania). Pobieranie odzwierciedla wyniki ostatnio zakończonych operacji aktualizacji, które miały miejsce w momencie ich rozpoczęcia. Ta klasa jest zgodna z tą samą specyfikacją co do funkcjonalności, co klasa `Hashtable`, ale realizuje to winny sposób.

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]:

```

1  if (!map.containsKey(key))
2      return map.put(key, value);
3  else
4      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]:

```

1  if (map.containsKey(key) && map.get(key).equals(value)) {
2      map.remove(key);
3      return true;
4  } 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]:

```

1  if (map.containsKey(key)) {
2      return map.put(key, value);
3  } 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]:

```

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

```

```

1  class SleepyHead {
2      private int i;
3      private boolean sleep;
4      SleepyHead(int i) {
5          this.i = i;
6      }
7      SleepyHead(int i, boolean sleep) {
8          this(i);
9          this.sleep = sleep;
10     }
11     @Override
12     public boolean equals(Object o) {
13         try {
14             if (sleep)
15                 TimeUnit.SECONDS.sleep(2);
16         } catch (InterruptedException e) {
17             e.printStackTrace();
18         }
19         return super.equals(o);
20     }
21     @Override
22     public int hashCode() {
23         return i;
24     }
25     @Override
26     public String toString() {
27         return "" + i;
28     }
29 }
30 class ThreadHelper {
31     static void printAllThreadInfo() {
32         ThreadMXBean threadMXBean = ManagementFactory.getThreadMXBean();
33         boolean objectMonitor = threadMXBean.isObjectMonitorUsageSupported();
34         boolean synchronizer = threadMXBean.isSynchronizerUsageSupported();
35         System.out.println("objectMonitorUsageSupported = " + objectMonitor);
36         System.out.println("synchronizerUsageSupported = " + synchronizer);
37         ThreadInfo[] threads = threadMXBean.dumpAllThreads(
38             objectMonitor, synchronizer);
39         for (ThreadInfo t : threads)
40             System.out.print(t);
41     }
42 }
43
44 public class TestCollections {
45     static int SIZE = 4;
46     static Map<SleepyHead, Integer> map = new ConcurrentHashMap<>();
47
48     public static void main(String[] args) throws InterruptedException {
49         for (int i = 0; i < 1000; i++)

```

```

50         addInitialDataToMap(i);
51         long t1 = System.nanoTime();
52         ExecutorService e = Executors.newCachedThreadPool();
53         for (int i = 0; i < 4; i++)
54             addTestDataToMapUsingExecutor(e, i);
55         e.shutdown();
56         TimeUnit.SECONDS.sleep(1);
57         ThreadHelper.printAllThreadInfo();
58         e.awaitTermination(1, TimeUnit.DAYS);
59         long t2 = System.nanoTime();
60         System.out.println("Time: " + (t2 - t1) / 1000 / 1000);
61     }
62
63     public static void addInitialDataToMap(int i) {
64         map.put(new SleepyHead(i), i);
65     }
66
67     public static void addTestDataToMapUsingExecutor(ExecutorService e, int i) {
68         e.execute(() -> {map.put(new SleepyHead(i, true), i);});
69     }
70 }

```

Klasa `ConcurrentHashMap.KeySetView<K,V>` implements `Set<K>` jest widokiem w postaci zbioru na `ConcurrentHashMap`. Wykonanie operacji dodawania może być opcjonalnie umożliwione poprzez użycie "common value".

```

1  import java.util.concurrent.ConcurrentHashMap;
2  import java.util.concurrent.ConcurrentHashMap.KeySetView;
3
4  public class TestCollections5 {
5
6      public static void main(String[] args) {
7          ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();
8          map.put(1, "one");
9          map.put(2, "two");
10         KeySetView<Integer, String> view = map.keySet("ABC");
11         System.out.println("Map : " + map);
12         System.out.println("View: " + view);
13         view.remove(1);
14         System.out.println("Map : " + map);
15         System.out.println("View: " + view);
16         System.out.println("MappedValue: " + view.getMappedValue());
17         view.add(3);
18         System.out.println("Map : " + map);
19         System.out.println("View: " + view);
20     }
21 }
22

```

4 Tablice kopiowane przy zapisie

"CopyOnWriteArrayList i CopyOnWriteArraySet to bezpieczne wątkowo kolekcje [2], 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?

