Advanced Java Programming Course

Java Synchronized Collections

Faculty of Information Technologies
Industrial University of Ho Chi Minh City

Introduction

- By default, Java collections are not synchronized
  - Thus, they are not thread-safe
- Thread safe: Able to be used concurrently by multiple threads.
  - Many of the Java library classes are not thread safe!
  - In other words, if two threads access the same object, things break.
- Examples:
  - ArrayList and other collections from java.util are not thread safe; two threads changing the same list at once may break it.
  - StringBuilder is not thread safe.
  - Java GUIs are not thread safe; if two threads are modifying a GUI simultaneously, they may put the GUI into an invalid state.

Overview of Java Synchronized Collections

- Java provides thread-safe collection wrappers via static methods in the Collections class:
- Method

  Collections.synchronizedCollection (coll)

  Collections.synchronizedList (list)

  Collections.synchronizedMap (map)

  Collections.synchronizedSet (set)

  ```
  Set<String> words = new HashSet<String>();
  words = Collections. synchronizedSet (words);
  ```
- These are essentially the same as wrapping each operation on the collection in a synchronized block.
  - Simpler, but not more efficient, than the preceding code.

Overview of Java Synchronized Collections

- Example: Multiple threads can thus access & update the synchronized collection

```
Map<Integer, String> map = new HashMap<>();

Runnable task1 = () -> {
        map.put(i++, "A");
};

 Runnable task2 = () -> {
     map.put(i++, "B");
 };

ExecutorService executorService = java.util.concurrent.Executors.newCachedThreadPool();
for (int i = 0; i < 10; i++) {
    executorService.execute(task1);
    executorService.execute(task2);
}
```

This implementation is not synchronized

Overview of Java Synchronized Collections

◊ Example: Multiple threads can thus access & update the synchronized collection

```java
Map<Integer, String> map = Collections.synchronizedMap(new HashMap<>());

AtomicInteger atomicInteger = new AtomicInteger( initialValue: 0);
Runnable task1 = () -> {
        map.put(atomicInteger.incrementAndGet(), "A");
};

 Runnable task2 = () -> {
        map.put(atomicInteger.incrementAndGet(), "B");
 };

ExecutorService executorService = java.util.concurrent.Executors.newCachedThreadPool();
for (int i = 0; i < 10; i++) {
    executorService.execute(task1);
    executorService.execute(task2);
}
```

Multiple threads can thus access & update the synchronized collection

Concurrent collections

- New package java.util.concurrent contains collections that are optimized to be safe for use by multiple threads:

```
class ConcurrentHashMap<K, V> implements Map<K, V>
class ConcurrentLinkedDeque<E> implements Deque<E>
class ConcurrentSkipListSet<E> implements Set<E>
class CopyOnWriteArrayList<E> implements List<E>
```

- These classes are generally faster than using a synchronized version of the normal collections because multiple threads are actually able to use them at the same time, to a degree.

Concurrent collections

- Example: Multiple threads can thus access & update the concurrent collection

```
Map<Integer, String> map = new ConcurrentHashMap<>();
  AtomicInteger i = new AtomicInteger( initialValue: 0);

Runnable task1 = () -> {
        map.put(i.incrementAndGet(), "A");
};

  Runnable task2 = () -> {
      map.put(i.incrementAndGet(), "B");
  };

 ExecutorService executorService = java.util.concurrent.Executors.newCachedThreadPool();
 for (int j = 0; j < 10; j++) {
     executorService.execute(task1);
     executorService.execute(task2);
 }
```