```

1  import java.util.ArrayList;
2  import java.util.Iterator;
3  import java.util.List;
4
5  public class TestCollections6 {
6      public static void main(String[] args) {
7          List<Integer> list = new ArrayList<Integer>();
8          list.add(1);
9          list.add(2);
10
11          try {
12              Iterator<Integer> it = list.iterator();

```

```

13         while (it.hasNext()) {
14             System.out.print(it.next() + ", ");
15             list.add(3);
16         }
17     } catch (Exception e) {
18         e.printStackTrace(System.out);
19     }
20     System.out.println();
21
22     for (int i : list)
23         System.out.print(i + ", ");
24 }
25 }

```

```

1 import java.util.Iterator;
2 import java.util.List;
3 import java.util.concurrent.CopyOnWriteArrayList;
4
5 public class TestCollections7 {
6     public static void main(String[] args) {
7         List<Integer> list = new CopyOnWriteArrayList<Integer>();
8         list.add(1);
9         list.add(2);
10
11         try {
12             Iterator<Integer> it = list.iterator();
13             while (it.hasNext()) {
14                 System.out.print(it.next() + ", ");
15                 list.add(3);
16             }
17         } catch (Exception e) {
18             e.printStackTrace(System.out);
19         }
20         System.out.println();
21
22         for (int i : list)
23             System.out.print(i + ", ");
24     }
25 }

```

5 Kolejki nieblokujące

Interfejs `java.util.Queue` extends `Collection<E>`:

	Zgłasza wyjątek	Zwraca specjalną wartość
Insert	<code>add(e)</code>	<code>offer(e)</code>
Remove	<code>remove()</code>	<code>poll()</code>
Examine	<code>element()</code>	<code>peek()</code>

Źródło: `java.util.Queue`, Java Development Kit, version 15

Interfejs `java.util.Dequeue` extends `Queue<E>`:

	Pierwszy element (head)		Ostatni element (tail)	
	Zgłasza wyjątek	Zwraca specjalną wartość	Zgłasza wyjątek	Zwraca specjalną wartość
Insert	<code>addFirst(e)</code>	<code>offerFirst(e)</code>	<code>addLast(e)</code>	<code>offerLast(e)</code>
Remove	<code>removeFirst()</code>	<code>pollFirst()</code>	<code>removeLast()</code>	<code>pollLast()</code>
Examine	<code>getFirst()</code>	<code>peekFirst()</code>	<code>getLast()</code>	<code>peekLast()</code>

Źródło: `java.util.Dequeue`, Java Development Kit, version 15

6 Kolejki blokujące

Interfejs `BlockingQueue<E>` extends `Queue<E>`

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

Źródło: `java.util.concurrent.BlockingQueue`, Java Development Kit, version 15

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

Klasy implementujące interfejs `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 LinkedBlockingQueue<E> extends AbstractQueue<E> implements BlockingQueue<E>, Serializable,`
- `public class LinkedTransferQueue<E> extends AbstractQueue<E> implements TransferQueue<E>, Serializable,`
- `public class PriorityBlockingQueue<E> extends AbstractQueue<E> implements BlockingQueue<E>, Serializable,`
- `public class SynchronousQueue<E> extends AbstractQueue<E> implements BlockingQueue<E>, Serializable.`

Interfejsy rozszerzające `BlockingQueue`:

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

Interfejs `BlockingDeque<E> extends BlockingQueue<E>, Deque<E>`

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

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

Źródło: `java.util.concurrent.BlockingDeque`, Java Development Kit, version 15

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

Klasy implementujące interfejs `BlockingDeque`:

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

Interfejs `java.util.TransferQueue` extends `BlockingQueue<E>`

Metody:

- `int getWaitingConsumerCount()` - zwraca oszacowaną liczbę konsumentów oczekujących na otrzymanie elementu (metody `take()` lub czasowego `poll()`),
- `boolean hasWaitingConsumer()` - zwraca `true` jeżeli istnieje co najmniej jeden konsument oczekujący na otrzymanie elementu (metody `take()` lub czasowego `poll()`),
- `void transfer(E e)` - przesyła element do konsumenta (oczekując jeżeli to konieczne),
- `boolean tryTransfer(E e)` - niezwłocznie przesyła element do oczekującego konsumenta (jeżeli jest to możliwe),
- `boolean tryTransfer(E e, long timeout, TimeUnit unit)` - przesyła element do oczekującego konsumenta jeżeli jest to możliwe przed upływem podanego czasu.

Klasy implementujące interfejs `TransferQueue`:

- `public class LinkedTransferQueue<E> extends AbstractQueue<E> implements TransferQueue<E> Serializable.`

```

1  import java.util.concurrent.LinkedBlockingQueue;
2  import java.util.concurrent.TimeUnit;
3
4  class Producer implements Runnable {
5      private boolean sleep;
6      private int i;
7
8      Producer(boolean sleep) {
9          this.sleep = sleep;
10     }
11
12     @Override
13     public void run() {
14         try {
15             while (true)
16                 produce();
17         } catch (InterruptedException e) {
18             e.printStackTrace();
19         }
20     }
21
22     final void produce() throws InterruptedException {
23         System.out.println("> " + ++i);
24         TestCollections8.queue.put(i);
25         System.out.println(">> " + i);
26         if (sleep)
27             TimeUnit.SECONDS.sleep(2);
28     }
29 }
30
31 class Consumer implements Runnable {
32     private boolean sleep;
33     private int i;
34
35     Consumer(boolean sleep) {
36         this.sleep = sleep;
37     }
38
39     @Override
40     public void run() {
41         try {
42             while (true)
43                 consume();
44         } catch (InterruptedException e) {
45             e.printStackTrace();
46         }
47     }
48
49     final void consume() throws InterruptedException {
50         System.out.println("< " + ++i);
51         int tmp = TestCollections8.queue.take();
52         System.out.println("<< " + i + ":" + tmp);
53         if (sleep)

```