Multiple threads can thus access & update the synchronized collection

Java BlockingQueue

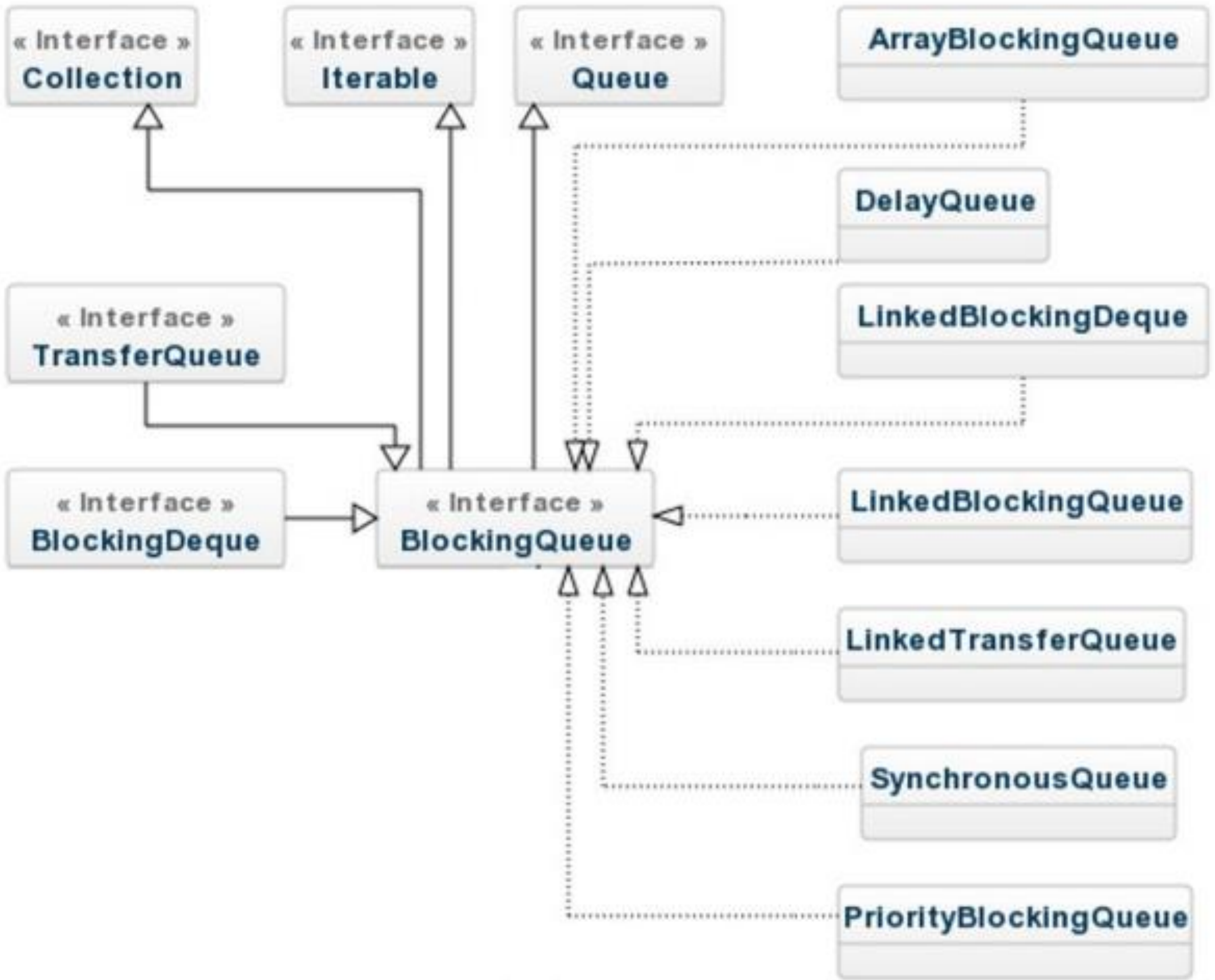
- A Queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.
- Summary of BlockingQueue methods

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`element()`	`peek()`	*not applicable*	*not applicable*

Java BlockingQueue

- A BlockingQueue does not accept null elements.
- A BlockingQueue may be capacity bounded.
- BlockingQueue implementations are designed to be used primarily for producer-consumer queues, but additionally support the Collection interface.
- BlockingQueue implementations are thread-safe.
- A BlockingQueue does not intrinsically support any kind of "close" or "shutdown" operation to indicate that no more items will be added.

Java BlockingQueue



ArrayBlockingQueue

- ArrayBlockingQueue class is Java concurrent and bounded blocking queue implementation backed by an array. It orders elements FIFO (first-in-first-out).
- The head of the ArrayBlockingQueue is that element that has been on the queue the longest time.
- The tail of the ArrayBlockingQueue is that element that has been on the queue for the shortest time.
- New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.

ArrayBlockingQueue Features

- ArrayBlockingQueue is a bounded queue of fixed size backed by an array.
- Once created, the capacity of the queue cannot be changed.
- It supplies blocking insertion and retrieval operations.
- It does not allow NULL objects.
- ArrayBlockingQueue is thread-safe.