```

54         TimeUnit.SECONDS.sleep(2);
55     }
56 }
57
58 public class TestCollections8 {
59     static LinkedBlockingQueue<Integer> queue = new LinkedBlockingQueue<Integer>(4);
60
61     public static void main(String[] args) throws Exception {
62         new Thread(new Producer(false)).start();
63         new Thread(new Consumer(true)).start();
64     }
65 }

```

Listing 5: Przykład użycia LinkedBlockingQueue {src/TestCollections8.java}

7 Inne kolekcje

Inne kolekcje bezpieczne wątkowo [2]:

- java.util.Vector<E >,
- java.util.Hashtable<K,V >.

8 Porównanie

```

1  enum MapType{
2      SynchronizedHashMap(Collections.synchronizedMap(new HashMap<>()), 1_000_000),
3      SynchronizedTreeMap(Collections.synchronizedMap(new TreeMap<>()), 1_000_000),
4      ConcurrentSkipListMap(new ConcurrentSkipListMap<>(), 1_000_000),
5      ConcurrentHashMap(new ConcurrentHashMap<>(), 1_000_000),
6      Hashtable(new Hashtable<>(), 1_000_000);
7
8      private Map<Integer, Integer> map;
9      private int loops;
10
11      MapType(Map<Integer, Integer> map, int loops){
12          this.map = map;
13          this.loops = loops;
14      }
15      Map<Integer, Integer> getMap(){
16          return map;
17      }
18      public int getLoops() {
19          return loops;
20      }
21 }
22
23 public class CompareMaps {
24
25     final static int SIZE = 4;
26
27     public static void main(String[] args) throws InterruptedException {
28         for(MapType mapType : MapType.values()) {
29             System.gc();
30             testMap(mapType);
31         }
32     }
33
34     public static void testMap(MapType mapType) throws InterruptedException {
35         Map<Integer, Integer> map = mapType.getMap();
36         long t1 = System.nanoTime();
37
38         ExecutorService e = Executors.newCachedThreadPool();
39         for (int i = 0; i < SIZE; i++) {
40             e.execute(()->{for(int j = 0; j < mapType.getLoops(); j++) map.put(j, j)});
41         }
42
43         e.shutdown();
44         e.awaitTermination(1, TimeUnit.DAYS);
45         long t2 = System.nanoTime();
46         System.out.println(mapType + " " + map.getClass() + " Time: " + (t2 - t1) / 1000 / 1000);
47     }
48 }

```

```

1  enum CollectionType{
2      SynchronizedArrayList(Collections.synchronizedList(new ArrayList<>()), 1_000_000),
3      SynchronizedLinkedList(Collections.synchronizedList(new LinkedList<>()), 1_000_000),
4      SynchronizedHashSet(Collections.synchronizedSet(new HashSet<>()), 1_000_000),

```

```

5      SynchronizedTreeSet(Collections.synchronizedSet(new TreeSet<>()), 1_000_000),
6      ConcurrentSkipListSet(new ConcurrentSkipListSet<>(), 1_000_000),
7      ConcurrentLinkedQueue(new ConcurrentLinkedQueue<>(), 1_000_000),
8      ConcurrentLinkedDeque(new ConcurrentLinkedDeque<>(), 1_000_000),
9      Vector(new Vector<>(), 1_000_000),
10     CopyOnWriteArraySet(new CopyOnWriteArraySet<>(), 100_000),
11     CopyOnWriteArrayList(new CopyOnWriteArrayList<>(), 100_000);
12
13     private Collection<Integer> collection;
14     private int loops;
15
16     CollectionType(Collection<Integer> collection, int loops){
17         this.collection = collection;
18         this.loops = loops;
19     }
20     Collection<Integer> getCollection(){
21         return collection;
22     }
23     public int getLoops() {
24         return loops;
25     }
26 }
27
28 public class CompareCollections {
29
30     final static int SIZE = 4;
31
32     public static void main(String[] args) throws InterruptedException {
33         for(CollectionType collectionType : CollectionType.values()) {
34             testCollection(collectionType);
35         }
36     }
37
38     public static void testCollection(CollectionType collectionType) throws InterruptedException {
39         Collection<Integer> collection = collectionType.getCollection();
40         long t1 = System.nanoTime();
41
42         ExecutorService e = Executors.newCachedThreadPool();
43         for (int i = 0; i < SIZE; i++) {
44             e.execute(()->{for(int j = 0; j < collectionType.getLoops(); j++) collection.add(j);});
45         }
46
47         e.shutdown();
48         e.awaitTermination(1, TimeUnit.DAYS);
49         long t2 = System.nanoTime();
50         System.out.println(collectionType + " " + collection.getClass() + " Time: " + (t2 - t1) / 1000 / 1000);
51     }
52 }

```

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

Materiały do przedmiotu dostępne są na stronie: <http://weaii-moodle.tu.kielce.pl/>